Chapter 1

Introduction

Online Educational Systems is an emerging technology nowadays. It started with school level education for courses like Maths, English, etc. Presently, online courses are available for many technical and advanced studies.

Earlier, 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. Online Educational Systems are generally interactive in nature. Students can be provided with questions for practice which can be evaluated by the system itself. This reduces the overhead of instructor and hence allows accomodation of a large number of students in the course. Problems can also be generated by such systems. Some systems also provide feedback to the students after evaluating their answers.

In this thesis, we propose an Online Educational System for Compilers. This has been created for the parsing phase of Compilers. The results of which have been detailed in Chapter.

1.1 Objective

The purpose of this Thesis is to create an Online Tutoring System for 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

The system is divided into three parts. Figure 1.1 gives an abstract view of the system.

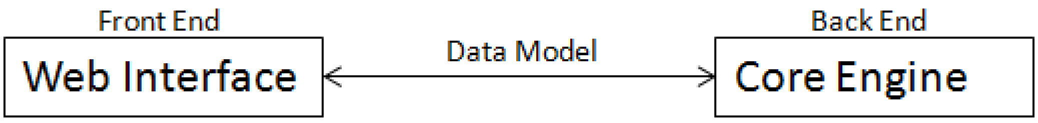

Figure 1.1: Framework of the System

The system is mainly consists of:

- **Core Engine:** This part of the system deals with generation of problems, evaluation the answers given by students and generating hint questions automatically based on the answer given by students.

- **Data Model:** It acts as an interface between front-end and back-end. It is used to transfer data back-and-forth between core engine and web interface.

- **Web Interface:** This part provides a user-friendly environment to the student. It deals with the presentation of problems and provides ease of giving solutions.

1.4 Contribution of Thesis

In this thesis, we have implemented a tool for the parsing phase of program compilation. It generates problems automatically for the different techniques involved in this phase, i.e. First set, Follow set, LL Parsing (which includes LL Parsing Table and LL Parsing Moves) and 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 to make the student 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,

Chapter 3,

Chapter 4,

Chapter 5,

Chapter 6,

Chapter 2

Background

This chapter provides an overview of the concepts required to understand this thesis. Brief explanations about concepts in Compiler Design and Intelligent Tutoring Systems, are covered here [**aho1977principles**].

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.

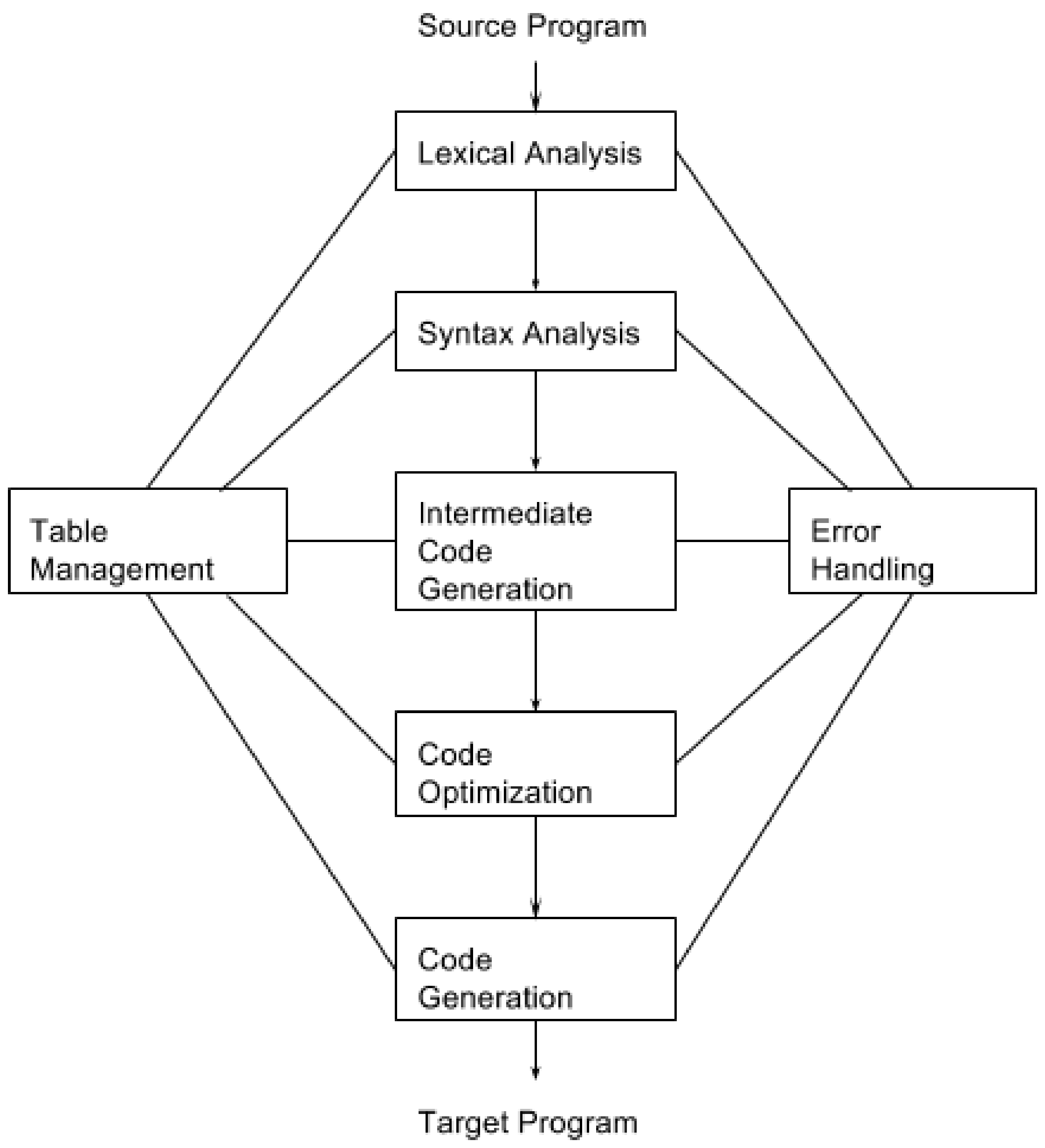

Figure 2.1: Phases of Compiler

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 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. The 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.

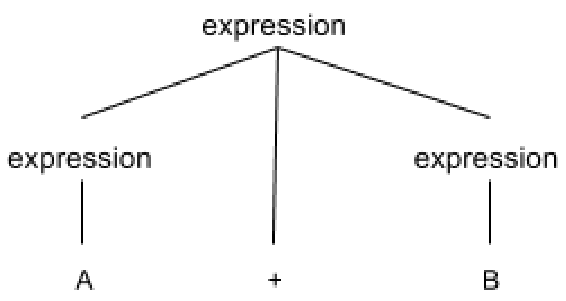

Figure 2.2: Parse Tree for A+B

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

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

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:*

   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

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:*

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 = $\{\{\epsilon$, a, b, aa, bb, ab, ba$\} \mid \{$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:*

$S \to aBSc$

$S \to abc$

$Ba \to aB$

$Bb \to bb$

- **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:*

  $S \to aSb$

  $S \to ab$

- **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 $A \rightarrow \beta$ 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 $S \overset{*}{\Rightarrow} \alpha A w \Rightarrow \alpha\beta w$, then $A \rightarrow \beta$ 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 $Y_1Y_2....Y_{i-1} \overset{*}{\Rightarrow} \epsilon$), 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,

$S \overset{*}{\Rightarrow} \alpha A a \beta$ 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., $\beta \overset{*}{\Rightarrow} \epsilon$), 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.

Example 2.12 *If the grammar is:*

$S \rightarrow cAd$

$A \rightarrow a \mid ab$

The 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

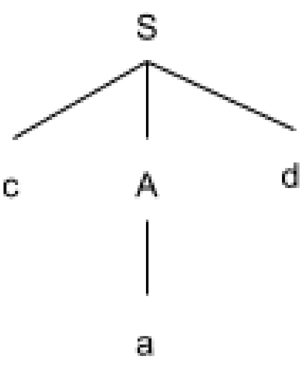

Figure 2.3: Parse Tree for string cad

that there is a derivation $A \overset{+}{\Rightarrow} A\alpha$ ($\overset{+}{\Rightarrow}$ 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 $\rightarrow$ 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 $A\rightarrow\ \alpha\beta\ /\ \alpha\gamma$ are two productions in the grammar, then after left-factoring, the production rules are of the form:*

$A \rightarrow \alpha A'$

$A' \rightarrow \beta\ /\ \gamma$

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.

Algorithm 1 Construction of LL Parsing Table

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

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:*

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

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.

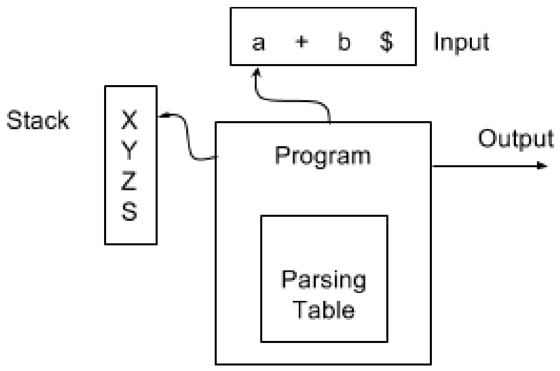

Figure 2.4: Model of a predictive parser

2. If X = a ≠ $, 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 → 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 → T E'*
$\$E'0$	$0 + 1\$$	*T → 0*
$\$E'$	$+1\$$	
$\$E+$	$+1\$$	*E' → + E*
$\$E$	$1\$$	
$\$E'T$	$1\$$	*E → T E'*
$\$E'1$	$1\$$	*T → 1*
$\$E'$	$\$$	
$\$$	$\$$	*E'* $\to \epsilon$

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” $\rightarrow$ .E

  E $\rightarrow$ .TE’

  T $\rightarrow$ .1

  T $\rightarrow$ .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* $\{[E'' \to .E], [E \to .TE'], [T \to .0], [T \to .1]\}$*, 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'.

Algorithm 2 Canonical collection of sets of SLR items Construction

```
1: C = CLOSURE($S' \to .S$)
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
```

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*

I0: T → .1

T → .0

E" → .E

E → .TE'

I1: E → T.E'

E' → .

E' → .+E

I2: T → 1.

I3: T → 0.

I4: E" → E.

I5: T → .1

E → .TE'

E' → +.E

T → .0

I6: E → TE'.

I7: E' → +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.

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].
```

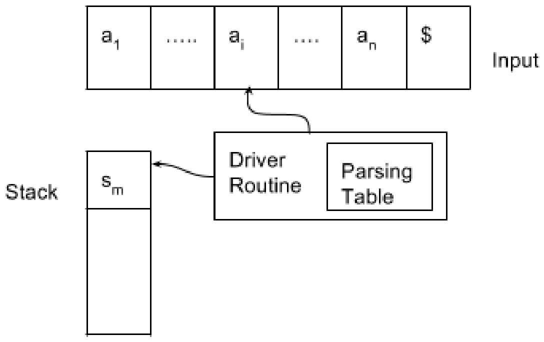

Figure 2.5: Model of a SLR parser

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*			

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.
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

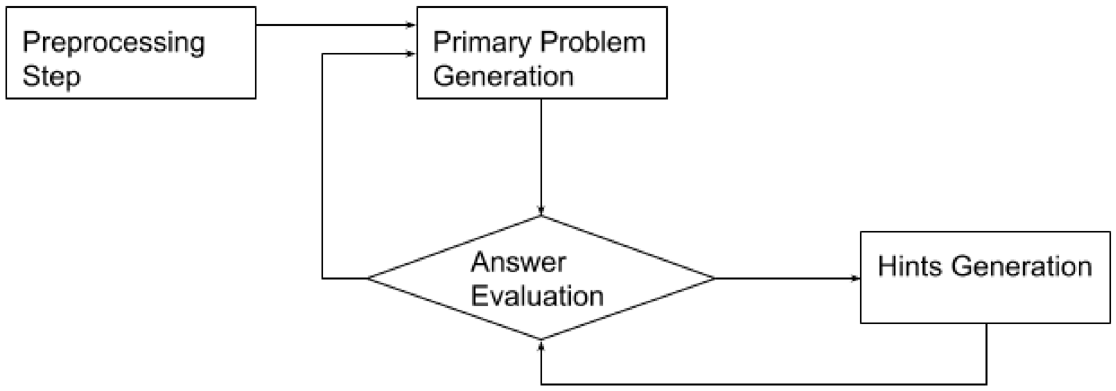

Figure 3.1: Core Engine

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, 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. They are of the form:*

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

Options: *terminals in the grammar*

For this grammar, non-terminals are E', E and T, and the terminals are 1, 0 and +.

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 *Suppose the grammar used in example 2.10, then 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.

Example 3.3 *Which grammar rules should be included in the highlighted cell of the LL parsing table ?*

Options: *T → 0   T → 1   E → T E'   E' → ϵ   E' → + E   ERROR*

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

Which grammar rules should be included in the highlighted cell of the LL parsing table ?

Options: *T → 0   T → 1   E → T E'   E' → ϵ   E' → + E   ERROR*

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

Which grammar rules should be included in the highlighted cell of the LL parsing table ?

Options: *T → 0   T → 1   E → T E'   E' → ϵ   E' → + E   ERROR*

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

Which grammar rules should be included in the highlighted cell of the LL parsing table ?

Options: *T → 0   T → 1   E → T E'   E' → ϵ   E' → + E   ERROR*

	0	***1***	***+***	***$***
T		*?*	*ERROR*	*ERROR*
E	*E → T E'*		*ERROR*	*ERROR*
E'	*ERROR*	*ERROR*	*E'→ + 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 data structure 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. This implies that the tool takes into account, the incorrect options which are marked in the solution as well as the options in the correct solution 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 3.4 *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 3.5 *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 3.6 *For the correct option '0' left out by the user in example 3.5, 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 3.7 *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' $\rightarrow \epsilon$, then the system will generate hint question as:*

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

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

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

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

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 as depicted below. These algorithms depict 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 helpers 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:         $answered \leftarrow false$
6:         while $answered = false$ do
7:             $answered \leftarrow$ 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:                 $H1 \leftarrow$ GENERATE__HINT__QUESTION($choice, P, D, 1$)
11:                 EVALUATE__HINT__QUESTION($H1$)
12:             else if choice∈ C then
13:                 $C \leftarrow C - \{choice\}$
14:             end if
15:         end for
16:         if C≠ ∅ then
17:             for all choice ∈ C do
18:                 $H2 \leftarrow$ GENERATE__HINT__QUESTION($choice, P, D, 2$)
19:                 EVALUATE__HINT__QUESTION($H2$)
20:                 $H1 \leftarrow$ 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.

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 10 Preprocessing for First

```
function PREPROCESS_FIRST($G$)
    N := set of all non-terminals in grammar G
    S := table for storing all first sets
    for all n in N do
        F := { t | t$\in$ first(n)}
        S[n] = F
    end for
    return S
end function
```

Algorithm 11 Primary problem generation for First

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

Algorithm 12 Hint question generation for First

```
function GENERATE_HINT_QUESTION_FIRST($choice, P, D, type, context$)
    if type = 1 then
        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."
        if choice$\in$ I then
            A := 4
        else if choice$\in$ C then
            A := 1 or 2 or 3
        end if
    else if type = 2 then
        H := "Should choice be included in FIRST[context] ?"
        A := "Yes"
    end if
    return (H, A)
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.

Algorithm 15 Hint question generation for Follow

```
function GENERATE_HINT_QUESTION_FOLLOW($choice, type, context$)
    if type = 1 then
        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.”
        if choice∈ I then
            A := 4
        else if choice∈ C then
            A := 1 or 2 or 3
        end if
    else if type = 2 then
        H := ”Should choice be included in FOLLOW[context] ?”
        A := ”Yes”
    end if
    return (H, A)
end function
```

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:

- **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 for that cell entry. 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 4 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.

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 := choose random cell index 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
```

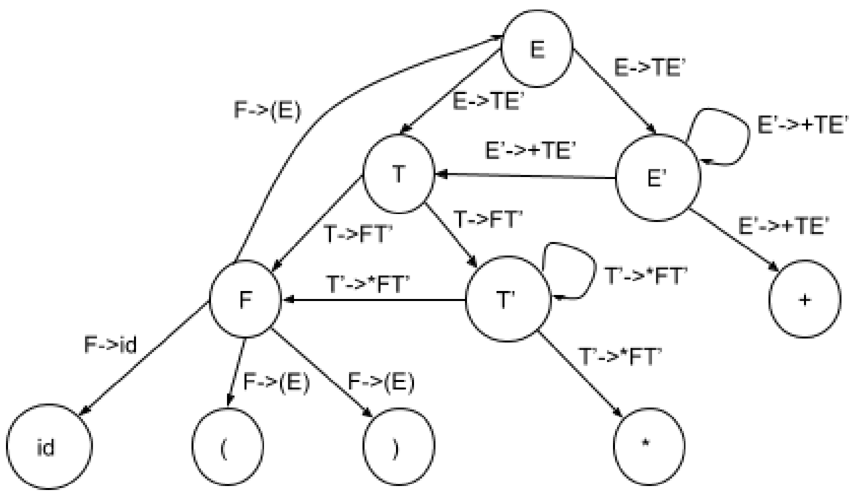

Figure 4.1: Graph generated on applying algorithm 23

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:

Example 4.1 *Suppose the grammar is*

E $\rightarrow$ *T E'*

E' $\rightarrow$ *+ T E' /* ϵ

T $\rightarrow$ *F T'*

T' $\rightarrow$ ** F T' /* ϵ

F $\rightarrow$ *(E) / id*

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 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 Algorithm. The path in the tree is shown in Figure 4.3.

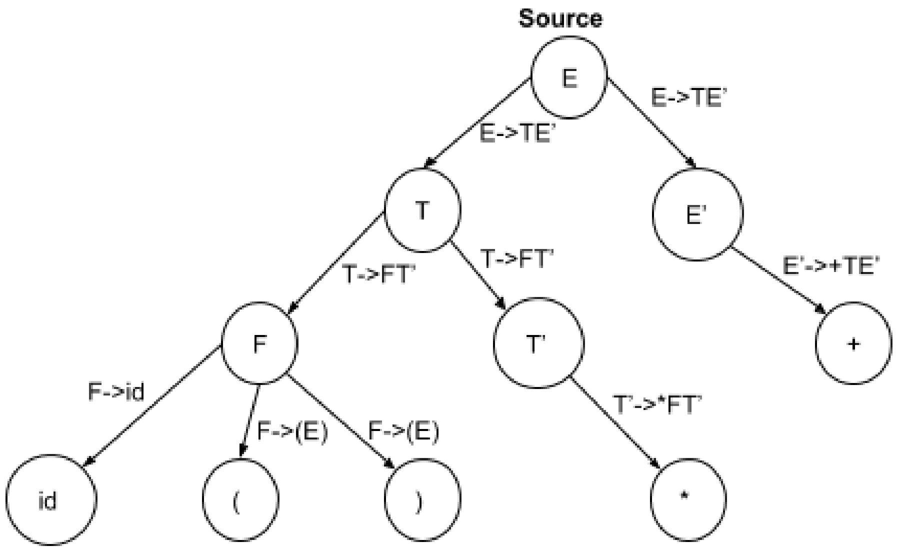

Figure 4.2: Tree after applying Dijkstra Algorithm first time

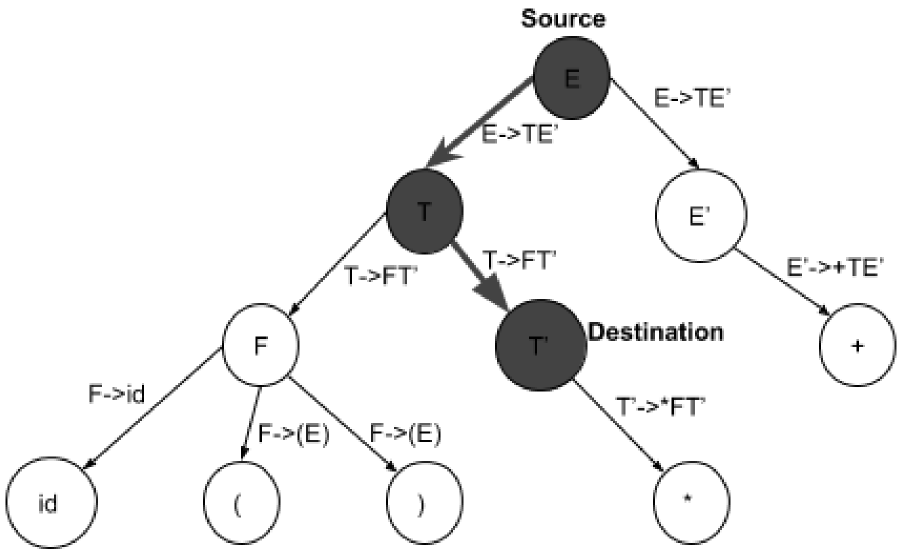

Figure 4.3: Path in the tree from E to T'

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,

- *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	
$E'T	
$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*

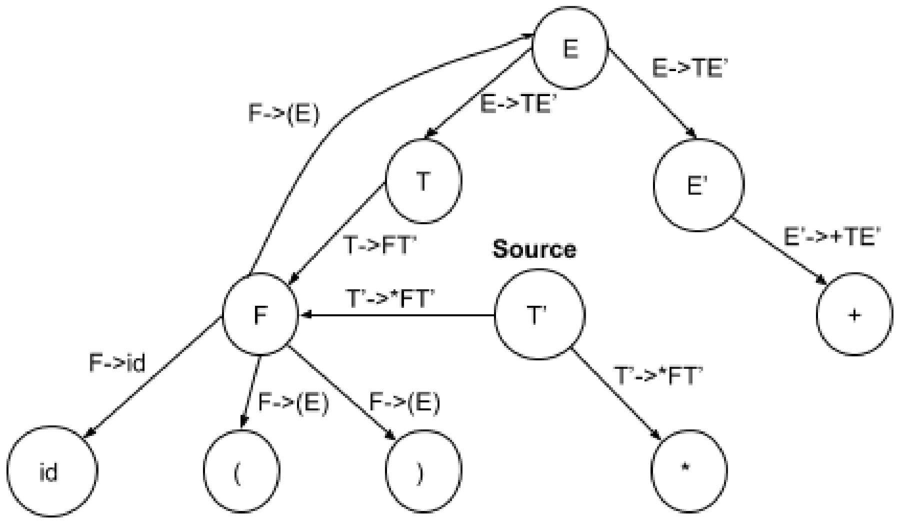

Figure 4.4: Tree after applying Dijkstra Algorithm second time

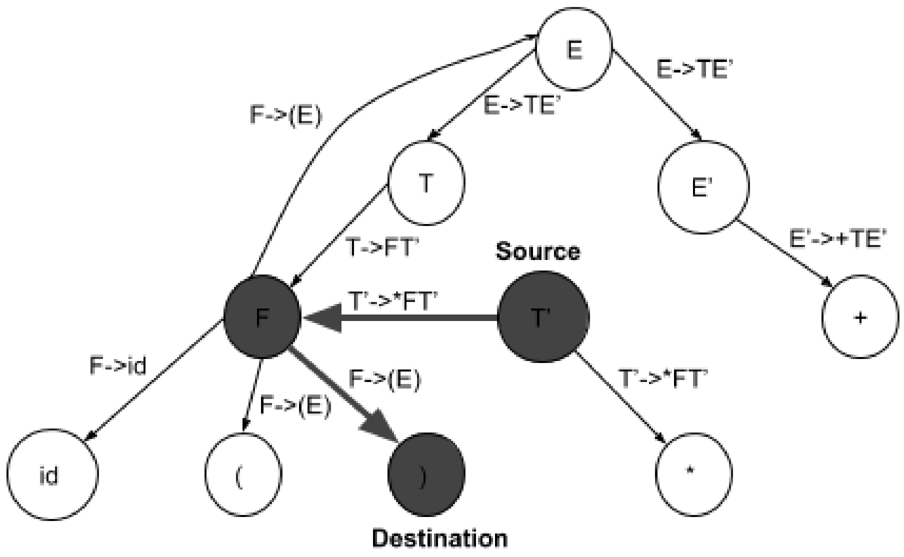

Figure 4.5: Path in the tree from T' to)

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, the node corresponding 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.*

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'ϵ	*id * (id*
\$E'T')E'	*id * (id*
\$E'T')ϵ	*id * (id*
\$E'T')	*id * (id*
\$E'T'	*id * (id)*
\$E'ϵ	*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 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

```
function EMPTY_STACK($G, STACK, PARTIAL, MINRULES$)
    NT := set of all non-terminals in G
    while TOP($STACK$)$\neq$\$ do
        top = POP($STACK$)
        if top$\in$NT then
            minr = REVERSE($MINRULES[top]$)    ▷ reverses the minimum length rule for top
            PUSH($STACK, minr$)
        else
            PARTIAL = APPEND($PARTIAL, top$)
        end if
    end while
    return PARTIAL
end function
```

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

```
function MINRULES_GEN($G$)
    RULES := table containing rules for every non-terminal in G
    N := set of all non-terminals in G
    MINRULES := table containing all minimum length rules of every n$\in$N
    for all n in N do
        R = RULES[n]                      ▷ set containing all rules of n
        Rm = MIN($R$)
        MINRULES[n] = Rm
    end for
    return MINRULES
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:                 if $LABELS[(n, symbol)] = null$ || LENGTH($rule$) < LENGTH($LABELS[(n, symbol)]$) then
16:                     LABELS[(n,symbol)] = rule
17:                 end if
18:             end for
19:         end for
20:     end for
21:     return (E,V, LABELS)
22: end function
```

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 **??**.

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
```

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, 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 of chapter 2. Questions are generated on concepts involving item sets (described in 2.1.2.3). These questions