PROBABILISTIC LR PARSING FOR GENERAL CONTEXT-FREE GRAMMARS*

See-Kiong Ng and Masaru Tomita
School of Computer Science and Center for Machine Translation
Carnegie Mellon University
Pittsburgh, PA 15213 U.S.A.

ABSTRACT

To combine the advantages of probabilistic grammars and generalized LR parsing, an algorithm for constructing a probabilistic LR parser given a probabilistic context-free grammar is needed. In this paper, implementation issues in adapting Tomita's generalized LR parser with graph-structured stack to perform probabilistic parsing are discussed. Wright and Wrigley (1989) has proposed a probabilistic LR-table construction algorithm for non-left-recursive context-free grammars. To account for left recursions, a method for computing item probabilities using the generation of systems of linear equations is presented. The notion of deferred probabilities is proposed as a means for dealing with similar item sets with differing probability assignments.

1 Introduction

Probabilistic grammars provide a formalism which accounts for certain statistical aspects of the language, allows stochastic disambiguation of sentences, and helps in the efficiency of the syntactic analysis. Generalized LR parsing is a highly efficient parsing algorithm that has been adapted to handle arbitrary context-free grammars. To combine the advantages of both mechanisms, an algorithm for constructing a generalized probabilistic LR parser given a probabilistic context-free grammar is needed. In Wright and Wrigley (1989), a probabilistic LR-table construction method has been proposed for non-left-recursive context-free grammars. However, in practice, left-recursive context-free grammars are not uncommon, and it is often necessary to retain this left-recursive grammar structure. Thus, a method for handling left-recursions is needed in order to attain probabilistic LR-table construction for *general* context free grammars.

In this paper, we concentrate on incorporating probabilistic grammars with generalized LR parsing for efficiency. Stochastic information from probabilistic grammar can be used in making statistical decision during runtime to improve performance. In Section 3, we show how to adapt Tomita's(1985, 1987) generalized LR parser with graph-structured stack to perform probabilistic parsing and discuss related implementation issues. In Section 4, we describe the difficulty in computing item probabilities for left recursive context-free grammars. A solution is proposed in Section 5, which involves encoding item dependencies in terms of a system of linear equations. These equations can then be solved by Gaussian Elimination (Strang 1980) to give the item probabilities, from which the stochastic factors of the corresponding parse actions can be computed as described in Wright and Wrigley (1989).

We also introduce the notion of *deferred probability* in Section 6 in order to prevent creating excessive number of duplicate items which are similar except for their probability assignments.

*This research was supported in part by National Science Foundation under contract IRI-8858085.

2 Background

Probabilistic LR parsing is based on the notions of probabilistic context-free grammar and probabilistic LR parsing table, which are both augmented versions of their nonprobabilistic counterparts. In this section, we provide the definitions for the probabilistic versions.

2.1 Probabilistic CFG

A *probabilistic context-free grammar* (PCFG) (Suppes 1970, Wetherall 1980, Wright and Wrigley 1989) G, is a 4-tuple $\langle N, T, R, S\rangle$ where N is a set of non-terminal symbols including S the start symbol, T a set of terminal symbols, and R a set of probabilistic productions of the form $< A \rightarrow \alpha,\ p >$ where $A \in N$, $\alpha \in (N \cup T)^*$, and p the production probability. The probability p is the conditional probability $P(\alpha|A)$, which is the probability that the non-terminal A which appears during a derivation process is rewritten by the sequence α. Clearly if there are k A-productions with probabilities $p_1, \ldots, p_k$, then $\sum_{i=1}^{k} p_i = 1$, since the symbol A must be rewritten by the right hand side of some A-production. The production probabilities can be estimated from the corpus as outlined in Fu and Booth(1975) or Fujisaki(1984).

It is assumed that the steps of every derivation in the PCFG are mutually independent, meaning that the probability of applying a rewrite rule de-

Figure 1: GRA1: A Non-left Recursive PCFG

(1)	$S \rightarrow NP\ VP$	1
(2)	$NP \rightarrow n$	$\frac{1}{3}$
(3)	$NP \rightarrow det\ n$	$\frac{2}{3}$
(4)	$VP \rightarrow v\ NP$	1

Figure 2: GRA2: A Left-recursive PCFG

(1)	$S \rightarrow NP\ VP$	$\frac{3}{4}$
(2)	$S \rightarrow S\ PP$	$\frac{1}{4}$
(3)	$NP \rightarrow n$	$\frac{1}{2}$
(4)	$NP \rightarrow det\ n$	$\frac{2}{5}$
(5)	$NP \rightarrow NP\ PP$	$\frac{1}{10}$
(6)	$PP \rightarrow prep\ NP$	1
(7)	$VP \rightarrow v\ NP$	1

pends only upon the presence of a given nonterminal symbol (the premis) in a derivation and not upon how the premis was generated. Thus, the probability of a derivation is simply the product of the production probabilities of the productions in the derivation sequence.

Figures 1, 2 and 3 show three example PCFGs GRA1, GRA2 and GRA3 respectively. Incidentally, GRA1 is non-left recursive, GRA2 and GRA3 are both left-recursive, although GRA3 is "more" left-recursive than GRA2. GRA2 is said to have *simple recursion* since there is only a finite number of distinct left-recursive loops[1] in the grammar. GRA3, on the other hand, is said to have *massive left recursions* because of the intermingled left recursions, which result in infinite (possibly uncountable) number of distinct left-recursive loops in the grammar.

[1]A loop is a derivation cycle in which the first and last productions used in the derivation sequence are the same and occur nowhere else in the sequence.

Figure 3: GRA3:A Massively Left-recursive PCFG

(1)	$S \rightarrow S\ a_1$	$\frac{1}{7}$
(2)	$S \rightarrow B\ a_2$	$\frac{4}{7}$
(3)	$S \rightarrow C\ a_3$	$\frac{2}{7}$
(4)	$B \rightarrow S\ a_3$	$\frac{1}{3}$
(5)	$B \rightarrow B\ a_2$	$\frac{1}{6}$
(6)	$B \rightarrow C\ a_1$	$\frac{1}{2}$
(7)	$C \rightarrow S\ a_2$	$\frac{1}{5}$
(8)	$C \rightarrow B\ a_3$	$\frac{4}{15}$
(9)	$C \rightarrow C\ a_1$	$\frac{1}{15}$
(10)	$C \rightarrow a_3\ B$	$\frac{1}{3}$
(11)	$C \rightarrow a_3$	$\frac{2}{15}$

2.2 Probabilistic LR Parse Table

A probabilistic LR table is an augmented LR table of which the entries in the ACTION-table contains an additional field which is the probability of the action. We call this probability *stochastic factor* because it is the factor used in the computation (multiplication) of the *runtime stochastic product*. The parser keeps this stochastic product during runtime for each possible derivation, reflecting their respective likelihoods. This product can be computed during runtime by multiplication using the precomputed stochastic factors of the parsing actions (or by addition if the stochastic factors are expressed in logarithms). The parser can use this stochastic information to disambiguate or direct/prune its search probabilistically.

Figures 4, 5 and 6 show the respective probabilistic parsing tables for GRA1, GRA2 and GRA3, as constructed by the algorithm outlined in Section 5. Note that the stochastic factors of distinct actions associated with a state add up to 1 as expected, since each action's stochastic factor is simply the probability of the parser making that action during that point of parse. The format of the GOTO-table is unchanged as no stochastic factor is associated with GOTO actions.

3 Generalized Probabilistic LR Parsers for Arbitrary PCFGs

In this section, we describe how the efficient generalized LR parser with graph-structured stack in (Tomita 1985, 1987) can be adapted to parse probabilistically using the augmented parsing table. In particular, we discuss how to maintain consistent runtime stochastic products base on three key notions of the graph-structured stack: merging, local ambiguity packing and splitting. We assume that the state number and the respective runtime stochastic product are stored at each stack node.

3.1 Merging

Merging occurs when an element is being shifted onto two or more of the stack tops. Figure 7 illustrates a typical scenario in which a new state (State 3) is pushed onto stack tops States 1 and 2, of which original stochastic products are p_1 and p_2 respectively. These two nodes's stochastic products are modified to p_1q_1 and p_2q_2 correspondingly. If the stochastic factors of the actions has been represented as logarithms in the parse table, then their new "product" (or rather, logarithmic sums) would be $p_1 + q_1$ and $p_2 + q_2$ instead. For the stochastic product of Node 3, we can either use the sum of its parents' products (giving p_3 as $p_1q_1 + p_2q_2$) if we adopt *strict probabilistic approach*, or the maximum of the products (ie, $p_3 = \max(p_1q_1, p_2q_2)$) if we adopt the

Figure 4: Probabilistic Parsing Table for GRA1

State	ACTION				GOTO		
	det	*n*	*v*	\$	*NP*	*VP*	*S*
0	$\langle sh2, \frac{2}{3}\rangle$	$\langle sh1, \frac{1}{3}\rangle$			4		3
1			$\langle re2, 1\rangle$	$\langle re2, 1\rangle$			
2		$\langle sh5, 1\rangle$					
3				$\langle acc, 1\rangle$			
4			$\langle sh6, 1\rangle$			7	
5			$\langle re3, 1\rangle$	$\langle re3, 1\rangle$			
6	$\langle sh2, \frac{2}{3}\rangle$	$\langle sh1, \frac{1}{3}\rangle$			8		
7				$\langle re1, 1\rangle$			
8				$\langle re4, 1\rangle$			

Figure 5: Probabilistic Parsing Table for GRA2

State	ACTION					GOTO			
	det	*n*	*v*	*prep*	\$	*NP*	*PP*	*VP*	*S*
0	$\langle sh2, \frac{4}{9}\rangle$	$\langle sh1, \frac{5}{9}\rangle$				3			4
1			$\langle re3, 1\rangle$	$\langle re3, 1\rangle$	$\langle re3, 1\rangle$				
2		$\langle sh5, 1\rangle$							
3			$\langle sh7, \frac{9}{10}\rangle$	$\langle sh6, \frac{1}{10}\rangle$			8	9	
4				$\langle sh6, \frac{1}{4}\rangle$	$\langle acc, \frac{3}{4}\rangle$		10		
5			$\langle re4, 1\rangle$	$\langle re4, 1\rangle$	$\langle re4, 1\rangle$				
6	$\langle sh2, \frac{4}{9}\rangle$	$\langle sh1, \frac{5}{9}\rangle$				11			
7	$\langle sh2, \frac{4}{9}\rangle$	$\langle sh1, \frac{5}{9}\rangle$				12			
8			$\langle re5, 1\rangle$	$\langle re5, 1\rangle$	$\langle re5, 1\rangle$				
9				$\langle re1, 1\rangle$	$\langle re1, 1\rangle$				
10				$\langle re2, 1\rangle$	$\langle re2, 1\rangle$				
11			$\langle re6, \frac{9}{10}\rangle$	$\langle re6, \frac{9}{10}\rangle$ $\langle sh6, \frac{1}{10}\rangle$	$\langle re6, \frac{9}{10}\rangle$		8		
12			$\langle re7, \frac{9}{10}\rangle$	$\langle re7, \frac{9}{10}\rangle$ $\langle sh6, \frac{1}{10}\rangle$	$\langle re7, \frac{9}{10}\rangle$		8		

maximum likelihood approach. Note that although the maximum likelihood approach is in some sense less "accurate" than the strict probabilistic approach, it is a reasonable approximate and has an added advantage when the stochastic factors are represented in logarithms, in which case the stochastic "products" of the parse stack can be maintained using only addition and subtraction operators(assuming, of course, that additions and subtractions are "cheaper" computationally than multiplications and divisions).

3.2 Local Ambiguity Packing

Local ambiguity packing occurs when two or more branches of the stack are reduced to the same non-terminal symbol. To be precise, this occurs when the parser attempts to create a GOTO state node (after a reduce action, that is) and realize that the parent already has a child node of the same state. In this case there is no need to create the GOTO node but to use that child node ("packing"). This is equivalent to the merging of shift nodes, and can be handled similarly: the runtime product of the child node is modified to the new "merged" product (either by summation or maximalization). This modification should be propagated accordingly to the successors of the packed child node, if any.

Figure 7: Merging

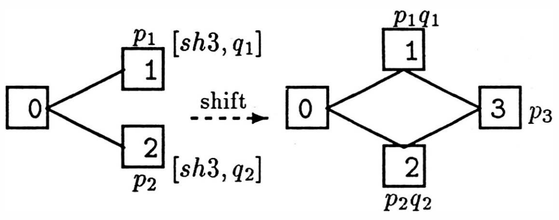


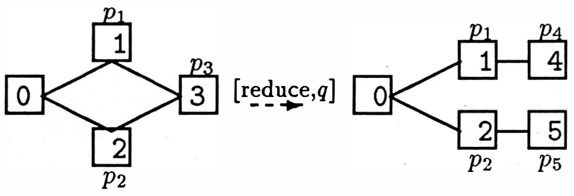
Figure 6: Probabilistic Parsing Table for GRA3

State	ACTION				GOTO		
	a_1	a_2	a_3	\$	S	B	C
0			$\langle sh1, 1\rangle$		2	3	4
1	$\langle re11, \frac{2}{7}\rangle$		$\langle re11, \frac{2}{7}\rangle$ $\langle sh1, \frac{5}{7}\rangle$		5	6	7
2	$\langle sh9, \frac{1}{7}\rangle$	$\langle sh8, \frac{33}{203}\rangle$	$\langle sh10, \frac{64}{203}\rangle$	$\langle acc, \frac{11}{29}\rangle$			
3		$\langle sh11, \frac{37}{48}\rangle$	$\langle sh12, \frac{11}{48}\rangle$				
4	$\langle sh13, \frac{107}{165}\rangle$		$\langle sh14, \frac{58}{165}\rangle$				
5	$\langle sh9, \frac{1}{7}\rangle$	$\langle sh8, \frac{66}{259}\rangle$	$\langle sh10, \frac{156}{259}\rangle$				
6	$\langle re10, \frac{77}{234}\rangle$	$\langle sh15, \frac{113}{234}\rangle$	$\langle re10, \frac{77}{234}\rangle$ $\langle sh12, \frac{22}{117}\rangle$				
7	$\langle sh16, \frac{128}{165}\rangle$		$\langle sh14, \frac{37}{165}\rangle$				
8	$\langle re7, 1\rangle$		$\langle re7, 1\rangle$				
9	$\langle re1, 1\rangle$	$\langle re1, 1\rangle$	$\langle re1, 1\rangle$	$\langle re1, 1\rangle$			
10	$\langle re4, 1\rangle$	$\langle re4, 1\rangle$	$\langle re4, 1\rangle$				
11	$\langle re2, \frac{29}{37}\rangle$ $\langle re5, \frac{8}{37}\rangle$	$\langle re2, \frac{29}{37}\rangle$ $\langle re5, \frac{8}{37}\rangle$	$\langle re2, \frac{29}{37}\rangle$ $\langle re5, \frac{8}{37}\rangle$	$\langle re2, \frac{29}{37}\rangle$			
12	$\langle re8, 1\rangle$		$\langle re8, 1\rangle$				
13	$\langle re6, \frac{96}{107}\rangle$ $\langle re9, \frac{11}{107}\rangle$	$\langle re6, \frac{96}{107}\rangle$	$\langle re6, \frac{96}{107}\rangle$ $\langle re9, \frac{11}{107}\rangle$				
14	$\langle re3, 1\rangle$	$\langle re3, 1\rangle$	$\langle re3, 1\rangle$	$\langle re3, 1\rangle$			
15	$\langle re2, \frac{74}{113}\rangle$ $\langle re5, \frac{39}{113}\rangle$	$\langle re2, \frac{74}{113}\rangle$ $\langle re5, \frac{39}{113}\rangle$	$\langle re2, \frac{74}{113}\rangle$ $\langle re5, \frac{39}{113}\rangle$	$\langle re2, \frac{74}{113}\rangle$			
16	$\langle re6, \frac{117}{128}\rangle$ $\langle re9, \frac{11}{128}\rangle$	$\langle re6, \frac{117}{128}\rangle$	$\langle re6, \frac{117}{128}\rangle$ $\langle re9, \frac{11}{128}\rangle$				

3.3 Splitting

Splitting occurs when there is an action conflict. This can be handled straightforwardly by creating corresponding new nodes for the new resulting states with the respective runtime products (such as the product of the parent's stochastic product with the action's stochastic factor). Splitting can also occur when reducing (popping) a merged node. In this case, the parser needs to recover the original runtime product of the merged components, which can be obtained with some mathematical manipulation from the runtime products recorded in the merged node's parents. Figure 8 illustrates a simple situation in which a merged node is split into two. In the figure, a reduce action (of which the corresponding production is of unit length) is applied at Node 3, and the GOTO's for Nodes 1 and 2 are states 4 and 5 respectively. In the case that strict probabilistic approach is used in merging (see above), we get $p_4 = \frac{p_1}{p_1+p_2}p_3q$ and $p_5 = \frac{p_2}{p_1+p_2}p_3q$. If the maximum likelihood approach is used, then $p_4 = \frac{p_1}{\max(p_1,p_2)}p_3q$ and $p_5 = \frac{p_2}{\max(p_1,p_2)}p_3q$. Furthermore, if the stochastic factors have been expressed in logarithms, then $p_4 = p_3 - \max(p_1,p_2) + p_1 + q$ and $p_5 = p_3 - \max(p_1,p_2) + p_2 + q$ (notice that only addition and subtraction are needed, as promised).

Figure 8: Splitting



In general, there may be more than one splitting corresponding to a reduce action (ie, we may have to pop more than one merged nodes). For every split node, we must recover the runtime products of its parents to obtain the appropriate stochastic products for the resulting new branches. This can be tricky and is one of the reasons why a tree-structured stack (described below) instead of graphs might perform better in some cases.

3.4 Using Stochastic Product to Guide Search

The main point of maintaining the runtime stochastic products is to use it as a good indicator function to guide search. In practical situation, the grammar can be highly ambiguous, resulting in many branches of ambiguity in the parse stack. As discussed before, the runtime stochastic product reflects the likelihood of that branch to complete successfully.

In Tomita's generalized LR parser, processes are synchronized by performing all the reduce actions before the shift actions. In this way, the processes are made to scan the input at the same rate, which in turn allows the unification of processes in the same state. Thus, the runtime stochastic products can be a good enough indicator of how promising each branch (ie. partial derivation) is, since we are comparing among partial derivations of same input length. We can perform beam search by pruning away branches which are less promising.

If instead of the breadth-first style beam search approach described above we employ a best-first (or depth-first) strategy, then not all of the branches will correspond to the same input length. Since the measure of runtime stochastic product is biased towards shorter sentences, a good heuristic would have to take into account of the number of input symbols consumed. Even so, handling best-first search can be tricky with Tomita's graph-structured stack without the process-input synchronization, especially with the merging and packing of nodes. Presumably, we can have additional data structure to serve as lookup table of the nodes currently in the graph stack: for instance, an n by m matrix (where n is the number of states in the parse table and m the input length) indexed by the state number and the input position storing pointers to current stack nodes. With this lookup table, the parser can check if there is any stack node it can use before creating a new one. However, in the worst case, the nodes that could have been merged or packed might have already been popped of the stack before it can be re-used. In this case, the parser degenerates into one with tree-structured stack (ie, only splitting, but no merging and packing) and the laborious book-keeping of the stochastic products due to the graph structure of the parse stack seems wasted. It might be more productive then to employ a tree-structured stack instead of a graph-structured stack, since the book-keeping of runtime stochastic products for trees is much simpler: as each tree branch represents exactly one possible parse, we can associate the respective runtime stochastic products to the leaf nodes (instead of every node) in the parse stack, and updating would involve only multiplying (or adding, in the logarithmic case) with the stochastic factors of the corresponding parse actions to obtain the new stochastic products. The major drawback of the tree-stack version is that it is merely a slightly compacted form of stack list (Tomita 1987) — which means that the tree can grow unmanageably large in a short period, unless suitable pruning is done. Hopefully, the runtime stochastic product will serve as good heuristic for pruning the branches; but whether it is the case that the simplicity of the tree implementation overrides that of the representational efficiency of the graph version remains to be studied.

4 Problem with Left Recursion

The approach to probabilistic LR table construction for non-left recursive PCFG , as proposed by Wright and Wrigley(1989), is to augment the standard SLR table construction algorithm presented in Aho and Ullman(1977) to generate a probabilistic version. The notion of a probabilistic item $\langle A \rightarrow \alpha \cdot \beta,\ p\rangle$ is introduced, with $\langle A \rightarrow \alpha \cdot \beta\rangle$ being an ordinary LR(0) item, and p the item probability, which is interpreted as the posterior probability of the item in the state. The major extension is the computation of these item probabilities from which the stochastic factors of the parse actions can be determined. Wright and Wrigley(1989) have shown a direct method for computing the item probabilities for non-left recursive grammars. The probabilistic parsing table in Figure 4 for the non-left recursive grammar GRA1 is thus constructed.

Since there is an algorithm for removing left recursions from a context-free grammar (Aho and Ullman 1977), it is conceivable that the algorithm can be modified to convert a left-recursive PCFG to one that is non left-recursive. Given a left-recursive PCFG, we can apply this algorithm, and then use Wright and Wrigley(1989)'s table construction method on the resulting non left-recursive grammar to create the parsing table. Unfortunately, the left-recursion elimination algorithm destructs the original grammar structure. In practice, especially in natural language processing, it is often necessary to preserve the original grammar structure. Hence a method for constructing a parse table without grammar conversion is needed.

For grammars with left recursion, the computation of item probabilities becomes nontrivial. First of all, item probability ceases to be a "probability", as an item which is involved in left recursion is effectively a coalescence of an infinite number of similar items along the cyclic paths, so its associated stochastic value is the sum of posteriori probabilities of these packed items. For instance, if starting from item $\langle A \rightarrow \alpha \cdot B\beta,\ p\rangle$ we derive the item $\langle C \rightarrow \cdot B\gamma,\ p \times p_B\rangle$, then by left recursion we must also have the items $\langle C \rightarrow \cdot B\gamma,\ p \times p_B^i\rangle$ for $i = 1, \ldots \infty$. The probabilistic item $\langle C \rightarrow \cdot B\gamma,\ q\rangle$, being a coalescence of these items, would have item probability $q = \sum_{i=1}^{\infty} p \times p_B^i = \frac{p}{1-p_B}$, and there is no guarantee that $q \leq 1$. This is understandable since $\langle C \rightarrow \cdot B\gamma, q \rangle$ is a coalescence of items which are not necessarily mutually exclusive. However, we need not be alarmed as the stochastic values of the underlying items are still legitimate probabilities.

Owing to this coalescence of infinite items into one single item in left recursive grammars, the computation of the stochastic values of items involves finding infinite sums of the items' stochastic values. For grammars with simple left recursion (that is, there are only finitely many left recursion loops) such as GRA2, we can still figure out the sum by enumeration, since there is only a finite number of the infinite sums corresponding to the left recursion loops. With massive left recursive grammars like GRA3 in which there is an infinite number of (intermingled) left recursion loops, the enumeration method fails. We shall illustrate this effect in the following sections.

4.1 Simple Left Recursion

For grammars with simple left recursion, it is possible to derive the stochastic values by simple cycle detection. For instance, consider the following set of LR(0) items for GRA2 in Figure 9.

Figure 9: An Example State for GRA2

I_0:	$[VP \rightarrow v \cdot NP,$	$S_0]$
I_1:	$[NP \rightarrow \cdot n,$	$S_1]$
I_2:	$[NP \rightarrow \cdot det\ n,$	$S_2]$
I_3:	$[NP \rightarrow \cdot NP\ PP,$	$S_3]$

Suppose the kernel set contains only I_0, with $S_0 = \frac{3}{7}$. Let $\mathcal{D}$ be a partial derivation before seeing the input symbol v. At this point, the possible derivations which will lead to item I_1 are:

$$\mathcal{D} \xRightarrow{S_0} VP \rightarrow v \cdot NP \xRightarrow{\frac{1}{2}} NP \rightarrow \cdot n$$
$$\mathcal{D} \xRightarrow{S_0} VP \rightarrow v \cdot NP \xRightarrow{\frac{1}{10}} NP \rightarrow \cdot NP\ VP \xRightarrow{\frac{1}{2}} NP \rightarrow \cdot n$$
$$\vdots$$
$$\mathcal{D} \xRightarrow{S_0} VP \rightarrow v \cdot NP \xRightarrow{\frac{1}{10}} NP \rightarrow \cdot NP\ VP \xRightarrow{\frac{1}{10}} \ldots \xRightarrow{\frac{1}{2}} NP \rightarrow \cdot n$$
$$\vdots$$

The sum of the posterior probabilities of the above possible partial derivations are:

$S_1 = (S_0 \times \frac{1}{2}) + (S_0 \times \frac{1}{10} \times \frac{1}{2}) + (S_0 \times \frac{1}{10}^2 \times \frac{1}{2}) + \ldots$
$= \frac{3}{7} \times \sum_{n=0}^{\infty} \frac{1}{10}^n \times \frac{1}{2} = \frac{5}{21}$

Similarly, $S_2 = \frac{3}{7} \times \sum_{n=0}^{\infty} \frac{1}{10}^n \times \frac{2}{5} = \frac{4}{21}$, and $S_3 = \frac{3}{7} \times \sum_{n=1}^{\infty} \frac{1}{10}^n = \frac{1}{21}$.

4.2 Massive Left Recursion

For grammars with intermingled left recursions such as GRA3, computation of the stochastic values of the items becomes a convoluted task. Consider the start state for GRA3, which is depicted in Figure 10.

Figure 10: Start State of GRA3

I_0:	$[S' \rightarrow \cdot S,$	$1]$
I_1:	$[S \rightarrow \cdot S a_1,$	$S_1]$
I_2:	$[S \rightarrow \cdot B a_2,$	$S_2]$
I_3:	$[S \rightarrow \cdot C a_3,$	$S_3]$
I_4:	$[B \rightarrow \cdot S a_3,$	$S_4]$
I_5:	$[B \rightarrow \cdot B a_2,$	$S_5]$
I_6:	$[B \rightarrow \cdot C a_1,$	$S_6]$
I_7:	$[C \rightarrow \cdot S a_2,$	$S_7]$
I_8:	$[C \rightarrow \cdot B a_3,$	$S_8]$
I_9:	$[C \rightarrow \cdot C a_1,$	$S_9]$
I_{10}:	$[C \rightarrow \cdot a_3 B,$	$S_{10}]$
I_{11}:	$[C \rightarrow \cdot a_3,$	$S_{11}]$

Consider the item I_1. In an attempt to write down a closed expression for the stochastic value S_1, we discover in despair that there is an infinite number of loops to detect, as S is immediately reachable by all non-terminals, and so are the other nonterminals themselves. This intermingling of the loops renders it impossible to write down closed expressions for S_1 through S_{11}.

5 Probabilistic Parse Table Construction for Left Recursive Grammars

In this section, we describe a way of computing item probabilities by encoding the item dependencies in terms of systems of linear equations and solving them by Gaussian Elimination (Strang 1980). This method handles arbitrary context-free grammar including those with left recursions. We incorporate this method with Wright and Wrigley's(1989) algorithm for computing stochastic factors for the parse actions to obtain a table construction algorithm which handles general PCFG. A formal description of the complete table construction algorithm is in the Appendix.

In the following discussion of the algorithm, lower case greek characters such as α and β will denote strings in $(N \cup T)^*$ and upper case alphabets like A and B denote symbols in N unless mentioned otherwise.

5.1 Stochastic Values of Kernel Items

For completeness, we mention briefly here how the stochastic values of items in the kernel set can be computed as proposed by Wright and Wrigley(1989):

The stochastic value of the kernel item $[S' \rightarrow \cdot S]$ in the start state is 1. Let State $m - 1$ be a prior state of the non-start State m. We want to compute the stochastic values of the kernel items of State m. Suppose in State $m-1$ there are k items which are expecting the grammar symbol X, their stochastic values being $S_1, S_2, \ldots, S_k$ respectively. Let $[A_i \rightarrow \alpha_i \cdot X\beta_i, S_i]$ be these item, $i = 1, \ldots, k$. Then the posterior probability of the kernel item $[A_i \rightarrow \alpha_i X \cdot \beta_i]$ of State m given those k items in State i and grammar symbol X as the next symbol seen on the parse stack is $\frac{S_i}{S_X}$, where $S_X = \sum_{i=1}^{k} S_i$.

5.2 Dependency Graph

The inter-dependency of items within a state can be represented most straightforwardly by a dependency forest. If we label each arc by the probability of the rule represented by that item the arc is pointing at, then the posterior probability of an item in a dependency forest is simply the total product of the root item's stochastic value and the arc costs along the path from the root to the item.

This dependency forest can be compacted into a dependency graph in which no item occurs in more than one node. That is, each graph node represents a stochastic item which is a coalesce of all the nodes in the dependency forest representing that particular item. The stochastic value of such an item is thus the sum of the posterior probabilities of the underlying items.

Figure 11 depicts the graphical relations of the items in the example state of GRA2 in Figure 9. We shall not attempt to depict the massively cyclic dependency graph of the start state for GRA3 (Figure 10) here.

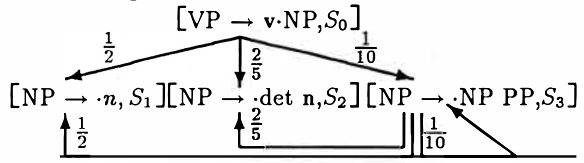


Figure 11: A Dependency Graph

5.3 Generating Linear Equations

Rather than attempting to write down a closed expression for the stochastic value of each item, we resort to creating a system of linear equations in terms of the stochastic values which encapsulate the possibly cyclic dependency structure of the items in the set.

Consider a state Ψ with k items, m of which are kernel items. That is, Ψ is the set of items $\{I_j \mid 1 \leq j \leq k\}$ such that I_j is a kernel item if $1 \leq j \leq m$. Again, let S_j be a variable representing the stochastic value of item I_j. The values of $S_1, \ldots, S_m$ are known since they can be computed as outlined in Section 5.1.

Consider a non-kernel item I_j, $m < j \leq k$. Let $\{I_{j_1}, \ldots, I_{j_{n'}}\}$ be the set of items in Ψ from which there is an arc into I_j in the dependency graph for Ψ. Also, let P_{j_i} denote the arc cost of the arc from item I_{j_i} to I_j. Then, the equation for the stochastic value of I_j, namely S_j, would be:

$$S_j = \sum_{i=1}^{n'} P_{j_i} \times S_{j_i} \qquad (1)$$

Note that Equation (1) is a linear equation of at most $(k-m)$ unknowns, namely $S_{m+1}, \ldots, S_k$. This means that from 1 we have a system of $(k-m)$ linear equations with $(k-m)$ unknowns. This can be solved using standard algorithms like simple Gaussian Elimination (Strang 1980).

The task of generating the equations can be further simplified by the following observations:

1. The cost of any incoming arc of a non-kernel item $I_i = [A_i \rightarrow \cdot\alpha_i, S_i]$ is the production probability of the production $\langle A_i \rightarrow \alpha_i, P_r \rangle$. In other words, $P_{j_i} = P_r$ for $i = 1 \ldots n'$. Equation (1) can then be simplified to $S_j = P_r \times \sum_{i=1}^{n'} S_{j_i}$.
2. Within a state, the non-kernel items representing any X-production have the same set of items with arcs into them. Therefore, these non-kernel items have the same value for $\sum_{x=1}^{n'} S_{j_x}$ (which is similar to the S_X in Section 5.1).

Thus, Equation (1) can be further simplified as $S_j = P_r \times S_{A_j}$ where $S_{A_j} = \sum_{x=1}^{n'} S_{j_x}$. With that, the system of linear equations for each state can be generated efficiently without having to construct explicitly the item dependency graph.

5.3.1 Examples

The system of linear equations for the state depicted in Figures 9 and 11 for grammar GRA2 is as follows:
$$\begin{array}{llll} S_0 = \frac{3}{7} & (Given) & S_2 = \frac{2}{5}(S_0 + S_3) \\ S_1 = \frac{1}{2}(S_0 + S_3) & & S_3 = \frac{1}{10}(S_0 + S_3) \end{array}$$

On solving the equations, we have $S_1 = \frac{5}{21}$, $S_2 = \frac{4}{21}$ and $S_3 = \frac{1}{21}$, which is the same solution as the one obtained by enumeration (Section 4.1).

Similarly, the following system of linear equations is obtained for the start state of massively left recursive grammar GRA3:

$$\begin{array}{ll} S_0 = 1 & S_6 = \frac{1}{2}(S_2 + S_5 + S_8) \\ S_1 = \frac{1}{7}(S_0 + S_1 + S_4 + S_7) & S_7 = \frac{1}{5}(S_3 + S_6 + S_9) \\ S_2 = \frac{4}{7}(S_0 + S_1 + S_4 + S_7) & S_8 = \frac{4}{15}(S_3 + S_6 + S_9) \\ S_3 = \frac{2}{7}(S_0 + S_1 + S_4 + S_7) & S_9 = \frac{1}{15}(S_3 + S_6 + S_9) \\ S_4 = \frac{1}{3}(S_2 + S_5 + S_8) & S_{10} = \frac{1}{3}(S_3 + S_6 + S_9) \\ S_5 = \frac{1}{6}(S_2 + S_5 + S_8) & S_{11} = \frac{2}{15}(S_3 + S_6 + S_9) \end{array}$$

On solving the equations, we have the solutions $1, \frac{29}{77}, \frac{116}{77}, \frac{58}{77}, \frac{64}{77}, \frac{32}{77}, \frac{96}{77}, \frac{3}{7}, \frac{4}{7}, \frac{1}{7}, \frac{5}{7}$ and $\frac{2}{7}$ for the stochastic variables S_0 through S_{11} respectively.

5.4 Solving Linear Equations with Gaussian Elimination

The systems of linear equations generated during table construction can be solved using the popular method *Gaussian Elimination* which can be found in many numerical analysis or linear algebra textbooks (for example, Strang 1980) or linear programming books (such as Vašek Chvátal, 1983). The basic idea is to eliminate the variables one by one by repeated substitutions. For instance, if we have the following set of equations:

$$(1) \quad S_1 = a_{11}S_1 + a_{12}S_2 + \ldots + a_{1n}S_n$$
$$\vdots$$
$$(n) \quad S_n = a_{n1}S_1 + a_{n2}S_2 + \ldots + a_{nn}S_n$$

We can eliminate S_1 and remove equation (1) from the system by substituting, for all occurrences of S_1 in equations (2) through (n), the right hand side of equation (1). We repeatedly remove variables S_1 through S_{n-1} in the same way, until we are left with only one equation with one variable S_n. Having thus obtained the value for S_n, we perform back substitutions until solutions for S_1 through S_n are obtained.

Complexity-wise, Gaussian elimination is a cubic algorithm(Vašek Chvátal, 1983) in terms of the number of variables (ie, the number of items in the closure set). The generation of linear equations per state is also polynomial since we only need to find the stochastic sum expressions — the S_{A_i}'s, for the nonterminals (Point 2 of Section 5.3). These expressions can be obtained by partitioning the items in the state set according to their left hand sides. There are $O(mn)$ possible LR(0) items (hence the size of each state is $O(mn)$) and $O(2^{mn})$ possible sets where n is the number of productions and m the length of the longest right hand side. Hence, asymptotically, the computation of the stochastic values would not affect the complexity of the algorithm, since it has only added an extra polynomial amount of work for each of the exponentially many possible sets.

Of course, we could have used other methods for solving these linear equations, for example, by finding the inverse of the matrix representing the equations(Vašek Chvátal, 1983). It is also plausible that particular characteristics of the equations generated by the construction algorithm can be exploited to derive the equations' solution more efficiently. We shall not discuss further here.

5.5 Stochastic Factors

Since the stochastic values of the terminal items in a parse state are basically posterior probabilities of that item given the root (kernel) item, the computation of the stochastic factors for the parsing actions, which is as presented in Wright and Wrigley(1989), is fairly straightforward. For *shift*-action, say from State i to State $i+1$ on seeing the input symbol x, the corresponding stochastic factor for this action would be S_x, the sum of the stochastic values of all the leaf items in State i which are expecting the symbol x. For *reduce*-action, the stochastic factor is simply the stochastic value S_i of the item representing the reduction, namely $[A_i \rightarrow \alpha_i \cdot, S_i]$ if the reduction is via production $A_i \rightarrow \alpha_i$. For *accept*-action, the stochastic factor is the stochastic value S_n of the item $[S' \rightarrow S\cdot, S_n]$, since acceptance can be treated as a final reduction of the augmented production $S' \rightarrow S$, where S' is the system-introduced start symbol for the grammar.

6 Deferred Probabilities

The introduction of probability created a new criterion for equality between two sets of items: not only must they contain the same items, they must have the same item probability assignment. It is thus possible that we have many (possibly infinite) sets of similar items of differing probability assignments. This is especially so when there are loops amongst the sets of items (ie, the *states*) in the automaton created by the table construction algorithm — there is no guarantee that the differing probability assignments of the recurring states would converge. Even if they do converge eventually, it is still undesirable to have a huge parsing table of which many states have exactly the same underlying item set but differing probabilities.

To remedy this undesirable situation, we introduce a mechanism called *deferred probability* which will guarantee that the item sets converge without duplicating too many of the states. Thus far, we have been precomputing item's stochastic values in an *eager* fashion — propagating the probabilities as early as possible. Deferred probability provides a means to defer propagating certain problematic probability assignments (problematic in the sense that it causes many similar states with differing probability assignments) until appropriate. In the extreme case, probabilities are deferred until *reduction* time, ie, the stochastic factors of REDUCE actions are the respective rule probabilities and all other parse actions have unit stochastic factors. A reasonable postponement, however, would be to defer propagating the probabilities of the kernel items (kernel probabilities) until the following state. By forcing the differing item sets to have some fixed predefined probability assignment (while deferring the propagation of the "real" probabilities until appropriate times), we can prevent excessive duplication of similar states with same items but different probabilities.

To allow for deferred probabilities, we extend the original notion of probabilistic item to contain an additional field q which is the deferred probability for that item. That is, a probabilistic item would have the form $\langle A \rightarrow \alpha \cdot \beta,\ p,\ q\rangle$. The default value of q is 1, meaning that no probability has been deferred. If in the process of constructing the closure states the table-construction program discovers that it is re-creating many states with the same underlying items but with differing probabilities or when it detects a non-converging loop, it might decide to replace that state with one in which the original kernel probabilities are deferred. That is, if the item $\langle A \rightarrow \alpha \cdot \beta,\ p,\ q\rangle$ is a kernel item, and $\beta \neq \epsilon$, we replace it with a deferred item $\langle A \rightarrow \alpha \cdot \beta,\ p',\ \frac{pq}{p'}\rangle$ and proceed to compute the closure of the kernel set as before (ie, ignoring the deferred probabilities). In essence we have reassigned a kernel probability of p' to the kernel items *temporarily* instead of its original probability. It is important that this choice of assignment of p' be fixed with respect to that state. For instance, one assignment would be to impose a uniform probability distribution onto the deferred kernel items, that is, let p' be the probability $\frac{1}{Number\ of\ kernel\ items}$. Another choice is to assign unit probability to each of the kernel items, which allows us to simulate the effect of treating each of the kernel items as if it forms a separate state.

Although in theory it is possible to defer the kernel probabilities until reduction time, in practice it is sufficient to defer it for only one state transition. That is, we recover the deferred probabilities in the next state. We can do this by enabling the propagation of the deferred probabilities in the next state, simply by multiplying back the deferred probabilities q into the kernel probabilities of the next state. In other words, as in Section 5.1, if $[A_i \rightarrow \alpha_i \cdot X\beta_i, S_i, q]$ is in State $m-1$, then the corresponding kernel item in State m would be $[A_i \rightarrow \alpha_i X \cdot \beta_i, \frac{S_i q}{S_X}, 1]$.

7 Concluding Remarks

In this paper, we have presented a method for dealing with left recursions in constructing probabilistic LR parsing tables for left recursive PCFGs. We have described runtime probabilistic LR parsers which use probabilistic parsing table. The table construction method, as outlined in this paper and more formally in the appendix, has been implemented in Common Lisp. The two versions of runtime parsers described in this paper have also been implemented in Common Lisp, and incorporated with various search strategies such as beam-search and best-first search (only for the tree-stack version) for comparison. The programs run successfully on various small toy grammars, including the ones listed in this paper. In future, we hope to experiment with larger grammars such as the one in Fujisaki(1984).

Appendix A. Table Construction Algorithm

A full algorithm for probabilistic LR parsing table construction for general probabilistic context-free grammar is presented here. The deferred probability mechanism as described in Section 6 is employed, the chosen reassignment of kernel probability being the unit probability.

A.1 Auxiliary Functions

A.1.1 CLOSURE

CLOSURE takes a set of ordinary nonprobabilistic LR(0) items and returns the set of LR(0) items which is the closure of the input items. A standard algorithm for CLOSURE can be found in Aho and Ullman(1977).

A.1.2 PROB-CLOSURE

Input: A set of k probabilistic items for some $k \geq 1$: $\{[A_i \rightarrow \alpha_i \cdot \beta_i, p_i, q_i] \mid 1 \leq i \leq k\}$.

Output: A set of probabilistic items which is the closure of the input probabilistic items. Each probabilistic item in the output set carries a stochastic value which is the sum of the posterior probabilities of that item given the input items.

Method:

Step 1: Let
$\mathcal{C} := \text{CLOSURE}(\{[A_i \rightarrow \alpha_i \cdot \beta_i] \mid 1 \leq i \leq k\})$;

Step 2: Suppose k' is the size of $\mathcal{C}$. Let I_i be the i-th item $[A_i \rightarrow \alpha_i . \beta_i]$ in $\mathcal{C}$, $1 \leq i \leq k'$. Also, for each item I_i, let S_i be a variable denoting its stochastic value.

1. For $1 \leq i \leq k$, $S_i := p_i$;
2. Let $\mathcal{E}_B$ be the set of items in $\mathcal{C}$ that are expecting B as the next symbol on the stack. That is, $\mathcal{E}_B$ is the set

$$\{I_j \mid I_j \in \mathcal{C},\ I_j = [A_j \rightarrow \alpha_j \cdot B\beta_j]\}$$

Let $S_B \overset{\text{def}}{=} \sum_{I_j \in \mathcal{E}_B} S_j$, where $B \in N$. For $k < i \leq k'$ such that $I_i = [A_i \rightarrow \cdot \beta_i]$, set $S_i := P_r \times S_{A_i}$, where P_r is the probability of the production $A_i \rightarrow \beta_i$.

Step 3: Solve the system of linear equations generated by **Step 2**, using any standard algorithm such as simple Gaussian Elimination (Strang 1980).

Step 4: Return $\{[A_i \to \alpha \cdot \beta, S_i, q_i] \mid 1 \le i \le k'\}$, where $q_i = 1$ for $k \le i \le k'$.

A.1.3 GOTO

Another useful function in table construction is GOTO($\{I_1 \ldots I_n\}, X$), where the first argument $\{I_1 \ldots I_n\}$ is a set of n probabilistic items and the second argument X a grammar symbol in $(N \cup T)$.

Suppose the probabilistic items in $\{I_1 \ldots I_n\}$ are such that those with symbol X after the dot are $[A_i \to \alpha_i \cdot X\beta_i, S_i, q_i]$, $1 \le i \le k$ for some $1 \le k \le n$. Let S_X be $\sum_{i=1}^{k} S_i$ and set GOTO($\{I_i\}, X$) to be PROB-CLOSURE($\{[A_i \to \alpha_i X \cdot \beta_i, \frac{S_i q_i}{S_X}, 1] \mid 1 \le i \le k\}$).

When $k = 0$, GOTO($\{I_i\}, X$) is undefined.

A.1.4 Sets-of-Items Construction

Let $\mathcal{U}$ be the canonical collection of sets of probabilistic items for the grammar G'. $\mathcal{U}$ can be constructed as described below.

Initially $\mathcal{U}$:= PROB-CLOSURE($\{[S' \to \cdot S, 1]\}$). Repeat the process of applying the GOTO function (as defined in Step A.1.3) with the existing sets in $\mathcal{U}$ and symbols in $(N \cup T)$ to generate new sets to be added to $\mathcal{U}$. If it is detected that an excessive number of states with similar underlying item sets but differing probabilities are created, use a state that is created by deferring the probabilities of the kernel items. That is, suppose the original kernel set is $\{[A_i \to \alpha_i \cdot \beta_i, p_i, q_i] \mid 1 \le i \le k\}$, use instead $\{[A_i \to \alpha_i \cdot \beta_i, 1, p_i q_i] \mid 1 \le i \le k \text{ and } \beta_i \ne \epsilon\}$.

The process stops when no new set can be generated.

Note that equality between two sets of probabilistic items here requires that they contain the same items with equal corresponding stochastic values, as well as deferred probabilities.

A.2 LR Table Construction

The algorithm is very similar to standard LR table construction (Aho and Ullman 1977) except for the additional step to compute the stochastic factor for each action (*shift*, *reduce*, or *accept*).

Given a grammar $G = \langle N, T, R, S \rangle$, we define a corresponding grammar G' with a system-generated start symbol S':

$$\langle N \cup \{S'\}, T, R \cup \{< S' \to S, 1 >\}, S' \rangle.$$

Input: $\mathcal{U}$, the canonical collection of sets of probabilistic items for grammar G'.

Output: If possible, a probabilistic LR parsing table consisting of a parsing action function ACTION and a goto function GOTO.

Method: Let $\mathcal{U} = \{\Psi_0, \Psi_1, \ldots, \Psi_n\}$, where Ψ_0 is that initial set in Sets-of-Items Construction. The states of the parser are then $0, 1, \ldots, n$, with state i being constructed from Ψ_i. The parsing actions for state i are determined as follows:

1. If $[A \to \alpha \cdot a\beta, q_a]$ is in Ψ_i, $a \in T$, and GOTO(Ψ_i, a) $= \Psi_j$, set ACTION$[i, a]$ to $\langle$ "*shift* j", $p_a \rangle$ where p_a is the sum of q_a's – that is the stochastic values of items in Ψ_i with symbol a after the dot.
2. If $[A \to \alpha \cdot, p]$ is in Ψ_i, set ACTION$[i, a]$ to $\langle$ "*reduce* $A \to \alpha$", $p \rangle$ for every $a \in$ FOLLOW(A).
3. If $[S' \to S \cdot, p]$ is in Ψ_i, set ACTION$[i, \$]$ (\$ is an end-of-input marker) to $\langle$ "*accept*", $p \rangle$.

The goto transitions for state i are constructed in the usual way:

4. If GOTO(I_i, A) $= I_j$, set GOTO$[i, A] = j$

All entries not defined by rules (1) through (4) are made "*error*".

The FOLLOW table can be constructed from G by a standard algorithm in Aho and Ullman(1977).

References

Aho, A.V. and Ullman, J.D. 1977. *Principles of Compiler Design.* Addison Wesley.

Fu, K. S. , and Booth, T. L. , 1975. Grammatical Inference: Introduction and Survey — Part II. *IEEE Trans on Sys., Man and Cyber.* SMC-5:409-423.

Fujisaki, T. 1984. An Approach to Stochastic Parsing. *Proceedings of COLING84.*

Strang, G. 1980. *Linear Algebra and Its Applications*, 2nd Ed. Academic Press, New York, NY.

Suppes, P. 1970. Probabilistic Grammars for Natural Languages. *Synthese* 22:95-116.

Tomita, M. 1985. *Efficient Parsing for Natural Language.* Kluwer Academic Publishers, Boston, MA.

Tomita, M. January-June, 1987. An Efficient Augmented Context-Free Parsing Algorithm. *Computational Linguistics* 13(1-2):31-46.

Vašek Chvátal, 1983. *Linear Programming*, Chapter 6.

Wetherall, C. S. 1980. Probabilistic Languages: A Review and Some Open Questions. *Computing Surveys* 12:361-379.

Wright, J.H. and Wrigley, E.N. 1989. Probabilistic LR Parsing for Speech Recognition. *International Parsing Workshop '89*, Carnegie Mellon University, Pittsburgh PA.