ParseIT: A Tool for Teaching Parsing Techniques

Nimisha Agrawal  
Indian Institute of Technology Kanpur  
Kanpur, UP, India  
nimisha@cse.iitk.ac.in

Amey Karkare  
Indian Institute of Technology Kanpur  
Kanpur, UP, India  
karkare@cse.iitk.ac.in

ABSTRACT

We describe ParseIT, a tool to help students understand the concept of parsing in an undergraduate Compiler Design course. Given a grammar, the tool generates multiple choice questions for concepts related to parsing. The tool evaluates students’ attempts and provide hints for incorrect attempts. The hints are also generated in the form of specific questions about any correct choice that is missed or any incorrect choice that is selected. For grammars that can be parsed through the textbook LL or LR algorithms, an interesting form of hint generated is an input string that helps student identify incorrectly filled cells of a parsing table. The tool enables interactive and active learning of parsing concepts, resulting in a better understanding.

Keywords

Intelligent Tutoring; Education; Programming; Compilers

1. INTRODUCTION

Recent development in technologies has enabled institutions to offer courses to large number of students. These Massive open online courses (MOOCs) [3, 4, 7] digitize the contents of the topics (lecture videos, notes etc), and allow students to access the contents beyond physical boundaries of classrooms. The increase in number of students has added challenges for the instructor for the tutoring aspects, such as creation of new problems for assignments, solving these problems, grading, and helping students master a concept through hands-on exercises [13]. These challenges have prompted researchers to develop automated tutoring systems to help the student to explore a course based on his skills and learning speed [10, 15, 14].

Compiler design is an important subject in the computer science curriculum for undergraduates. Compilers are one of the success stories of Computer Science, where sound theoretical concepts (e.g. Automata, Grammars, Graph Theory, Lattice Theory etc.) are backed by practical implementations (Lexical analyzers, Parsers, Code Optimizers etc.) to solve the real world problem of fast and resource-efficient compilation. Most existing compiler courses [9, 6, 8, 1, 2] divide the curriculum into modules corresponding to the phases of compilation. Instructors discuss the theory in lectures while students typically work on a semester long project implementing compiler for some small language.

In a typical course about 15%-22% of the total time is spent on syntax analysis phase (also called parsing techniques, see Table 1). A number of concepts are introduced to explain the internals of parsers, for example first sets, follow sets, item set, goto and closure sets, parse tables and the parsing algorithms [9], making the understanding difficult. While parser generators (YACC and its variants) allow the students to experiment with grammars, the working of the parser generated by the tools is still opaque¹.

In this paper we present ParseIT, a tool for teaching parsing techniques. ParseIT helps students to understand the parsing concepts 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 parse table provided by the user, this algorithm enables creation of an input string that distinguishes a successful parse from an

¹The generated parsers do produce debugging information when used with appropriate options, but this information is of little didactical value as one needs to know the parsing algorithms to understand it.
unsuccessful one.

We describe some of the systems developed by other for teaching compiler concepts in Sec. 2. The algorithms for problem and hint generation are described in Sec. 3. We conclude in Sec. 4 and present directions for future work.

2. RELATED WORK

Several efforts exist to automate teaching phases of compilers and to help developing a compiler as a course project. LISA [14] helps students learn compiler technology through animations and visualizations. The tool uses animations to explain the working of 3 phases of compilers, namely, lexical analysis, syntax analysis and semantic analysis. Lexical analysis is taught using animations in DFAs. For syntax analysis, animations are shown for 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 understand the working of phases by modifying the specification and observing the corresponding changes in the animation.

Lorenzo et. al. present a system for test-case based automated evaluation of compiler projects. Test cases (inputs and corresponding desired outputs) designed by the instructor are given as input to students' compilers. The tool then assess the compiler in three distinct steps–compilation, execution and correction. The system automatically generates different reports (for instructors and students) by analyzing the logs generated at each of these steps.

Demaille et. al. [11, 12] introduce several tools to improve teaching of compiler construction projects and make it relevant to the core curriculum. They made changes to Bison [5] to provide detailed textual and graphical descriptions of the automata, to simplify handling of symbols, location of the symbols. Other tools are included for AST and Code generation. Waite [18] proposed 3 strategies for teaching compilers–software project, application of theory and support for communicating with computer. Various other tools are also available to teach different phases of compiler like understanding code generation [16] , and understanding symbol tables through animations [17].

Our work differs from all these in that we use question-answering as a means to explain working of parsing technology. We guide the students towards correct solution through automated directed hints.

3. THE ParseIT TOOL

ParseIT takes as input a context free grammar and uses it as a basis for generating questions. These questions are of the form of Multiple Choice Questions (MCQ), and deal with various concepts related to parsing. The normal workflow involves the following steps:

1. User provides an input grammar and the choice of topic.
   The topics refer to the concepts related to parsing that includes FIRST set, FOLLOW set, LL Parsing Table, LL Parsing Moves, LR(0) Item-sets, LR(0) Parsing Table, L(0) Parsing Moves, etc.
2. A primary multiple choice question is generated based on the above two pieces of information.
3. If the user answers the problem incorrectly, then hints are generated for the same question in the form of questions
4. When a correct solution to the problem is received, another question for the same topic is generated and presented to the user.

In the preprocessing step, the system takes a grammar as input and generates the information required for correct solutions. In particular, the tool determines the FIRST set and FOLLOW set for all non-terminals, creates LL Parsing Table, canonical set of items for LR parsers, LR parsing table and the data structures generated in this step are then used in other steps.

For primary problem for the selected topic, ParseIT uses the data-structures to form MCQs having multiple correct answers. Users have to select all valid options, and no invalid option, for the answer to be deemed correct. The options are also generated using the preprocessed data.

In the answer evaluation step, the solution given by the user is compared with the solution computed by the tool in the preprocessing step. If the solutions match, then the control transfers back to the primary problem generation step to generate the next question. However, if the solution is wrong, the tool collects: a) the incorrect options which are selected in the solution, and b) the correct options which are not selected by the user and passes them to the hint generation step.²

For hints, the tool generates multiple hint questions for each of the incorrect choices. These questions are MCQs having a single correct choice. These questions force user to revise the concept required to get correct solution to the primary question, and guides her towards the correct solution.

3.1 Problem Generation

Parsing techniques require solving three main types of problems: a) computation of sets of elements, for example FIRST, FOLLOW, LR Items, GOTO, CLOSURE, b) computation of entries in a parse table, and c) steps of a parser on a given input string.

Since all the sets and tables are computed by ParseIT in the preprocessing step, generation of questions is easy. Further, a set of candidate choices is obtained by adding to the set of correct choices a few mutations (addition/removal of a term) or using a term from the solution for another similar problem. Evaluation of user solution is a simple comparison with the computed solution. We omit the details of problem generation and answer evaluation for brevity.

3.2 Hint Generation

The tool generates hint questions to guide the user into reaching the correct solution of the primary problem and understanding her mistakes. ParseIT generates 3 types of hints:

- Hint_Rule: These types of questions are generated for all the options given as input to the Hint Generation step. It helps the user to understand the standard rules used in the technique, by which a particular option must or must not be a part of the correct solution.³

²In the rest of the paper, unless specified otherwise, we use the term incorrect choice for both the types of mistakes, i.e., the missing valid choice and the selected invalid choice.

³Hint_Rules are generated even when the user solution is correct (with a small probability). Otherwise, whenever the hint is generated, the user will know that her original answer was wrong, and may change it without understanding the concept.
3.3 Input String Generation

A parsing table is a two-dimensional array used to parse an input string. A grammar is considered LL/LR if the corresponding LL/LR parsing table has no duplicate entries. We generate hint of type Hint_String only for grammars accepted by LL(k) or LR(k) parsers, where k ∈ {0, 1}. For such grammars, if any cell of the parsing table is filled incorrectly by the user, then there exists a valid string for that cell during parsing of the string. Due to incorrectly filled entry, the parsing with incorrect table for the string fails.

Input String Generation for LL Parsing.

A valid input string is automatically generated by the tool for an incorrectly filled cell of LL parsing table. A valid input string is the one which can be parsed by the correct parse table but not by the user’s table. To generate such a string for a cell of LL parsing table, a graph is generated using the grammar. All the symbols in the grammar appear as the vertices of the graph. Edges are created from the symbol on the left side to the symbols on the right side of a rule. An edge is labeled with the shortest rule used for creating it. For each non-terminal, we also compute and record a terminal-only string derivable from that non-terminal.

To generate the desired string, in the first step, the tool first finds a path (sequence of edge labels) from the start symbol of the grammar to the non-terminal N representing the row containing the incorrectly filled cell in the parsing table. Every non-terminal that appears in the path is expanded with the rule on the label of the edge between the corresponding vertex and its successor. This expansion may introduce new non-terminals which are replace by their terminal-only strings.

In the next step, the tool finds a path from the non-terminal N to the terminal T representing the column containing the incorrectly filled cell in the parsing table. This path is also expanded in the same way as described above.

The two string from step 1 and 2 are concatenated to get the final input string. The generation of input string for LL parsing is described by the algorithms 1-4.

The following example depicts the work-flow of input string generation algorithm for LL parsing table:

Example 3.1. Consider the grammar:

E → T E' | \epsilon
T → F T' | \epsilon
F → (E) | \epsilon
E → T E' | \epsilon
T → T' | \epsilon
T' → * F T' | \epsilon
F → ( ( E ) ) | id

The correct LL parsing table M for this grammar is:

<table>
<thead>
<tr>
<th>( )</th>
<th>id</th>
<th>*</th>
<th>+</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T→FT'</td>
<td>T→FT'</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>F→(E)</td>
<td>F→id</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>E→TE'</td>
<td>E→TE'</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T'</td>
<td>T'→ε</td>
<td>T'→*FT'</td>
<td>T'→ε</td>
<td>T'→ε</td>
</tr>
<tr>
<td>E'</td>
<td>E'→ε</td>
<td>E'→ε</td>
<td>E→TE'</td>
<td>E→ε</td>
</tr>
</tbody>
</table>

Suppose a user fills the cell M[T'[i]] incorrectly. Fig. 1 shows the graph generated by ParseIT for this grammar.
Algorithm 2: Graph generation for LL Parsing

1: function gen_ll_graph(Grammar)
2:   Rules := table containing rules for every non-terminal in Grammar
3:   N := set of non-terminals in Grammar
4:   T := set of terminals in Grammar
5:   Vertices = N ∪ T ⊃ set of vertices in graph
6:   Edges = {} ⊃ set of edges in graph
7:   Labels := table with keys belonging to Edges
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:     Edges = Edges ∪ (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 for
19: end for
20: return (Edges, Vertices, Labels)
21: end function

Fig. 2: Path in the tree from E to T'

Fig. 2 shows the shortest path from the start symbol E to T' (the non-terminal representing the erroneous cell) computed using Dijkstra’s shortest path algorithm.

In order to build the desired input string, a stack is used. Initially it consists of end marker ‘$’ and start symbol E. 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 terminal symbol appears on top of the stack, it is popped from the stack and appended to the input string. If a non-terminal appears on the top of the stack then,

- if it corresponds to a node in the path, the label of the outgoing edge from this node in the path is pushed on to the stack in reverse order.
- otherwise the terminal string for this non-terminal is pushed on to the stack in reverse order.

The steps involved in computing the path string from E to T’ are shown below:

Algorithm 3: Generate partial input string for LL parsing

1: function find_ll_input(Partial, Path, Labels, TermStrings, Grammar, Stack)
2:   N := set of all non-terminals in Grammar
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 ∈ N then
12:       minr = REVERSE(TermStrings[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 4: Generation of final input string

1: function empty_ll_stack(Grammar, Stack, Partial, TermStrings)
2:   N := set of all non-terminals in Grammar
3:   while top(Stack) ≠ $ do
4:     top = POP(Stack)
5:     if top ∈ N then
6:       minr = REVERSE(TermStrings[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

<table>
<thead>
<tr>
<th>stack</th>
<th>input string</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E$</td>
<td>$E'T$</td>
</tr>
<tr>
<td>$E'T'$</td>
<td>$F$</td>
</tr>
</tbody>
</table>

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 ($F → id$). The configuration now becomes

<table>
<thead>
<tr>
<th>stack</th>
<th>input string</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E'T'$</td>
<td>$id$</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>stack</th>
<th>input string</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E'T'$</td>
<td>id</td>
</tr>
</tbody>
</table>

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 shortest path algorithm is applied, to find shortest path from $T'$ to $\)$ as shown in fig. 3.

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 exercising the entry $M[T'[\)]$. We choose shortest rule derivable from $T'$, just like we did for other non-terminals.

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.

### Input String Generation for LR Parsing.

The algorithm for input string generation for LR parsing differs from LL parsing in that in case of LR parsing, we use finite automata (DFA) of viable prefixes to find the string. We further need heuristics to pick appropriate reduce rules from the parse table to make sure that the parser stack is in accept state at the end of input. We explain the algorithm with an example due to lack of space.

The generation of input string for LR parsing is described by the algorithms 5-13.
Algorithm 6 Graph generation from grammar for LR Parsing

1: function GENERATE_LR_GRAPH(Items, Grammar)
2:   Symbols := set of symbols in Grammar
3:   Vertices = Items ⊖ set of vertices in graph
4:   Edges = {} ⊖ set of edges in graph
5:   Labels := table with keys belonging to Edges
6:   for all itemset in Items do
7:     for all S in Symbols do
8:       dest = goto(itemset, S)
9:       if dest ≠ ∅ then
10:          Edges = Edges ∪ (itemset, dest)
11:          Labels[(itemset, dest)] = S
12:     end if
13:   end for
14: end for
15: return (Vertices, Edges, Labels)
16: end function

Algorithm 7 Convert reduce problem to shift problem in LR parsing

1: function FIND_CONFIGURATION(Items, Table, Grammar, Symbol, Entry, MinRules)
2:   stk =
3:   rule = reverse(Rules[entry[1]])
4:   push(stk, rule)
5:   Str = EXPAND_RULE(stk, MinRules, Str, Grammar)
6:   State_Mapping = mapping between bottom and corresponding up symbol
7:   Up = ""
8:   Bottom = ""
9:   Sum = ""
10:   Bottom_Set = {}
11: while Entry[0] ≠ "shift" do
12:   Sym = KEY(Rules[entry[1]])
13:   for all row in Table do
14:     key = KEY(row)
15:     value = Table[key][Sym]
16:     if value[0] ≠ "error" then
17:       Bottom_Set = Bottom_Set ∪ key
18:     end if
19:   end for
20:   (State_Mapping, Bottom_Set) = FIND_STATE_MAPPING(Symbol, Table, Bottom_Set, Sym)
21:   value_set = VALUES(State_Mapping)
22:   Up = pick a value from value_set
23:   Entry = Table[Up][Symbol]
24: end while
25: Bottom = KEY(State_Mapping[Up])
26: return (Bottom, Sym, Up, Str)
27: end function

Algorithm 8 Expand non-terminal in the path with its rule

1: function EXPAND_RULE(stk, MinRules, Partial, Grammar)
2:   NonTerminals := set of non-terminals in Grammar
3:   while stk ≠ ∅ do
4:     top = POP(stk)
5:     if top ∈ NonTerminals then
6:       minr = REVERSE(MinRules[top])
7:       PUSH(stk, minr)
8:     else
9:       Partial = APPEND(Partial, top)
10:   end if
11: end while
12: return Partial
13: end function

Figure 4: DFA for LR parsing

Example 3.2 depicts the work-flow of input string generation algorithm for LR parsing table:

Example 3.2. Suppose the augmented grammar is

0) S' → S
1) S → CC
2) C → aC
3) C → d

The collection of sets of items for this grammar is:

I0: S' → · S
   S → · CC
   C → · aC
   C → · d

I1: S' → S·
   S → · CC·
   C → · aC·
   C → · d·

The DFA for this is shown in Fig. 4. State 0 is the initial state and 1, 4, 5, 6 is the set of final states. The correct LR parsing table L for this grammar is shown below:
Algorithm 9 Mapping between states below and states above the found symbol

1: function FIND_STATE_MAPPING(Sym, Table, Bottom_Sett, Symbol)
2:   State_Mapping = {}
3:   for all key in Bottom_Sett do
4:     value = Table[key][Sym]
5:     State_Mapping[key] = value
6:   end for
7:   valueset = values(State_Mapping)
8:   shift_set = {}
9:   reduce_set = {}
10:   for all val in valueset do
11:     en = Table[val][Symbol]
12:     if en[0] == "error" then
13:       valueset = valueset - val
14:     else if en[0] == "shift" then
15:       shift_set = shift_set
16:     end if
17:   end for
18:   if shift_set !empty then
19:     reduce_set = valueset - shift_set
20:     valueset = valueset - reduce_set
21:   else
22:     setitem = ""
23:     for all rval in reduce_set do
24:       en = Table[rval][Symbol]
25:       if en[0] == "accept" then
26:         reduce_set = rval - valueset
27:         setitem = rval
28:       else
29:         rule = Rules[en[1]]
30:         if VALUE(rule) != Sym then
31:           rval = rval - valueset
32:         end if
33:       end if
34:     end for
35:   end if
36:   end for
37:   for all key in State_Mapping do
38:     if reduce_set does not contain key then
39:       REMOVE(State_Mapping[key])
40:     end if
41:   end for
42:   Bottom_set = KEYS(State_Mapping)
43: end function

Algorithm 10 Generate partial input string for LR parsing

1: function FIND_LR_INPUT(Partial, Path, Labels, Grammar, Stack, MinRules)
2:   while Path !empty do
3:     next = NEXT(Path)
4:     PUSH(Stack, next)
5:     dest = SUCCESSOR(Path, next)
6:     label = Labels[(next, dest)]
7:     if label !empty then
8:       PUSH(Stack, label)
9:       stk = label
10:      Partial = EXPAND_RULE(stk, MinRules, Partial, Grammar)
11:   end if
12: end while
13: return (Stack, Partial)
14: end function

Algorithm 11 Find path to one of the final states

1: function FINAL_PATH(Final_States, SH)
2:   path_set = {}
3:   for all fs in Final_States do
4:     path_set = path_set ∪ SH[fs]
5:   end for
6:   Path = MIN(path_set)
7: return Path
8: end function

Suppose user makes an incorrect entry in the cell L[3][d], then a path is found from 0 (initial state) to 3 (row containing incorrectly filled cell) using Dijkstra’s shortest path algorithm as shown in fig. 5.

All the symbols that appear on the path are pushed on a stack. The terminals which appear on the path are kept in a partial input string. Then, then edge labeled with d is considered. The symbol d and destination state 4 of edge are then pushed on the stack. As state 4 is one of the final states, so further path is required to be found. The configuration achieved by this is shown below:

```
<table>
<thead>
<tr>
<th>STATE</th>
<th>ACTION</th>
<th>GOTO</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>s3 d</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>s3 d</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>s3 d</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>r3 r3</td>
<td>r3</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>r2 r2</td>
<td>r2</td>
</tr>
</tbody>
</table>
```

Suppose user makes an incorrect entry in the cell L[3][d], then a path is found from 0 (initial state) to 3 (row containing incorrectly filled cell) using Dijkstra’s shortest path algorithm as shown in fig. 5.

The top symbol on the path are pushed on a stack. The terminals which appear on the path are kept in a partial input string. Then, then edge labeled with d is considered. The symbol d and destination state 4 of edge are then pushed on the stack. As state 4 is one of the final states, so further path is required to be found. The configuration achieved by this is shown below:

```
stack partial temporary
0a d4 ad X temporary
```

Now, as there is no symbol in temporary string and state 4 is on the top of stack, the symbols representing the cells containing entries in the row of state 4 in LR parsing table are kept in set X. As all the entries are same, so any symbol corresponding to these cells can be chosen as temporary string from X. Suppose ‘a’ is chosen as temporary string, then action is taken by considering entry L[4][a] in LR parsing table. This results in the following configuration:
Algorithm 12 Empty stack and generate final input string for LR parsing

1: function EMPTY_LR_STACK(Grammar, Stack, Partial, MinRules, X, Table, Itemsets)
2:    Rules := table containing Grammar rules with numbers as keys
3:    while top(Stack) ≠ "accept" do
4:        top = POP(Stack)
5:        X = FIND_SUBSTRINGSET(Grammar, Table, Itemsets, top, X)
6:        substring := Choose any 1 value from X
7:        entry = Table[top][substring]
8:        if entry[0] = "shift" then
9:            push(Stack, substring)
10:           push(Stack, entry[1])
11:        end if
12:        else if substring ≠ "$" then
13:            Partial = APPEND(Partial, substring)
14:            substring = ""#end if
15:        else
16:            while Stack ≠ ∅ do
17:                POP(Stack)
18:            end while
19:            entry = Table[top][entry[0]]
20:            push(Stack, entry[0])
21:        end if
22:    end while
23:    return Partial
24: end function

Algorithm 13 Find set symbols which can be appended into input string

1: function FIND_SUBSTRINGSET(Grammar, Table, Itemsets, top, X)
2:    Terminals := set of terminals in Grammar
3:    if X = ∅ then
4:        for all term in terminals do
5:            if Table[top][term] ≠ empty then
6:                X = X ∪ term
7:        end if
8:    end for
9:    end if
10:   shift_symbols = ∅
11:   for all terminal in X do
12:       entry = Table[top][terminal]
13:       if entry[0] = "shift" then
14:           shift_symbols = shift_symbols ∪ a L l terminal
15:       end if
16:    end for
17:    X = X - shift_symbols
18:    if X ≠ ∅ then
19:        temp = ""
20:        previous = ∅
21:        sym := terminal corresponding to the cell in Table having shift command to the state containing itemset with dot at rightmost position
22:        for all symbol in shift_symbols do
23:            en = Table[top][symbol]
24:            current = Itemsets[en[1]]
25:            previous = RIGHTMOST(previous, current)
26:            sym = symbol
27:        end for
28:        X = X ∪ l sym
29:    end if
30:    return X
31: end function
Figure 5: Path from 0 to 3

<table>
<thead>
<tr>
<th>stack</th>
<th>partial</th>
<th>X</th>
<th>temporary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0a3d4</td>
<td>ad</td>
<td>{a, d, $}</td>
<td>a</td>
</tr>
<tr>
<td>0a3C6</td>
<td>ad</td>
<td>{a, d, $}</td>
<td>a</td>
</tr>
<tr>
<td>0C2</td>
<td>ad</td>
<td>{a, d}</td>
<td>d</td>
</tr>
<tr>
<td>0C2d4</td>
<td>add</td>
<td>{a, d, $}</td>
<td>a</td>
</tr>
<tr>
<td>0C2C5</td>
<td>add</td>
<td>{$}</td>
<td>$</td>
</tr>
<tr>
<td>0S1</td>
<td>add</td>
<td>{$}</td>
<td>$</td>
</tr>
<tr>
<td>accept</td>
<td>add</td>
<td>{$}</td>
<td>$</td>
</tr>
</tbody>
</table>

Now, 6 is on top of stack and has entries on cells of same symbols as X. So, X remains the same and ‘a’ is chosen, leading to the following configuration:

<table>
<thead>
<tr>
<th>stack</th>
<th>partial</th>
<th>X</th>
<th>temporary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0a3d4</td>
<td>ad</td>
<td>{a, d, $}</td>
<td>a</td>
</tr>
<tr>
<td>0a3C6</td>
<td>ad</td>
<td>{a, d, $}</td>
<td>a</td>
</tr>
<tr>
<td>0C2</td>
<td>ad</td>
<td>{a, d}</td>
<td>d</td>
</tr>
</tbody>
</table>

As stack has reached to “accept”, so the input string in partial is the final input string for incorrectly filled cell L[3][d] of LR parsing table.

4. CONCLUSIONS AND FUTURE WORK

In this paper, we described ParseIT for teaching parsing techniques. Our approach is question-answering based: problems are generated automatically and given to students to explain working of a parser. Further, the hints provided by the tool are also in forms of targeted questions that help a student discover her mistake and revise the concept at the same time. ParseIT allows students to learn the techniques at their own pace, according to their convenience. 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 and teaching assistants.

Similar tools exist to teach few other phases of compiler. In future we plan to integrate these tools with ParseIT, and develop new tools to automate tutoring of all the phases of compiler. We also plan to build animations around these concepts to improve student experience and understanding.

5. REFERENCES