Exploiting Language Variants Via Grammar Parsing Having Morphologically Rich Information

Qaiser Abbas
Fachbereich Sprachwissenschaft
Universität Konstanz
78457 Konstanz, Germany
qaiser.abbas@uni-konstanz.de

Abstract

In this paper, the development and evaluation of the Urdu parser is presented along with the comparison of existing resources for the language variants Urdu/Hindi. This parser was given a linguistically rich grammar extracted from a treebank. This context free grammar with sufficient encoded information is comparable with the state of the art parsing requirements for morphologically rich and closely related language variants Urdu/Hindi. The extended parsing model and the linguistically rich grammar together provide us promising parsing results for both the language variants. The parser gives 87% of f-score, which outperforms the multi-path shift-reduce parser for Urdu and a simple Hindi dependency parser with 4.8% and 22% increase in recall, respectively.

1 Introduction

An Urdu invariant of Hindavi came into existence during the muslim rule from 1206 AD to 1858 AD (Khan, 2006). They used Persian/Urdu script for Urdu in contrast to the Devanagari script for Hindavi. The informal versions of the two language variants are quite similar, in fact so similar that they can really be called dialects of a same language. Loose examples would be how a Spanish speaker could comprehend Portuguese or Swedish speaker could comprehend Norwegian. However the formal version of the two languages will be much more different as Urdu vocabulary is influenced heavily from Persian, Arabic and Turkish whilst the emphasis in Hindi is on Sanskrit. Urdu became a literary language after existence of an increasing number of literature during the 18th and the 19th century (McLane, 1970). Urdu/Hindi is the national language of Pakistan and an official language in India. According to a report by the SIL Ethnologue (Lewis, 2013), Urdu/Hindi has 456.2 million speakers in the whole world.

Getting state of the art parsing results for morphologically rich languages (MRLs) is a challenge to date. According to Tsarfaty et al. (2010; 2013), without proper handling of morphological entities in the sentences, promising results for MRLs can not be achieved. Complex morphosyntactic interactions may impose constraints, which lead to explicit encoding of such information. The best broad coverage and robust parsers to date have grammars extracted from treebanks and the depth of information encoded in an annotation correlates with the parsing performance (Tsarfaty et al., 2013).

To fulfill the encoding of information in an annotation, a treebank for Urdu known as the URDU.KON-TB treebank with sufficient encoded information at morphological, POS, syntactic and functional level was constructed (Abbas, 2012). Its annotation was found reliable according to the Krippendorffs α values achieved in (Abbas, 2014) but its reliability or the suitability for machine learning (ML) can be evaluated with the development of an Urdu parser presented in Section 3. A context free grammar (CFG) is extracted from the URDU.KON-TB treebank computationally. The development procedure and the depth of encoded information in the grammar is presented in Section 2. The grammar is then given to an extended dynamic programming parsing model known as the Earley parsing algorithm (Earley, 1970). The extended parsing model for Urdu is then called as the Urdu parser and given in Section 3. This algorithm is language independent and is capable to parse the MRLs like the CKY (Cocke-Kasami-Younger) parsing algorithm as advocated in (Tsarfaty et al., 2013) and (Abbas et al., 2009). Issues faced during the parsing are discussed in Section 4. By applying a rich grammar along with the extended parsing model, promising results obtained are discussed in Section 5. Conclusions along with the future directions are presented in Section 6. Similarly, the related work of language variants is described in Section 1.1, which set a path towards the construction of the Urdu parser.

1.1 Related Work

In the Urdu ParGram project (Butt and King, 2007), the XLE[1] parser is in use. The encoding of LFG grammar in XLE interface is not a simple task. Such a grammar can be encoded only by those persons who have expertise in theoretical linguistics as well. The team of the ParGram project has made a tremendous effort in this regard. This project of Urdu LFG grammar development is still in progress and the parser evaluation results are not available yet. Similarly, the parser for evaluation of the NU-FAST treebank (Abbas et al., 2009) used a built in utility available in the inference engine of the Prolog to parse the Urdu sentences. This utility can only be used if you have a definite clause grammar (DCG) or probabilistic definite clause grammar (PDCG). In this work, a parser was not designed but a built-in prolog parser was used, due to which it was not considered to be the candidate for comparison.

A simple dependency parser for Hindi was developed by Bharati et al. (2009). The parser used a grammar oriented approach, which was designed on the basis of Paninian grammatical model (Begum et al., 2008; Bharati et al., 1995). The annotation scheme was designed on the basis of chunks, intra-chunks and karakas[2]. This scheme (Begum et al., 2008) of dependency structure (DS) is different from the annotation scheme (Abbas, 2012; Abbas, 2014) of phrase structure (PS) and the hyper dependency structure (HDS) of the URDU.KON-TB treebank along with the different data sets used. As compared to phrase/constituent structure, the dependency structure lacks in information at non-terminal nodes (Bharati et al., 2008) and often the information at POS level. This information can also be provided at dependency annotation but people are stick to the standard norms. The Hindi treebank is rich in functional information as compared to morphological, POS and syntactical information. Due to differences in the designs of the simple dependency parser for Hindi and the Urdu parser, only performance results are compared and presented in Section 5.

Ali and Hussain used the MaltParser with its default settings in Urdu dependency parser (Ali and Hussain, 2010). When somebody performs experiments with MaltParser with its default settings then such evaluation results are advised not to be compared according to MaltParser license.[3] The same exercise for parsing Hindi was performed in (Agrawal et al., 2013), but it was clearly mentioned in the work that MaltParser was used for error detection in the annotation of Hindi/Urdu treebank (HUTB).[4] Similarly, the Urdu sentences were parsed in (Bhat et al., 2012b) using the same MaltParser. The experiments were performed to identify the parsing issues of Urdu and a development of parser was not claimed. Moreover, these data-driven systems are highly criticized on a given set of annotated corpus because they are not able to observe all morphological variants of a word form from it (Tsarfaty et al., 2013).

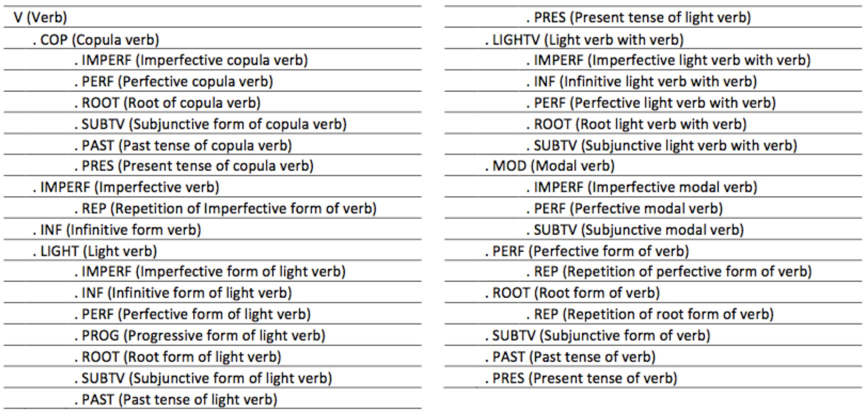
A multi-path shift-reduce parsing algorithm was proposed in (Jiang et al., 2009) for Chinese. Later on, this algorithm was used for Urdu parsing by Mukhtar et al. (2012b). A probabilistic context free grammar (PCFG) developed in (Mukhtar et al., 2011) was given to the multi-path shift-reduce Urdu parsing model. A multi-path shift-reduce parser for Urdu has some limitations. It takes a POS tagged sentence as input and is not able to parse sentences without the POS tagging. The stack used has a fixed memory size, which is not reliable and it can overflow during the parsing of long sentences. A PCFG used in this parsing model is ambiguous (Mukhtar et al., 2012a). Both the fixed memory size and the ambiguous grammar can resist the parsing of long sentences,thats why the parser could not parse the sentences with length more than 10 words (Mukhtar et al., 2012b). In this work, the results were not evaluated properly by using some measure e.g. PARSEVAL. A number of 74 sentences having length not more than 10 words were parsed successfully from 100 sentences, which were then quoted as a 74% of accuracy. The raw corpus used in the development of this parser is partially the same as compared to the Urdu parser (Section 3). A comparative study made is detailed in Section 5.

[1]http://www2.parc.com/isl/groups/nltt/xle/

[2]Karakas are the syntactico-semantic relations between the verbs and other related constituents in a sentence (Bharati et al., 1996)

[3]http://www.maltparser.org/

[4]http://faculty.washington.edu/fxia/treebank/



V (Verb)
. COP (Copula verb)
. IMPERF (Imperfective copula verb)
. PERF (Perfective copula verb)
. ROOT (Root of copula verb)
. SUBTV (Subjunctive form of copula verb)
. PAST (Past tense of copula verb)
. PRES (Present tense of copula verb)
. IMPERF (Imperfective verb)
. REP (Repetition of Imperfective form of verb)
. INF (Infinitive form verb)
. LIGHT (Light verb)
. IMPERF (Imperfective form of light verb)
. INF (Infinitive form of light verb)
. PERF (Perfective form of light verb)
. PROG (Progressive form of light verb)
. ROOT (Root form of light verb)
. SUBTV (Subjunctive form of light verb)
. PAST (Past tense of light verb)
. PRES (Present tense of light verb)
. LIGHTV (Light verb with verb)
. IMPERF (Imperfective light verb with verb)
. INF (Infinitive light verb with verb)
. PERF (Perfective light verb with verb)
. ROOT (Root light verb with verb)
. SUBTV (Subjunctive light verb with verb)
. MOD (Modal verb)
. IMPERF (Imperfective modal verb)
. PERF (Perfective modal verb)
. SUBTV (Subjunctive modal verb)
. PERF (Perfective form of verb)
. REP (Repetition of perfective form of verb)
. ROOT (Root form of verb)
. REP (Repetition of root form of verb)
. SUBTV (Subjunctive form of verb)
. PAST (Past tense of verb)
. PRES (Present tense of verb)

Figure 1: A verb V example from the URDU.KON-TB treebank

2 Setup

The URDU.KON-TB treebank having phrase structure (PS) and the hyper dependency structure (HDS) annotation with rich encoded information (Abbas, 2012; Abbas, 2014) is used for the training of the Urdu parser discussed in Section 3. The treebank has a semi-semantic POS (SSP) tag set, a semi-semantic syntactic (SSS) tag set and a functional (F) tag set. The morphological information in the labeling of the parsers lexicon can be explained by discussing the POS tag set of the URDU.KON-TB treebank.

The SSP tag set hierarchy has 22 main tag categories which are divided into sub-categories based on morphology and semantics. In Figure 1, an example of only a verb V is given. A dot '.' symbol is used for the representation of morphology and semantics at POS level. In Figure 1, the hierarchy of tag labels for verb V is divided into three levels of depth. The first level contains only one label to distinguish a verb V from other POS labels. The second level contains 11 subcategories of V to represent different morphological or functional forms e.g. V.COP (V as a copula verb (Abbas and Raza, 2014)), V.IMPERF (V has an imperfective form (Butt and Rizvi, 2010; Butt and Ramchand, 2001)), V.INF (V has an infinitive form (Butt, 1993; Abbas and Nabi Khan, 2009)), etc. The third level contains further 25 subcategories to represent the morphological information in depth e.g. V.COP.IMPERF (copula verb has an imperfective form), V.COP.PERF (copula verb has a perfective form), V.COP.ROOT (copula verb has a ROOT form), V.COP.PAST (copula verb has a past tense), V.LIGHT.PAST (light verb has a past tense (Butt and Rizvi, 2010; Butt, 2003)), etc. These types of combinations are also possible in case of an auxiliary verb as described in (Abbas, 2014). This short discussion is about the idea of morphological and functional information encoded at POS level. This lexical information can be passed up to the syntactical level because the lexical items have some relationship with other lexical items in a sentence. The detail of syntactic (SSS) and functional (F) tag sets can be seen in (Abbas, 2012).

A stack based extraction Algorithm 1 was designed to extract a context free grammar (CFG) from the URDU.KON-TB treebank. The CFG obtained is then given to the Urdu parser (Section 3) for sentence parsing.

3 Urdu Parser

The URDU.KON-TB treebank is a manually annotated set of 1400 parsed sentences, which were then recorded in a text file on a computer in the form of 1400 bracketed sentences. Initial twenty bracketed-sentences from each hundred were separated in another text file, whose total 280 sentences were then used for the development of a test suite. The test suite was further divided into two halves representing test data and held out data resulting in 140 sentences in each half.

The held out data was used in the development of the Urdu parser, while the test data was used for the evaluation of results after the completion of the Urdu parser. From the first residual text file with 1120 bracketed sentences, a context free grammar (CFG) was extracted using a stack based extraction module given in Algorithm 1. The CFG was then processed by the Urdu parser to produce a grammar database with unique productions. During this process, production type (TYPE) labeling as lexical (L) and non-lexical (NL) at the end of each production was done. The productions having only the lexical items at their right hand side (RHS) were labelled as L and the productions containing non-lexical items on their RHS only were labelled as NL. The purpose of this labeling is to provide an already processed mechanism, through which the Urdu parser can identify a production type L or NL speedily without checking it thoroughly.

Algorithm 1 A CFG extraction algorithm

```
Input: A input and an empty output file
1: $(Sentence, Top, Counter) \leftarrow 0$
2: Read: InputString
3: while $InputString \neq Input.EOF()$ do        ▷ Loop until end of file
4:     if $InputString$ = $ then
5:         Print: $++Sentence$
6:         Read: InputString                      ▷ Read a string from an input file
7:         Write: \n \n                           ▷ Writing two newlines in output file
8:         $(Stack[0], StrArray[0]) \leftarrow \emptyset$   ▷ Initializing stack and array
9:         $(Top, Counter) \leftarrow 0$   ▷ Initializing stack and array variables
10:    end if
11:    if $InputString \neq$ ")" then
12:        $Stack[Top] \leftarrow InputString; Top++$
13:    else                                       ▷ When ')' comes
14:        $Top--$
15:        while $Stack[Top] \neq$ "(" do
16:            $StrArray[Counter] = Stack[Top]$
17:            $Stack[Top] = \emptyset; Counter++; Top--$
18:        end while
19:        $Counter--$
20:        $Stack[Top] = StrArray[Counter]$
21:        $Top++$ and $Check = Counter$
22:        while $Counter \geq 0$ do
23:            if $Counter = Check$ then
24:                Write: $StrArray[Counter]$ →; $StrArray[Counter] = \emptyset$
25:            else
26:                Write: $StrArray[Counter]$ + ""; $StrArray[Counter] = \emptyset$
27:            end if
28:            $Counter--$
29:        end while
30:        Write: \n; $Counter = 0$               ▷ In output file
31:    end if
32:    Read: InputString                          ▷ Read a string from an input file
33: end while
Output: An output file having complete CFG productions for each sentence
```

Without handling the issues discussed in Section 4, the Earley's algorithm simply was not able to provide the state of the art evaluation results for Urdu. These issues caused the parsing discontinuities, due to which extensions are made on the basic algorithm. The extended version of the Urdu parser is depicted in Algorithm 2. The grammar database of the Urdu parser has three fields in the form of a left hand side (LHS), a RHS and a TYPE. After taking a sentence as input, variables are initialized along with a starting value of the chart as `ROOT @ S`. In place of a dot symbol '•' used in the Earley algorithm, here an '@' symbol is used because the dot symbol is extensively used in the hierarchal annotation of the URDU.KON-TB treebank, from which the grammar productions are extracted. The working of the algorithm is similar to the Earley's algorithm except the modifications in the `PREDICTOR()`, `SCANNER()` and a `COMPLETER()` presented in Sections 4.1, 4.2 and 4.7, respectively. Besides these some additional functions are introduced like an `EDITOR()` for an automatic editing of discontinuous parses, a `BUILDER()` for building the parse trees and a `BACKPOINTER()` for calculating the back-pointers.

Algorithm 2 Urdu Parser

```
1: function URDU-PARSER($grammar$)
2:     Input: $Sentence$                          ▷ reading a sentence
3:     $(id, fi, fj, fid) \leftarrow 0$
4:     $chart[0]$.add("$id$", "ROOT @ S", "0,0", " ", "Seed")
5:     for $i \leftarrow 0$ to LENGTH($sentence$[]) do
6:         $scannerFlag \leftarrow$ false, $id \leftarrow 1$
7:         Print: $chart[i] \rightarrow$ (StateId, Rule, @Position, BackPointer, Operation)
8:         for $j \leftarrow 0$ to $ChartSize[i]$ do      ▷ Loop for chart entries
9:             $currentRule \leftarrow chart[i]$.getRule(j).split(" ")
10:            $(tempString, index) \leftarrow$(string-after-@, @Position) in $currentRule$
11:            if $tempString$ = " " then
12:                call COMPLETER()               ▷ calling completer procedure
13:            else
14:                $rs \leftarrow$ All grammar rules with LHS = $tempString$
15:                if $rs.next() \neq$ false then   ▷ checking $rs$ is not empty
16:                    call PREDICTOR()           ▷ calling predictor procedure
17:                else
18:                    call SCANNER()
19:                end if
20:            end if
21:            if $scannerFlag$=false & $j$+1=$chartSize[i]$ & $i \neq$LENGTH($sentence$[]) then
22:                call EDITOR()
23:            end if
24:        end for
25:    end for
26:    call BUILDER()
27: end function
```

During processing of `COMPLETER()`, `PREDICTOR()` and `SCANNER()`, some sort of parsing discontinuities can happen. To check these types of phenomena, an `EDITOR()` will come into an action and it will remove all the faulty states and the charts causing discontinuity up to a right choice of parsing as discussed in Section 4.6. At the end of external loop, the generated parsed-solutions have been stored in the form of the charts with entries, but not in the form of parsed trees. To represent parsed solutions in the form of bracketed parsed trees, a `BUILDER()` function will be executed, which will construct the parsed trees of solutions by manipulating the back-pointers calculated in the `COMPLETER()` function. The `BUILDER()` is able to display all parsed solutions of a given sentence as discussed in Section 4.4 and then the Algorithm 2 for the Urdu parser is exited with the complete generation of charts and bracketed parsed trees. The algorithms called by the Urdu parser are discussed briefly in Section 4 along with their issues.

4 Issues Analysis and Their Evaluation Through Extensions

4.1 Eliminating L Type Useless Predictions

Earley's Predictor() adds useless productions in charts which causes the Urdu parser to end up with a discontinuous parse for a given sentence. Suppose, the current token to be parsed in an input sentence is a proper noun خان 'Khan' and there is a NL type production NP $\rightarrow$ @ N.PROP N.PROP residing in the current chart of the parser, where N.PROP is the tag for proper noun. The '@' symbol before a non-terminal on the RHS of the production is the case of predictor and the non-extended `PREDICTOR()` adds all the available L type productions of N.PROP into the chart from the grammar, even they are not required. Only the relevant production N.PROP $\rightarrow$ @ خان has to be added in the chart. This addition of irrelevant/useless productions is also true for other lexical items e.g. adjectives, personal pronouns, case markers, etc. These useless additions cause the wastage of time and increase the chance of misleading direction towards a discontinuous parse. To resolve this issue, the `PREDICTOR()` of the existing Earley's Algorithm is modified in the Urdu parser as follows.

When the main parsing Algorithm 2 calls the extended `PREDICTOR()` then it checks the type of production either as NL or L in contrast of the Earley algorithm. The handling of NL type productions is same but in dealing of L type of productions, the `PREDICTOR()` is introduced with another condition, which enforces the predictor to add only the relevant productions into the respective charts. It matches the token at the RHS of the predicted-production with the current token in an input sentence. This condition eliminates the limited possibility of misleading direction towards the discontinuous state. The wastage-time factor is reduced to $O(n)$ after the removal of irrelevant matching, where n is the number of tokens in an input sentence.

4.2 Irrelevant POS Selection

In the Earley's parsing algorithm, the `Scanner()` only matches the RHS of the L type production with the current token in a given sentence and causes a selection of L type production with the wrong POS. For example, the verb ہے 'is' has different tags in the grammar. It can act as an auxiliary in a sentence with present tense e.g. VAUX.PRES $\rightarrow$ ہے. It can behave as a copula verb e.g. V.COP.PRES $\rightarrow$ ہے and it can also act as a main verb e.g. V.PRES $\rightarrow$ ہے. This concept of having more than one tag is true for other lexical items. So, if this L type production VAUX.PRES $\rightarrow$ @ ہے is existed in a current chart as right candidate then the `Scanner()` of the Earley algorithm can select other available productions from the grammar due to a check on the RHS only. This can cause the wrong solution or a discontinuous state during the parsing. To remove this issue, the `Scanner()` is extended in the same way as was done with the `PREDICTOR()` in Section 4.1. At this level, this solution solves the issue described below, but it is completed in Section 4.6.

When the `SCANNER()` is called, it extracts a relevant L type production from the grammar after matching the LHS and the RHS completely. It adds only the true L type production in a new chart after checking three additional conditions. At first, it checks that the chart number is not exceeding the length of a sentence. At second, it checks that the token in the current processing L type production is equal to the current token in a given sentence. After that if the `scannerFlag` is false, then the new entry of the matched L type production is added into the new chart. During this process, the `scannerFlag` is set to a true value along with a record of some variables `fi`, `fj`, and `fid`, which will be used in the `EDITOR()` discussed in Section 4.6. By introducing this modification, the possibility of wrong selection of the production from the grammar is abandoned. An issue related to this problem is still remained, which is addressed and resolved in Section 4.6.

4.3 Back-Pointers Calculation

Earley parsing algorithm is a generator or a recognizer and hence can not produce the parse trees or the bracketed trees. To produce the parse trees or the bracketed trees, an unimplemented idea of back-pointers by Earley (1968) is implemented and presented in Algorithm 3. To understand the calculation of the back pointers, a sentence given in example 1 is parsed from the Urdu parser. The charts generated through the Urdu parser are depicted in Figure 2. Only the relevant states are displayed as can be inferred from the non-sequential values of the STATEID column. The column DOT-POSITION is basically the position of '@' in productions.

Algorithm 3 Back Pointer

```
1:  function BACKPOINTER($previousRule$, $dummy@Position$, $i$, $chartSize$, $chart$)
2:      $backPointer \leftarrow$ “”
3:      $tempIndex \leftarrow previousRule$.indexOf(“@”)
4:      $tempIndex \leftarrow tempIndex$-1                          ▷ subtracting index
5:      $NT \leftarrow previousRule$.get($tempIndex$)
6:      $k \leftarrow dummy@Position[0]$
7:      for $l \leftarrow i$ to k step -1 do                       ▷ loop for backward backpointers
8:          if $tempIndex > 0$ then
9:              for $m \leftarrow 0$ to $chartSize[l]$-1 do
10:                 $pString \leftarrow chart[l]$.getRule($m$).split(“ ”)
11:                 $cRule$.add($pString$[])                 ▷ store $pString$ in $cRule$
12:                 $tIndex \leftarrow cRule$.indexOf(“@”)
13:                 if ($NT = cRule[0]$) & ($tIndex$+1 = SIZE(cRule)) then
14:                     $backPointer \leftarrow$ ($l$ + “-” + $chart[l]$.getStateId($m$) +“ ”+$backPointer$)
15:                     $dummy@P = chart[l]$.get@Position($m$).split(”,”)    ▷ getting ‘@’ position
16:                     $l \leftarrow dummy@P[0]$                  ▷ updating loop counter $l$
17:                     $l \leftarrow l + 1$
18:                     $tempIndex \leftarrow tempIndex$-1
19:                     $NT \leftarrow previousRule$[tempIndex]
20:                     break
21:                 else
22:                     cRule.clear()
23:                 end if
24:             end for
25:         else
26:             break
27:         end if
28:     end for
29: end function
```

Algorithm 4 Builder

```
1:  function BUILDER($Sentence[]$, $chartSize[]$, $chart[]$)
2:      $num$=0, $chartN$ = LENGTH($Sentence[]$)
3:      for $count \leftarrow chartSize$[LENGTH($Sentence[]$)]-1 to 0 step -1 do
4:          $dummystr \leftarrow$“S” and $rule \leftarrow chart[chartN]$.getRule($count$).split(“ ”)
5:          if $rule[0] = dummystr$ then
6:              $num = num + 1$
7:              $bp$.add($chartN$+“-”+$chart[chartN]$.getStateId($count$))
8:          end if
9:      end for
10:     $tree[] \leftarrow$ new BTree[$num$]
11:     for $i \leftarrow 0$ to SIZE($bp$)-1 do
12:         $tree[i]$.build($bp$.get($i$), $chart$)          ▷ building tree with pointers
13:     end for
14:     for $i \leftarrow$0 to SIZE($bp$)-1 do
15:         $tree[i]$.prepare($chart$)
16:     end for
17:     for $i \leftarrow$0 to SIZE($bp$)-1 do          ▷ loop for displaying all parsed trees
18:         $bracketedSentenceLength \leftarrow tree[i]$.getSize() and $left \leftarrow 0$
19:         if $bracketedSentenceLength > 0$ then
20:             Print : Bracketed Parse Tree “+($i$+1)+” of “+SIZE($bp$)+”
21:             for $j \leftarrow$0 to $bracketedSentenceLength$-1 do
22:                 if $tree[i]$.getString($j$) = “(” then
23:                     $left = left + 1$ and Print : newline
24:                     for $tab \leftarrow$0 to $left$-1 do
25:                         Print : eight spaces
26:                     end for
27:                     Print : $tree[i]$.getString($j$)
28:                 else if $tree[i]$.getString($j$) = “)” then
29:                     $left = left - 1$ and Print : $tree[i]$.getString($j$)
30:                 else
31:                     Print : space+$tree[i]$.getString($j$)+space
32:                 end if
33:             end for
34:         end if
35:     end for
36: end function
```

(1) ان کا ذکر بھی یہاں ضروری ہے

un	*kA*	*zikr*	*bHI*	*yahAN*
their/P.PERS	of/CM	reference/N	also/PT.INTF	here/ADV.SPT
zarUrI	*hE*			
essential/ADJ.MNR	is/V.COP.PRES			

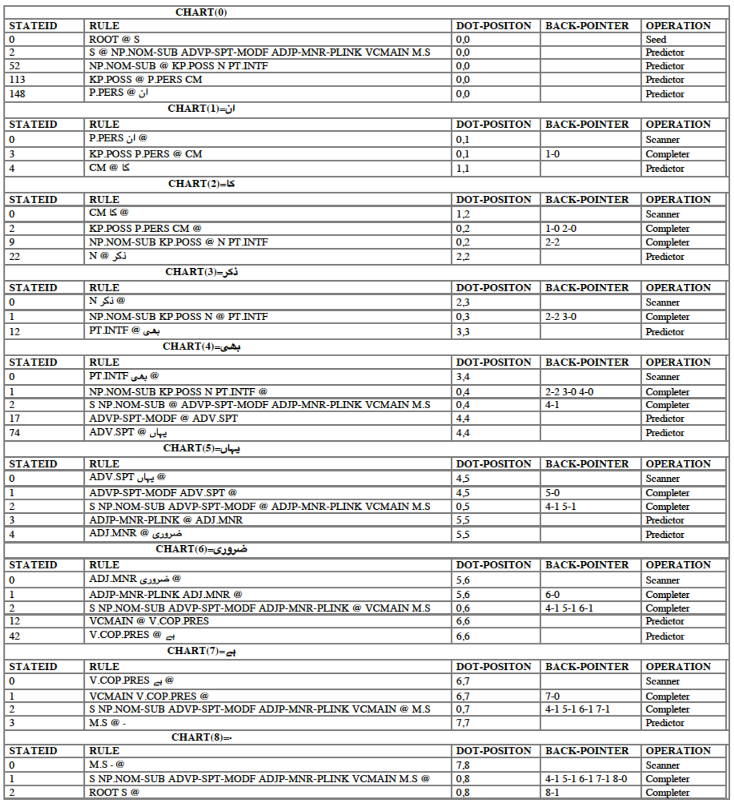
'Their reference is also essential here'

The `COMPLETER()` calls the Algorithm 3 of `BACKPOINTER()` to calculate the values of the back-pointers. For example, during the processing of a production KP.POSS P.PERS @ CM at `STATEID` 3 in chart 1 of Figure 2, a processed non-terminal P.PERS before the '@' in the RHS of the production is located in the chart 1 at 0^{th} position. The located “1-0” value of the `backPointer` is then displayed by the `COMPLETER()` in the same state of the chart. The rest of the back pointers are calculated in the same way. These back-pointers are further used in building the bracketed parse trees discussed in Sections 4.4 and 4.7.

4.4 Building Bracketed Parse Trees

The possible bracketed parse trees are evaluated and displayed by the `BUILDER()` function displayed in Algorithm 4. Both the `BUILDER()` and the `BACKPOINTER()` contribute to shift our Algorithm 2 from a generator to a parser in contrast of the Earley's algorithm. After displaying chart entries in Figure 2 for a sentence given in example 1, the Urdu parser calls the `BUILDER()`. At first, it locates all the solution productions from the last chart and stores their back-pointers in a list e.g. “8-1” value for the solution production ROOT S @ in the last chart of Figure 2. A user defined method `build()` is called then. This method builds an unformatted intermediate bracketed parse tree with the interlinked back-pointers from the chart states and reveals the leaf nodes only as (8-1 (4-1 (2-2 (1-0 ان) (2-0 کا)) (3-0 ذکر) (4-0 بھی)) (5-1 (5-0 یہاں)) (6-1 (6-0 ضروری)) (7-1 (7-0 ہے)) (8-0 .)). This intermediate parse tree can be understood well by looking at the given back-pointers in the respective chart states.

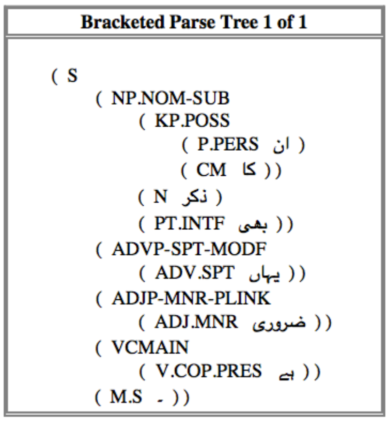
Another user defined method `prepare()` prepares the intermediate parse tree into a complete unformatted parse tree as (S (NP.NOM-SUB (KP.POSS (P.PERS ان) (CM کا)) (N ذکر) (PT.INTF بھی)) (ADVP-SPT-MODF (ADV.SPT یہاں)) (ADJP-MNR-



CHART(0)				
STATEID	RULE	DOT-POSITON	BACK-POINTER	OPERATION
0	ROOT @ S	0,0		Seed
2	S @ NP.NOM-SUB ADVP-SPT-MODF ADJP-MNR-PLINK VCMAIN M.S	0,0		Predictor
52	NP.NOM-SUB @ KP.POSS N PT.INTF	0,0		Predictor
113	KP.POSS @ P.PERS CM	0,0		Predictor
148	P.PERS @ ان	0,0		Predictor
CHART(1)=ان				
STATEID	RULE	DOT-POSITON	BACK-POINTER	OPERATION
0	P.PERS ان @	0,1		Scanner
3	KP.POSS P.PERS @ CM	0,1	1-0	Completer
4	CM @ کا	1,1		Predictor
CHART(2)=کا				
STATEID	RULE	DOT-POSITON	BACK-POINTER	OPERATION
0	CM کا @	1,2		Scanner
2	KP.POSS P.PERS CM @	0,2	1-0 2-0	Completer
9	NP.NOM-SUB KP.POSS @ N PT.INTF	0,2	2-2	Completer
22	N @ ذکر	2,2		Predictor
CHART(3)=ذکر				
STATEID	RULE	DOT-POSITON	BACK-POINTER	OPERATION
0	N ذکر @	2,3		Scanner
1	NP.NOM-SUB KP.POSS N @ PT.INTF	0,3	2-2 3-0	Completer
12	PT.INTF @ بھی	3,3		Predictor
CHART(4)=بھی				
STATEID	RULE	DOT-POSITON	BACK-POINTER	OPERATION
0	PT.INTF بھی @	3,4		Scanner
1	NP.NOM-SUB KP.POSS N PT.INTF @	0,4	2-2 3-0 4-0	Completer
2	S NP.NOM-SUB @ ADVP-SPT-MODF ADJP-MNR-PLINK VCMAIN M.S	0,4	4-1	Completer
17	ADVP-SPT-MODF @ ADV.SPT	4,4		Predictor
74	ADV.SPT @ یہاں	4,4		Predictor
CHART(5)=یہاں				
STATEID	RULE	DOT-POSITON	BACK-POINTER	OPERATION
0	ADV.SPT یہاں @	4,5		Scanner
1	ADVP-SPT-MODF ADV.SPT @	4,5	5-0	Completer
2	S NP.NOM-SUB ADVP-SPT-MODF @ ADJP-MNR-PLINK VCMAIN M.S	0,5	4-1 5-1	Completer
3	ADJP-MNR-PLINK @ ADJ.MNR	5,5		Predictor
4	ADJ.MNR @ ضروری	5,5		Predictor
CHART(6)=ضروری				
STATEID	RULE	DOT-POSITON	BACK-POINTER	OPERATION
0	ADJ.MNR ضروری @	5,6		Scanner
1	ADJP-MNR-PLINK ADJ.MNR @	5,6	6-0	Completer
2	S NP.NOM-SUB ADVP-SPT-MODF ADJP-MNR-PLINK @ VCMAIN M.S	0,6	4-1 5-1 6-1	Completer
12	VCMAIN @ V.COP.PRES	6,6		Predictor
42	V.COP.PRES @ ہے	6,6		Predictor
CHART(7)=ہے				
STATEID	RULE	DOT-POSITON	BACK-POINTER	OPERATION
0	V.COP.PRES ہے @	6,7		Scanner
1	VCMAIN V.COP.PRES @	6,7	7-0	Completer
2	S NP.NOM-SUB ADVP-SPT-MODF ADJP-MNR-PLINK VCMAIN @ M.S	0,7	4-1 5-1 6-1 7-1	Completer
3	M.S @ -	7,7		Predictor
CHART(8)=-				
STATEID	RULE	DOT-POSITON	BACK-POINTER	OPERATION
0	M.S - @	7,8		Scanner
1	S NP.NOM-SUB ADVP-SPT-MODF ADJP-MNR-PLINK VCMAIN M.S @	0,8	4-1 5-1 6-1 7-1 8-0	Completer
2	ROOT S @	0,8	8-1	Completer

Figure 2: A back-pointer calculation example of the Urdu parser

PLINK (ADJ.MNR ضروری)) (VCMAIN (V.COP.PRES ہے)) (M.S .)) . This `prepare()` method only replaces the back-pointers with the LHS of the relevant productions. Finally, the bracketed parse tree is displayed in a formatted way as depicted in Figure 3.



Bracketed Parse Tree 1 of 1

```
( S
    ( NP.NOM-SUB
        ( KP.POSS
            ( P.PERS  ان )
            ( CM  کا ))
        ( N  ذکر )
        ( PT.INTF  بھی ))
    ( ADVP-SPT-MODF
        ( ADV.SPT  یہاں ))
    ( ADJP-MNR-PLINK
        ( ADJ.MNR  ضروری ))
    ( VCMAIN
        ( V.COP.PRES  ہے ))
    ( M.S  - ))
```

Figure 3: An output of the `BUILDER()` method

4.5 Empty Productions

Empty productions are divided into two categories. The first one is related to diacritic productions and the second one is related to non-diacritic productions. It can cause the discontinuity during the parsing because the lexical item may or may not present for both the categories in a given sentence. Only the first category of diacritic productions is discussed here to provide an idea about the issues related to empty productions.

In modern Urdu, the diacritics may or may not appear in the text e.g. آبِ حَیَات *AbE h2ayAt* ‘The water of life’ and تقرِیباً *taqrIban* ‘almost’. The first example is related to compound words and the second one is an independent word. The *zErE-Iz3Afat* (a diacritic for addition) under the last letter بِ *b* of the first word in the first example is still in use in the modern Urdu writing. Similar is the case of *tanwin* (a diacritic for final post-nasalization) on the last letter اً *a* in the second example. There are also other diacritics in use as well e.g. *zEr, zabar, pEsh, taSdId*, etc.

In the grammar of the Urdu parser, a DIA tag is used to represent the diacritics e.g. DIA → *, where ‘*’ represents the absence of a diacritic or an empty production. During parsing, a compound word may or may not appear with a diacritic e.g.شہرِمکہ *Sehr makkah* ‘The city of

Makkah'. This example has two words شہر *Sehr* 'city' and the مکہ *makkah* 'Makkah', but the diacritic is absent between the two words. In such cases, its presence is by default understood by the native speakers. The production extracted from the grammar to handle this compound word is the NP-SPT → @ N.SPT DIA N.PROP.SPT. After processing of the first word *Sehr*/N.SPT by the `SCANNER()`, the production becomes NP-SPT → N.SPT @ DIA N.PROP.SPT. Now, the `PREDICTOR()` deals this DIA empty production implicitly by moving the '@' ahead and adds the updated production NP-SPT → N.SPT DIA @ N.PROP.SPT in the same chart. Similarly, the second word *makkah*/N.PROP.SPT is processed by the `SCANNER()` and the production final state becomes like this NP-SPT → N.SPT DIA N.PROP.SPT @. The problem with this solution adopted from (Aycock and Horspool, 2002) is that it performs the transaction silently with the compound words and also with the non-compound words at such positions where it is not needed. For example, If this is the case as discussed then the solution is perfect, but in the case of the non compound words, if two independent words گھر *gHar* 'The house' and مکہ *makkah* 'Makkah' appear in the same position like compound words e.g. گھر مکہ میں ہے *gHar makkah mEN hE* 'The house is in Makkah', then this solution can not identify the context and it applies the transaction in the same way due to the same POS tagging of *gHar* and the *Sehr*. This solution causes frequent discontinuity during the parsing and its property of self decision at the irrelevant places makes the things more worse.

Due to high frequency of the DIA productions in the grammar, the proposed solution (Aycock and Horspool, 2002) was implemented in the `PREDICTOR()` but the results found were not promising. So, an explicit method to represent the absent value has been chosen, through which an asterisk '*' is usually typed in a given sentence to represent the absence of the diacritics, arguments, lexical items, etc. At present, due to this explicit approach, the Urdu parser is jelling with the grammar without any issue related to empty productions.

4.6 Lexical Dynamic Behavior

The issue is related to a class of words which has the following attributes like the homonym, homograph, homophone, heteronym and the polysemes. A strict definition is considered to these attributes, that means at least the words have the same spelling. The case of homonym words in a strict sense is discussed here and the same concept is applicable on other attributes as well.

For example, the word کی *kI* is a homonym in Urdu. It can behave in two ways e.g. a possessive case marker and a verb. Being a possessive case marker, it contains a possessive meaning 'of' in 2. On the other hand, it contains a meaning of 'did' in 3. In the grammar, this word has different POS tags as a case marker (CM), a perfective verb (V.PERF) and a perfective light verb (V.LIGHT.PERF). Suppose the word 'kI' actually comes as a V.PERF at the end of a given sentence. For its processing, the `Scanner()` can pick up the wrong choice with the CM and the V.LIGHT.PERF, if these choices are available in the current chart at earlier positions as compared to the right choice. Due to this wrong selection, the relevant productions of a verb will not be completed in the next chart and the parser will go into the discontinuous state. To address this issue, the `Scanner()` of the Earley algorithm is modified, which records the failed state in variables fi, fj and fid. These failed states are then utilized by the `EDITOR()` in Algorithm 5, which is called by the Urdu parser to heal this discontinuous state. The failed chart and the states are deleted first. After skipping the wrong choice e.g. the CM → کی in a chart, the next choice from available homonyms is selected and tried to parse. In this way, the next choice V.PERF → کی is located and the i_{th} and j_{th} loop variables of the Urdu parser are set to that choice for further processing. Continuing in this way, the parser finally gets a direction towards the optimal solution.

Algorithm 5 Editor

```
1: function EDITOR($i, id, fi, fj, fid, chart, chartSize$)
2:     Drop and re-initialize $chart[i+1]$
3:     for $z \leftarrow i$ to $fi$+1 step -1 do
4:         Drop and re-initialize $chart[z]$
5:     end for
6:     $rule \leftarrow chart[fi]$.getRule(fj).split(" ")   ▷ splitting rule with space
7:     for $z \leftarrow 0$ to $chartSize[fi]$-1 do
8:         $temprule \leftarrow chart[fi]$.getRule(z).split(" ")
9:         if $temprule[2] = rule[2]$ then
10:            if !($temprule[0] = rule[0]$) then
11:                $j \leftarrow z-1, i \leftarrow fi, id \leftarrow z$
12:                break
13:            end if
14:        end if
15:    end for
16: end function
```

(2) جولیا کی کتاب

jUlIA=kI	*kitAb*
Julia.Fem.Sg=Poss	book.Fem.Sg

‘The book of Julia’

(3) اس نے ایک بات کی ہے

us=nE	*Ek*	*bAt*	*kI*	*hE*
he.Sg=Erg	a	talk.Fem.Sg	do.Perf.Sg	be.Pres.Sg

‘He did a talk’

4.7 Subordinate Clause Limitations

Basically, the issue is related to conjuncted sub-sentences or the subordinate clause, when the number of conjuncted sub-sentences becomes greater than one. The issue does not appear often and it is related to the NL type productions, specially the conjuncted sub-sentences denoted by SBAR as below. A sentence of 23 tokens with two conjuncted sub-sentences highlighted with the SBAR is an evidence of this issue. During the processing of a production for the sentence marker M.S → - @ in the last (23rd) chart, the order of the complete productions should be as follows. The ‘@’ at the end represents the complete status of the productions.

M.S →	- @
SBAR →	C.SBORD NP.NOM-SUB **SBAR** ADVP-MNR-MODF NP.NOM-MNR-OBJ VCMAIN M.S @
S →	KP-INST-MODF KP.DAT-SUB NP.NOM-OBJ VCMAIN **SBAR** @

But, unfortunately, the parser went into a discontinuous state during the processing of the last chart with the following productions.

M.S →	- @
SBAR →	C.SBORD NP.NOM-SUB SBAR ADVP-MNR-MODF NP.NOM-MNR-OBJ VCMAIN M.S @
SBAR →	C.SBORD NP.NOM-SUB SBAR @ ADVP-MNR-MODF NP.NOM-MNR-OBJ VCMAIN M.S
ADVP-MNR-MODF →	@ ADV.MNR

Up to completion of the first SBAR → . . . @ production, the parser performed well. Then `Completer()` went back to search another production which contained an incomplete non-terminal SBAR having ‘@’ before it e.g. @ SBAR. The `Completer()` made a fault there in chart 12 in the presence of wrong choices at higher precedence. It found an incomplete SBAR production. After moving the ‘@’ forward, it added the updated production in the last chart as can be seen in the given productions. Afterwards, the `PREDICTOR()` became activated by seeing the ‘@’ before the ADVP-MNR-MODF and the parser went into a wrong direction. To resolve this issue, it is needed to allow the Earley’s `Completer()` to go back further until a successful parse. The description of the extended `Completer()` is as follows.

When the Urdu parser called the `COMPLETER()`, it first sets the `completerCheck` flag to false, which will be used to back track a right choice among the NL type productions. After calculating the back-pointers, the updated production entry is then added and printed by setting the `completerCheck` to true. If the `completerCheck` is found to be true and the chart number is less than the length of a sentence then a solution has been found and there is no need to go back. However, if the `completerCheck` is found to be true and the chart number is greater or equal to the length of a sentence then the `COMPLETER()` is allowed to back track by setting its flag to its default value.

5 Results

The division of training and test data is discussed in Section 3. To make the test data more valuable and reliable for results, the beginning ten sentences from each hundred of 1400 sentences of the URDU.KON-TB treebank were selected. The test data so contained 140 sentences in all. In test data, the minimum, average and the maximum length is found to be 5, 13.73 and 46 words per sentence. All items which can exists in a normal text are considered e.g. punctuation, null elements, diacritics, headings, regard titles, Hadees (the statements of prophets), antecedents and anaphors within a sentence, and others except the unknown words, which will be dealt in future. The PARSEVAL measures are used to evaluate the results. The PARSEVAL measures are calculated in two ways which are depicted in Table 1.

At first, the values as per columns headings in Table 1 are calculated on the basis of constituents for each individual sentence. Then these values are stored in a text file with these headings. The values existed in each column of the text file are summed up and then divided by the total number of 140 values in each column. The results thus obtained are recorded in a row A-1 of Table 1 on average basis. Similarly, all the values in the Length, Matched, Gold and the Test columns are summed up individually from that text file and their sums are recorded as can be seen in row T-2 of the table. Their respective results for the Precision, Recall, F-score and the Crossing Brackets are calculated

	Sentences	Length	Matched	Gold	Test	Precision	Recall	F-score	Crossing
A-1	140	13.73	17	22	18	0.952	0.811	0.848	2
T-2	140	1922	2449	3107	2531	0.968	0.788	0.869	329

Table 1: Evaluation results of the Urdu parser

from these sums, which is a standard method of calculation.

The Urdu parser outperforms the simple Hindi dependency parser by Bharati et al. (2009) with an additional recall of 22%. In (Bharati et al., 2009), only precision and recall percentages are given. Thats why only the precision and recall percentages of labeled attachment (LA) are compared. For chunks, intra-chunks and karakas, the precision percentages of LA (LA-P) achieved by the simple Hindi dependency parser are 82.3%, 71.2% and 74.1%, respectively. The average of these LA-P percentages is 75.9%, which is 20.9% less precision than the Urdu parser in row T-2. Similarly, Hindi dependency parser achieved LA recalls in case of chunks, intra-chunks and karakas as 65.4%, 58.2% and 46.7% respectively. The average of these percentages is calculated as 56.8%, which is now the final LA recall percentage of the Hindi dependency parser. For comparison, the recall percentage of the Urdu parser used is mentioned in row T-2 as 78.8%. The values obtained for the language variant parsers concludes that the Urdu parser outperforms the simple Hindi dependency parser with 22% increase in recall.

Multi-path shift-reduce parser (Mukhtar et al., 2012b) for Urdu parsed 74 sentences successfully out of 100 and it was then reported as a 74% of accuracy. This evaluation is very weak because the successful parsed sentences were not compared with the gold standard. Recall is a value obtained through dividing the Matched constituents with the constituents available in the Gold data. As recall percentage in our case is 78.8%, so we can say that the Urdu parser beats the multi-path shift-reduce parser with a 4.8% increase in recall. On the other hand, from the first 100 sentences of the test data, the Urdu parser provides 89 sentences with parsed solutions. Comparatively, the Urdu parser has 15% more accuracy than the Multi-path shift-reduce parser, but the parsed solutions were not compared with the Gold data. So, by considering the safe side, we can repeat our argument that the Urdu parser beats the multi-path shift-reduce parser with a 4.8% increase in recall.

6 Conclusion

The extended Urdu parser with rich encoded information in the form of a grammar is a state of the art parsing candidate for morphologically rich language variant Urdu. After removal of issues, the output of the parser is so directed, speedy and refined in a sense that no extra or the irrelevant L type productions can be introduced by the Urdu parser. It is really hard now that the Urdu parser will select a wrong choice of production. If it happens then the Urdu parser has a tendency to correct itself automatically. These all features enables the Urdu parser comparable or better than the state of the art in the domain of both the language variants. Urdu parser can help the linguists analyze the Urdu sentences computationally and can be useful in Urdu language processing and machine learning domains. By using this parser, the limited size of the URDU.KON-TB treebank can also be increased. This can be done after getting the partial parsed trees of unknown sentences. These partial parsed trees can be corrected and then imported into the URDU.KON-TB treebank.

Acknowledgment

I would like to express my sincere gratitude to my Supervisor Prof. Dr. Miriam Butt for her motivation, enthusiasm, and immense knowledge. Her guidance helped me in all the time of this work.

References

Qaiser Abbas and A Nabi Khan. 2009. Lexical Functional Grammar For Urdu Modal Verbs. In *Emerging Technologies, 2009. ICET 2009. International Conference on*, pages 7–12. IEEE.

Qaiser Abbas and Ghulam Raza. 2014. A Computational Classification Of Urdu Dynamic Copula Verb. *International Journal of Computer Applications*, 85(10):1–12, January.

Qaiser Abbas, Nayyara Karamat, and Sadia Niazi. 2009. Development Of Tree-Bank Based Probabilistic Grammar For Urdu Language. *International Journal of Electrical & Computer Science*, 9(09):231–235.

Qaiser Abbas. 2012. Building A Hierarchical Annotated Corpus Of Urdu: The URDU.KON-TB

Treebank. *Lecture Notes in Computer Science*, 7181(1):66–79.

Qaiser Abbas. 2014. Semi-Semantic Part Of Speech Annotation And Evaluation. In *Proceedings of 8th ACL Linguistic Annotation Workshop*, pages 75–81, Dublin, Ireland. Association for Computational Linguistics.

Bhasha Agrawal, Rahul Agarwal, Samar Husain, and Dipti M Sharma. 2013. An Automatic Approach To Treebank Error Detection Using A Dependency Parser. In *Computational Linguistics and Intelligent Text Processing*, pages 294–303. Springer.

Wajid Ali and Sarmad Hussain. 2010. Urdu Dependency Parser: A Data-Driven Approach. In *Proceedings of Conference on Language and Technology (CLT10)*.

John Aycock and R Nigel Horspool. 2002. Practical Earley Parsing. *The Computer Journal*, 45(6):620–630.

Rafiya Begum, Samar Husain, Arun Dhwaj, Dipti Misra Sharma, Lakshmi Bai, and Rajeev Sangal. 2008. Dependency Annotation Scheme For Indian Languages. In *IJCNLP*, pages 721–726.

Akshar Bharati, Vineet Chaitanya, Rajeev Sangal, and KV Ramakrishnamacharyulu. 1995. *Natural Language Processing: A Paninian Perspective*. Prentice-Hall of India New Delhi.

Akshar Bharati, Medhavi Bhatia, Vineet Chaitanya, and Rajeev Sangal. 1996. Paninian Grammar Framework Applied To English. Technical report, Technical Report TRCS-96-238, CSE, IIT Kanpur.

Akshar Bharati, Samar Husain, Dipti Misra Sharma, and Rajeev Sangal. 2008. A Two-Stage Constraint Based Dependency Parser For Free Word Order Languages. In *Proceedings of the COLIPS International Conference on Asian Language Processing 2008 (IALP)*.

Akshar Bharati, Mridul Gupta, Vineet Yadav, Karthik Gali, and Dipti Misra Sharma. 2009. Simple Parser For Indian Languages In A Dependency Framework. In *Proceedings of the Third Linguistic Annotation Workshop*, pages 162–165. Association for Computational Linguistics.

Riyaz Ahmad Bhat, Sambhav Jain, and Dipti Misra Sharma. 2012b. Experiments On Dependency Parsing Of Urdu. In *In Proceedings of The 11th International Workshop on Treebanks and Linguistic Theories (TLT11)*.

Miriam Butt and Tracy Holloway King. 2007. Urdu In A Parallel Grammar Development Environment. *Language Resources and Evaluation*, 41(2):191–207.

Miriam Butt and Gillian Ramchand. 2001. Complex Aspectual Structure In Hindi/Urdu. *M. Liakata, B. Jensen, & D. Maillat, Eds*, pages 1–30.

Miriam Butt and Jafar Rizvi. 2010. Tense And Aspect In Urdu. *Layers of Aspect. Stanford: CSLI Publications*.

Miriam Butt. 1993. Hindi-Urdu Infinitives As NPs. *South Asian Language Review: Special Issue on Studies in Hindi-Urdu*, 3(1):51–72.

Miriam Butt. 2003. The Light Verb Jungle. In *Workshop on Multi-Verb Constructions*.

Jay Clark Earley. 1968. *An Efficient Context-Free Parsing Algorithm*. Ph.D. thesis, Carnegie Mellon University, Pittsburgh, PA, USA. AAI6907901.

Jay Earley. 1970. An Efficient Context-Free Parsing Algorithm. *Communications of the ACM*, 13(2):94–102.

Wenbin Jiang, Hao Xiong, and Qun Liu. 2009. Mutipath Shift-Reduce Parsing With Online Training. *CIPS-ParsEval-2009 shared task*.

Abdul Jamil Khan. 2006. *Urdu/Hindi: An Artificial Divide: African Heritage, Mesopotamian Roots, Indian Culture & Britiah Colonialism*. Algora Pub.

Gary F. Simons & Charles D. Fennig Lewis, M. Paul. 2013. *Ethnologue: Languages Of The World, 17th Edition*. Dallas: SIL International.

John R McLane. 1970. *The Political Awakening In India*. Prentice Hall.

Neelam Mukhtar, Mohammad Abid Khan, and Fatima Tuz Zuhra. 2011. Probabilistic Context Free Grammar For Urdu. *Linguistic and Literature Review*, 1(1):86–94.

Neelam Mukhtar, Mohammad Abid Khan, and Fatima Tuz Zuhra. 2012a. Algorithm For Developing Urdu Probabilistic Parser. *International journal of Electrical and Computer Sciences*, 12(3):57–66.

Neelam Mukhtar, Mohammad Abid Khan, Fatima Tuz Zuhra, and Nadia Chiragh. 2012b. Implementation Of Urdu Probabilistic Parser. *International Journal of Computational Linguistics (IJCL)*, 3(1):12–20.

Reut Tsarfaty, Djamé Seddah, Yoav Goldberg, Sandra Kuebler, Marie Candito, Jennifer Foster, Yannick Versley, Ines Rehbein, and Lamia Tounsi. 2010. Statistical Parsing Of Morphologically Rich Languages (SPMRL): What, How And Whither. In *Proceedings of the NAACL HLT 2010 First Workshop on Statistical Parsing of Morphologically-Rich Languages*, pages 1–12. Association for Computational Linguistics.

Reut Tsarfaty, Djamé Seddah, Sandra Kübler, and Joakim Nivre. 2013. Parsing Morphologically Rich Languages: Introduction To The Special Issue. *Computational Linguistics*, 39(1):15–22.