Natural Language Engineering **22** (5): 775–810. © Cambridge University Press 2015
doi:10.1017/S1351324915000133

Morphologically rich Urdu grammar parsing using Earley algorithm

QAISER ABBAS

Fachbereich Sprachwissenschaft, Universität Konstanz, 78457 Konstanz, Germany
e-mail: qaiser.abbas@uni-konstanz.de

(*Received 13 August 2013; revised 28 February 2015; accepted 2 March 2015; first published online 16 April 2015*)

Abstract

This work presents the development and evaluation of an extended Urdu parser. It further focuses on issues related to this parser and describes the changes made in the Earley algorithm to get accurate and relevant results from the Urdu parser. The parser makes use of a morphologically rich context free grammar extracted from a linguistically-rich Urdu treebank. This grammar with sufficient encoded information is comparable with the state-of-the-art parsing requirements for the morphologically rich Urdu language. The extended parsing model and the linguistically rich extracted-grammar both provide us better evaluation results in Urdu/Hindi parsing domain. The parser gives 87% of f-score, which outperforms the existing parsing work of Urdu/Hindi based on the tree-banking approach.

1 Introduction

The language Hindavi has its origin in Khariboli dialect. An Urdu invariant of Hindavi came into existence during the Muslim rule from 1206 AD to 1858 AD (Khan 2006). The Muslims used Persian/Urdu script for Urdu in contrast with the Devanagari script for Hindavi. Both Urdu and Hindavi (Hindi) are almost same in structure and vocabulary, but Urdu became a literary language after the production of an increasing bulk of the literature during the eighteenth and the nineteenth century (McLane 1970). Urdu/Hindi is the national language of Pakistan and an official language in India. According to a report by SIL Ethnologue (Lewis 2013), Urdu/Hindi has 456.2 million speakers in the whole world. Urdu is an under-resourced language and is in need of a number of resources to remain alive in the computational literary world.

State-of-the-art parsing systems are based on treebank grammars but unfortunately, for both, constituency and dependency parsing, treebank based techniques are suffering in case of morphologically rich languages (MRLs) such as Czech (Collins *et al.* 1999), German (Dubey and Keller 2003), Italian (Corazza *et al.* 2004), French (Arun and Keller 2005), Modern Standard Arabic (Kulick *et al.* 2006), Modern Hebrew (Tsarfaty and Sima'an 2007) and many others (Tsarfaty *et al.* 2010; Abbas 2014b). The data driven parsing models used for dependency parsing are not guaranteed to observe all morphological variants of word form. Efficient

https://doi.org/10.1017/S1351324915000133 Published online by Cambridge University Press

parsing results for MRLs are hard to achieve without explicit encoding of linguistic information (Tsarfaty *et al.* 2013; Abbas 2014b).

Getting state-of-the-art parsing results for an MRL is a challenge till date. According to Tsarfaty *et al.* (2010, 2013), without proper handling of morphological entities in sentences, promising results for MRLs cannot be achieved. To overcome this problem, one approach is to embed solutions in a parser in the form of morphological component for segmentation and a syntactic component for parsing. A parser generates a formal output by looking at the events in data after assuming independence between the events. Complex morpho-syntactic interactions among events may impose constraints, which lead to explicit encoding of morphological information at the syntactic level. Assigning a correct morphological signature to each word in the presence of extreme data sparseness is almost impossible, due to which this solution is suffering. However, in the other state-of-the-art approach, the best broad coverage and robust parsers available to date have grammars extracted from treebanks, which are a collection of syntactically annotated sentences by humans. In these treebanks, the depth of information encoded in their annotation positively correlates with the parsing performance (Tsarfaty *et al.* 2013). However, the languages lying in the category of MRLs are not a single homogenous class of languages due to their own cross lingual-variations such as the extent of morphology, flexible ordering, syncretism, fine-grained and unambiguous morphological markers, etc.

Tsarfaty *et al.* (2010) discussed issues related to encoding of morphological information and the annotation schemes to be adopted for MRLs. These issues highly advocate the need of treebanks annotated with sufficient morphological information. To fulfill this purpose, a treebank for Urdu called the URDU.KON-TB treebank with sufficient information encoded at morphological, part of speech (POS), syntactic and functional level was constructed by Abbas (2012). The reliability of the treebank annotation scheme can be evaluated manually and automatically. For manual evaluation of the URDU.KON-TB treebank, a statistical evaluation of this treebank (the URDU.KON-TB treebank) was performed through annotated data of annotators (Abbas 2014a; 2014c), through which it was concluded that the annotation scheme is reliable, but the answer of a question that the treebank is suitable for machine learning (ML) was not searched at that time and can be computed through automatic parsing. To achieve this objective, an automated evaluation of the treebank through parser is attempted. Successively, a parser is developed, provided with a grammar. This 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 3.

Extracted grammar mentioned above is then given to an extended dynamic programming parsing-model which is known as the Earley algorithm (Earley 1970). The extensions made to the Earley's algorithm are given in Section 4. This algorithm is language independent and is capable to parse MRLs like the Cocke–Younger–Kasami (CYK) algorithm as advocated in Tsarfaty *et al.* (2013) and Abbas *et al.* (2009). The reason that the Earley's algorithm was adopted instead of the CYK algorithm is that the CYK takes grammar in Chomsky normal form (CNF) and produces parse trees in the same form or binary trees, while the Earley algorithm

https://doi.org/10.1017/S1351324915000133 Published online by Cambridge University Press

does not need a special form of grammar and produces parse trees in any form, hence well suited to treebank based grammars e.g, a CFG. A Chomsky normal form can be converted into a CFG computationally if the CYK algorithm is desired to be used but a model (Earley algorithm) with less issues was adopted because more issues in addition to conversion (if CYK would become the choice at that time) were expected during development as discussed in Section 4.1. Grammar extracted from the URDU.KON-TB treebank is in the form of CFG, so it was better to use the Earley algorithm directly rather than the CYK algorithm (after conversion of extracted CFG into Chomsky normal form). Both algorithms have the same time complexity $O(n^3)$. This $O(n^3)$ time complexity is only the upper bound of the Earley algorithm and it performs better than the CYK algorithms on some grammars in $O(n^2)$ time (Hopcroft *et al*. 2001). Similarly, the space complexity of CYK algorithm is $O(n^2)$, while the Earley algorithm has the $O(n)$ space complexity (Earley 1970). Despite this, both algorithms are competitors of each other (Sikkel and Nijholt 1997) and both can be used in accordance with the needs.

The related work of parsers development in the domain of Urdu/Hindi language is presented shortly in Section 2 along with their designs and techniques, which set a path toward the construction of a state-of-the-art Urdu parser. Before the construction of the Urdu parser, the preliminary work needed is presented in Section 3, which includes the description of the URDU.KON-TB treebank (Section 3.1) and how the linguistically rich CFG is extracted from this treebank computationally (Section 3.2). To see the reliability of the URDU.KON-TB treebank for machine learning, an Urdu parser was constructed and its algorithmic design has been described in Section 4. The issues removed from the Earley algorithm to improve accuracy and the extensions applied to transform the Earley recognizer to Urdu parser are presented in Section 4.1. After which a run example of the Urdu parser is discussed in Section 5. By applying a sufficiently rich grammar along with the extended parsing model, promising results are obtained and discussed in Section 6. The section initially describes the training and test data and then the Urdu parser results are evaluated through PARSEVAL measure. Finally, the comparison of results with other parsers is presented in the end of this Section 6. Section 7 concludes the work of Urdu parser having morphologically rich grammar with future directions. This parser can help linguists analyze the Urdu sentences computationally and it can be useful in Urdu language processing (ULP) and machine learning domains. By using this parser, the size of the treebank can also be increased. This can be done after getting partial parsed trees of unknown sentences. These partial parsed trees can be corrected and then imported into the URDU.KON-TB treebank. Using this methodology, one can speedily increase the size of a treebank.

2 Related work

As for as the works related to the Urdu parser development are concerned, the first attempt was made in the ParGram[1] project by Butt and King (2007) using an lexical functional grammar (LFG) framework, then a two stage constraint based

[1] http://ling.uni-konstanz.de/pages/home/pargram_urdu/

https://doi.org/10.1017/S1351324915000133 Published online by Cambridge University Press

dependency parser (CBP) for Hindi was introduced by Bharati *et al.* (2008). Similarly, the NU-FAST treebank based grammar was used to parse the Urdu sentences using a built-in Prolog parser by Abbas *et al.* (2009) and then a dependency based parser was developed by Bharati *et al.* (2009) for Hindi. Later, a dependency parser for Urdu was tried by Ali and Hussain (2010) using MaltParser (Nivre *et al.* 2007). Similarly, some experiments on dependency parsing for Urdu were performed by Bhat *et al.* (2012) using MaltParser and minimum-spanning tree (MST; McDonald *et al.* 2005) parsers. In the same year, an Urdu probabilistic parser, using shift-reduce algorithm, was developed by Mukhtar *et al.* (2012).

In the Urdu ParGram project (Butt and King 2007), the XLE[2] parser has been used. The encoding of lexical functional grammar in XLE interface is not a simple task. The persons having expertise in theoretical linguistics can only be able to encode such a grammar. The team of the ParGram project made a tremendous effort in this regard. The project of Urdu lexical functional grammar development is in progress and the grammar evaluation results are not yet available.

A two stage dependency parser for Hindi by Bharati *et al.* (2008) was trained on the Hindi treebank (Begum *et al.* 2008), which was annotated according to the Paninian grammatical model (Begum *et al.* 2008). The annotation scheme was designed on the basis of chunks, intra-chunks and karakas[3]. The scheme adopted in Begum *et al.* (2008) of dependency structure (DS) is different from the annotation scheme of phrase structure (PS) and the hyper dependency structure (HDS) of the URDU.KON-TB treebank (Abbas 2012) along with the different data sets used. In comparison to phrase/constituent structure, the DS lacks in information at non-terminal nodes (Bharati *et al.* 2008) and often the information at POS level. This information can be provided at dependency annotation but people are stick to the standard norms. The example of PS and DS is discussed next along with the Figure 1. The Hindi treebank is rich in functional information as compared to morphological, POS and syntactical information. Due to differences in designs of the two stage dependency parser for Hindi and the Urdu parser, only the results are compared, which are presented in Section 6.

PS and DS are the two traditional annotation structures for treebank annotation. A simple but large difference between phrase and dependency annotation structure is that in PS annotation, the nodes represent phrases or constituents only e.g. a noun phrase (NP) and a verb phrase (VP) as shown in Figure 1(a) for a sentence *Michael loves Sara*, while in a DS, the nodes represent head words (with or without its syntactic tag) of its dependent words as shown in Figure 1(b) for the same sentence. The PS annotation was first introduced by Chomsky (1956) and its main focus is on building small simple constituents or phrases from the words of a sentence as shown in Figure 1(a). The DS annotation has its focus on building relations between the head word and its dependents. It was first proposed by Tesnière. After his death, this theory of DS was presented in Tesnière and Fourquet (1959). In the beginning, there was no information coded to the DS as depicted in Figure 1(b) but later on

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

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

https://doi.org/10.1017/S1351324915000133 Published online by Cambridge University Press

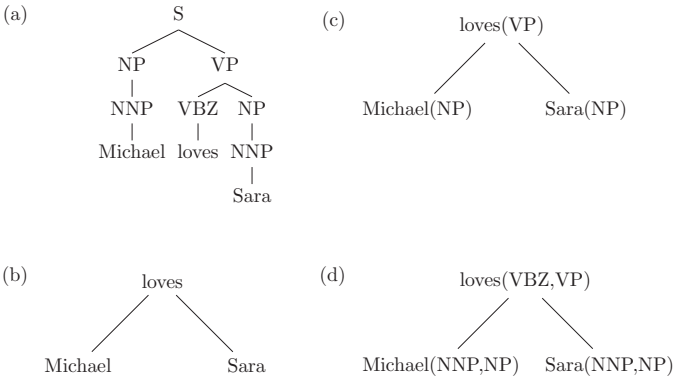


Fig. 1. Phrase and dependency structure example.

the syntactic annotation became the part of DS presented in Figure 1(c). Further information like POS can be the part of DS annotation shown in Figure 1(d).

The Prolog parser used for evaluation of the NU-FAST treebank (Abbas *et al.* 2009) was in fact not the case of parser development, which means that a built-in utility in the inference engine of Prolog was used to parse the sentences. This utility can only be used if you have a definite clause grammar (DCG) or probabilistic definite clause grammar (PDCG). A CFG was extracted computationally through the NU-FAST treebank and then probabilities of grammar rules were counted. Through this process, a probabilistic context free grammar (PCFG) was obtained, which was converted into a probabilistic definite clause grammar format computationally. This probabilistic definite clause grammar was then given to the built-in utility of Prolog inference engine, which is basically a built-in Prolog parser to parse natural languages. Basically, parser was not developed but a built-in parser was used, which concludes that the work is not an algorithmic development of parser that can be compared as explicitly mentioned in Abbas *et al.* (2009).

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 (Bharati *et al.* 1995; Begum *et al.* 2008) discussed earlier. The differences between the two stage dependency parser for Hindi (Bharati *et al.* 2008) and the Urdu parser presented earlier are also applied in the case of the simple dependency parser. Due to which only performance results can be compared but not the designs. The maximum results reported in labeled-attachment recall were 65.4% and 58.2% for chunks and intra-chunks, respectively. An average recall of karakas is calculated as 46.67%. A comparative study has been performed and presented in Section 6.

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.[4] The same exercise for parsing Hindi

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

https://doi.org/10.1017/S1351324915000133 Published online by Cambridge University Press

was performed by Agrawal *et al.* (2013), but it was clearly mentioned in the work that MaltParser was being used for error detection in the annotation of Hindi/Urdu treebank (HUTB).[5] Similarly, the Urdu sentences were parsed by Bhat *et al.* (2012) using the same MaltParser. The experiments were performed to identify the parsing issues of Urdu and a development of parser was not claimed, neither it is the case of parser development. Moreover, the data-driven systems are highly criticized on a given set of annotated corpus because these systems 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 by Jiang *et al.* (2009) for Chinese. Later, this algorithm was used for Urdu parsing by Mukhtar *et al.* (2012). A probabilistic context free grammar developed by Mukhtar *et al.* (2011) was given to the multi-path shift-reduce Urdu parsing model. It takes a POS tagged sentence as input. A *stack* and a *queue* are initialized with a null and a tagged word sequence, respectively and then this initialization *ini* is stored in an array V. The total bunches of transitions are $2|x| - 1$ for which a *step* loop executes. A *state* loop runs for each state in V. This loop contains another *action* loop that decides about shift or reduce action to be taken on each state. After applying the action on a state, it is stored in a *next* variable which is then recorded in a buffer BUF. When the *state* loop ends, the BUF contains all constructed states in it. From which K best states are stored further in an array V. The *step* loop continues and the whole process is repeated until it terminates. Finally, a tree is derived from the best available states in V and displayed accordingly.

A multi-path shift-reduce parser (MPSRP) for Urdu has some limitations. It takes a tagged sentence as input and is not able to parse sentences without POS tagging. The stack used has a fixed memory size, which is not reliable and it can overflow during parsing of long sentences. A probabilistic context free grammar used in this parsing model is ambiguous (Mukhtar *et al.* 2012). Both the fixed memory size and ambiguous grammar can resist the parsing of long sentences, that is why the parser could not parse sentences with length more than ten words (Mukhtar *et al.* 2012). The results were not evaluated properly by using some measure e.g. PARSEVAL. A number of seventy-four sentences having length not more than ten words were parsed successfully from 100 sentences, which were then quoted as a 74% of accuracy. The raw corpus used in the development of the parser is partially same as compared to the Urdu parser (Section 4). A comparative study made is detailed in Section 6.

3 Preliminary work

This section describes the preliminary work needed to parse the Urdu sentences through the parser discussed in Section 4. Section 3.1 describes the URDU.KON-TB treebank briefly to give an idea of its annotation scheme, which has a rich morphological, POS, syntactical and functional information. A CFG is then extracted from this treebank presented in Section 3.2.

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

https://doi.org/10.1017/S1351324915000133 Published online by Cambridge University Press

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)

Fig. 2. A verb V example from the URDU.KON-TB treebank.

3.1 The URDU.KON-TB treebank description

A treebank having the PS and the hyper dependency structure annotation with rich information is described in Abbas (2012). This URDU.KON-TB treebank is used for the training of the Urdu parser discussed in Section 4. The URDU.KON-TB treebank has a semi-semantic POS (SSP) tag set, a semi-semantic syntactic (SSS) tag set and a functional (F) tag set. Morphological information in the labeling of the parser's lexicon is contained in the SSP tag set of the URDU.KON-TB treebank. A detailed discussion can be seen in Abbas (2012), however, a short overview is presented here for ease of comprehension.

The SSP tag set hierarchy has twenty-two main tag categories which are divided into sub-categories based on morphology and semantics. In Figure 2, an example of only a verb V is given. A dot '.' symbol is used for the representation of morphology and semantics at the POS level. In Figure 2, the hierarchy of tag labels for the 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 eleven 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 Ramchand 2001; Butt and Rizvi 2010)), V.INF (V has an infinitive form Butt (1993); Abbas and Nabi Khan (2009); Abbas *et al.* (2015)), etc. The third level contains further twenty-five 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 2003, 2010)), etc. These types of combinations are also possible in case of an auxiliary verb as described in Abbas (2012). This short discussion is about the idea of morphological and verb-functional information encoded at the POS level. This lexical information can be passed up to the syntactical level because the lexical items have some relationship with the other

https://doi.org/10.1017/S1351324915000133 Published online by Cambridge University Press

NP (Noun Phrase)	-PLINK (Predicate Link)
.ACC (Accusative)	-SUB (Subject)
-OBJ (Direct Object)	-SUB (Subject)
-SUB (Subject)	-TMP (Temporal)
.DAT (Dative)	-OBJ (Direct Object)
-OBJ2 (Indirect Object)	-OBL (Oblique)
-SUB (Subject)	-PLINK (Predicate Link)
-MNR (Manner)	-SUB (Subject)
.NOM (Nominative)	.POSS (Possessive)
-MNR (Manner)	-SUB (Subject)
-OBJ (Direct Object)	-MNR (Manner)
-SUB (Subject)	-MODF (Modifier/Adjunct)
-MODF (Modifier/Adjunct)	-MODF (Modifier/Adjunct)
-OBJ (Direct Object)	-SPT (Spatial)
-OBJ2 (Indirect Object)	-MODF (Modifier/Adjunct)
-OBL (Oblique)	-OBJ (Direct Object)
-PLINK (Predicate Link)	-OBL (Oblique)
-SPT (Spatial)	-TMP (Temporal)
-OBJ (Direct Object)	-MODF (Modifier/Adjunct)

Fig. 3. A noun phrase (NP) example from the URDU.KON-TB treebank.

lexical items in a sentence. We will explain it after presenting a short overview of the SSS and F tag set hierarchy.

The SSS and F tag sets contain twenty-six and eighteen tag categories, respectively. The SSS tag set hierarchy is divided into three more levels known as a hierarchical design. An example of a simple noun phrase (NP) is depicted in Figure 3. This is taken from the SSS tag set discussed in Abbas (2012). A NP is divided into four levels of depth, as in the case of V. The first level is divided into nine subcategories e.g. NP.ACC (accusative case of NP), NP.DAT (dative case of NP), NP-MNR (NP having a manner concept in it), NP.NOM (nominative case of NP), etc. The features are separated by dot '.' and dash '-'. In SSS annotation, '.' is used for representing syntactical features while the '-' is used for annotating semantic/functional features. The nine subcategories at the second level of NP annotation are further divided into nineteen subcategories e.g. NP.ACC-OBJ (accusative case of NP acting as a direct object of a sentence), NP.ACC-SUB (accusative case of NP acting as a subject in a sentence), NP.DAT-OBJ2 (dative case of NP acting as an indirect object in a sentence), etc. At the last hierarchical level of the NP, NP.NOM-MNR, NP.NOM-SPT and NP.NOM-TMP are divided into two, three and four subcategories e.g. NP.NOM-MNR-OBJ (NP with nominative case having a concept of manner is acting as a direct object of the sentence), NP.NOM-SPT-OBJ (NP with nominative case having a spatial sense acting as a direct object in a sentence), NP.NOM-TMP-PLINK (NP with nominative case having a temporal sense acting as a predicate link of a sentence), etc.

(1) (a) میں اسے جنگل سے گزاروں گا

mEN *usE* *jangal* *sE* *guzArUN* *gA*
P.PERS P.PERS N.SPT CM V.SUBTV VAUX.FUTR
'I will pass him through the forest'

https://doi.org/10.1017/S1351324915000133 Published online by Cambridge University Press

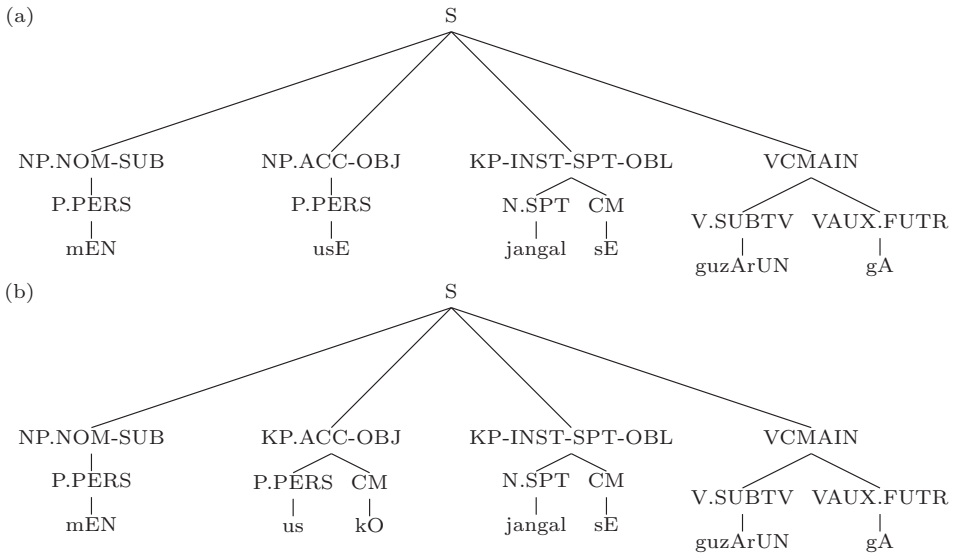


Fig. 4. Passing up lexical information to the syntactical level.

(b) میں اس کو جنگل سے گزاروں گا

mEN	*us*	*kO*	*jangal*	*sE*	*guzArUN*	*gA*
P.PERS	P.PERS	CM	N.SPT	CM	V.SUBTV	VAUX.FUTR

'I will pass him through the forest'

After presenting the SSP and SSS, we can now understand the passing of lexical information up to the syntactical level. Two sentences with respective trees are given in Example 1 and Figure 4. Between these two sentences, the difference is only the presence of case marker *kO*. The passing up of lexical information at the POS to the syntactic level can be seen in the two different respective trees. In case of object marking as in Figure 4(a), the case marker (CM) *kO* with a personal pronoun (P.PERS) *us* 'him' is physically absent and the phrase contains only an inflectional form of personal pronoun as *usE* 'him' concluding a NP. This inflectional form has a hidden sense of CM *kO*, identifying an accusative case of direct object as NP.ACC-OBJ. On the other hand, the Figure 4(b) contains a CM preceded by a P.PERS. The physical existence of CM concludes a case phrase (KP) rather than a NP and disambiguating the trees as well. In the case of oblique OBL argument, the phrase *jangal*/*N.SPT sE*/*CM* 'through/from the forest' is annotated with KP due to instrumental case marker CM in both the trees. The instrumental INST semantic property of the CM is added up to the KP and similarly the word *jangal* 'forest' is a spatial noun whose spatial SPT semantic property is passed up to KP in order, which is acting as an oblique argument OBL in the sentence. The purpose of such information at morphological, POS, syntactical and functional level of annotation is not only to disambiguate the parse trees but it is essential to parse MRLs as concluded by Tsarfaty *et al.* (2013).

https://doi.org/10.1017/S1351324915000133 Published online by Cambridge University Press

3.2 Grammar extraction

The URDU.KON-TB treebank is a manually annotated set of 1,400 parsed sentences, which is recorded in a text file on a computer in the form of 1,400 bracketed sentences. Initial twenty bracketed sentences from each 100 are separated in another text file concluding 280 sentences, which are then used for the development of a test suite. The test suite is 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 evaluation of results after the completion of the Urdu parser. From the first residual text file with 1,120 bracketed sentences, a CFG was extracted using a stack based extraction module given in Algorithm 1. The text file contained bracketed sentences (displayed in Figure 5) separated with a '$' sign and is given as an input file to the algorithm. The algorithm used a *Stack* and a *StrArray* to process the productions between the left and the right parenthesis e.g. '(' and ')' of bracketed sentences. For example, If there is a production like (NP (N.SPT gHar)), then the algorithm pushes all the strings one-by-one on to the *Stack* as (NP (N.SPT gHar until a first right parenthesis ')' after a token *gHar* 'house' is introduced. Without storing ')' on the *Stack*, already pushed strings are popped back until a left parenthesis '(' comes before the POS tag N.SPT and is stored respectively in an array *StrArray* in reverse order as gHar N.SPT without '('. At this point, a production/rule extraction has been completed in *StrArray*. The last element N.SPT of *StrArray* is copied back at the top of the *Stack* for further usage, because it is the part of the next production to be extracted. The *StrArray* elements are recorded in an output file in its original order using lines twenty-four to thirty-three of the algorithm concluding a production as N.SPT → gHar. After making the *StrArray* empty, reading of the strings from the input file remains continued. As the ')' appears, the popping back process from the *Stack* starts and the new production arrives in the *StrArray* as N.SPT NP, which is then re-ordered as NP → N.SPT. This whole process continues until the end of file and finally an output file with a complete CFG productions of 1,120 bracketed sentences is obtained. The extracted CFG is then processed by the Urdu parser to obtain a CFG database having unique productions from it. During this process, the labeling as L and NL at the end of each production is done. The productions having only lexical items at their right-hand side are labeled as L and the productions which have non-lexical items on their right-hand side are labeled as NL. The purpose of this labeling is to provide an already processed mechanism through which the parsing algorithm can identify a production as L type or NL type speedily without checking it thoroughly. This identification also helps the `PREDICTOR()`, `COMPLETER()` or `SCANNER()` of the Urdu parser to process further. A CFG is depicted in Figure 6 for the example sentence in 2.

(2)

vO	*xUS*	*hOtE*	*tHE*	*aOr*	*ina2Am*
he.Masc.Pl=Nom	happy	be.Cop.Imperf.Masc.Pl	be.Past.Masc.Pl	and	reward
dEtE	*tHE*	.			
give.Imperf.Masc.Pl	be.Past.Masc.Pl	full-stop			

'They had been happy and gave rewards.'

https://doi.org/10.1017/S1351324915000133 Published online by Cambridge University Press

Algorithm 1 A CFG extraction algorithm

Input: A input and an empty output file

```
(Sentence, Top, Counter) ← 0
Read : InputString                                   ▷ Read a string from an input file
while InputString ≠ Input.EOF() do                   ▷ Loop until end of file
    if InputString = $ then
        Print : ++Sentence                           ▷ Displaying sentence number on the screen
        Read : InputString                           ▷ Read a string from an input file
        Write : \n \n                                ▷ Writing two newlines in output file
        (Stack[0], StrArray[0]) ← ∅                  ▷ Initializing stack and array
        (Top, Counter) ← 0                           ▷ Initializing stack and array variables
    end if
    if InputString ≠ ")" then
        Stack[Top] ← InputString; Top++              ▷ Filling stack with strings until ')' comes
    else                                             ▷ When ')' comes
        Top − −
        while Stack[Top] ≠ "(" do                    ▷ Loop for popping back strings until '('
            StrArray[Counter] = Stack[Top]           ▷ Storing a rule within '(' and ')'
            Stack[Top] = ∅; Counter++; Top − −
        end while
        Counter − −
        Stack[Top] = StrArray[Counter]               ▷ Storing rule LHS for further process
        Top++ and Check = Counter
        while Counter ≥ 0 do                         ▷ Loop for writing CFG productions in output file
            if Counter = Check then
                Write : StrArray[Counter] →; StrArray[Counter] = ∅
            else
                Write : StrArray[Counter] + ""; StrArray[Counter] = ∅
            end if
            Counter − −
        end while
        Write : \n; Counter = 0                      ▷ In output file
    end if
    Read : InputString                               ▷ Read a string from an input file
end while
```

Output: An output file having complete CFG productions for each sentence

4 Urdu parser

Without handling the issues discussed in Section 4.1, the Earley's algorithm was not able to provide state-of-the-art evaluation results for Urdu. These issues caused the parsing discontinuities or the parsing failures (the parser failed to parse next during processing of a grammar production), due to which modifications or extensions were applied on the basic algorithm to remove these parsing anomalies. The extended

https://doi.org/10.1017/S1351324915000133 Published online by Cambridge University Press

(2)

vO	*xUS*	*hOtE*	*tHE*	*aOr*
he.Masc.Pl=Nom	happy	be.Cop.Imperf.Masc.Pl	be.Past.Masc.Pl	and
ina2Am	*dEtE*	*tHE*	.	
reward	give.Imperf.Masc.Pl	be.Past.Masc.Pl	full-stop	

'They had been happy and gave rewards.'

```
( S
   ( NP.NOM-SUB-1
      ( P.PERS vO ) )
   ( ADJP-PLINK
      ( ADJ xUS ) )
   ( VCMAIN
      ( V.COP.IMPERF hOtE ) ( VAUX.PAST tHE ) )
   ( C.CORD aOr )
   ( NP.NOM-SUB-L-1  * )
   ( NP.NOM-OBJ
      ( N  ina2Am ) )
   ( VCMAIN
      ( V.IMPERF dEtE ) ( VAUX.PAST tHE ) )
   ( M.S    ) )
```

Fig. 5. Bracketed annotation of Example 2.

```
P.PERS → vOh L
NP.NOM-SUB-1 → P.PERS NL
ADJ → xUS L
ADJP-PLINK → ADJ NL
V.COP.IMPERF → hOtE L
VAUX.PAST → tHE L
VCMAIN → V.COP.IMPERF VAUX.PAST NL
C.CORD → aOr L
NP.NOM-SUB-L-1 → * L
N → ina2Am L
NP.NOM-OBJ → N NL
V.IMPERF → dEtE L
VAUX.PAST → tHE L
VCMAIN → V.IMPERF VAUX.PAST NL
M.S → . L
S → NP.NOM-SUB-1 ADJP-PLINK VCMAIN C.CORD NP.NOM-SUB-L-1 NP.NOM-OBJ
     VCMAIN M.S NL
```

Fig. 6. A CFG of the example sentence in 2.

version is depicted in Algorithm 2. In this algorithm, the extracted CFG given as a database has three fields/columns in the form of LHS (Left-Hand Side of production), RHS (Right-Hand Side of production) and Type (Type of production either L or NL), which can be inferred through matching with the CFG in Figure 6. After taking a sentence as input, various variables are initialized along with a `chart[]` array, which is an array of objects. The starting value of the chart is given as `ROOT @ S`. In place of bullet/dot symbol '•' used in the Earley algorithm, here an '@' symbol is used because the bullet/dot symbol is extensively used in the hierarchical 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 modifications in the `PREDICTOR()`, `COMPLETER()` and a `SCANNER()` along with the additional functions i.e. an `EDITOR()` for an automatic

https://doi.org/10.1017/S1351324915000133 Published online by Cambridge University Press

editing of discontinuous parses and a `BUILDER()` for building parse trees presented at the end of the algorithm.

Algorithm 2 Urdu Parser

```
1: function URDU-PARSER(grammar)
2:     Input: Sentence                                          ▷ reading a sentence
3:     (id, fi, fj, fid) ← 0          ▷ initializing StateID, failed i, failed j, and failed StateID
4:     chart[0].add("id", "ROOT @ S", "0,0", " ", "Seed")       ▷ initializing chart
5:     for i ← 0 to LENGTH(sentence[]) do    ▷ Loop for each word in a sentence
6:         scannerFlag ← false, id ← 1
7:         Print: chart[i] → (StateId, Rule, @Position, BackPointer, Operation)
8:         for j ← 0 to ChartSize[i] do                         ▷ Loop for chart entries
9:             currentRule ← chart[i].getRule(j).split(" ")     ▷ splitting rule with space
10:            (tempString, index) ←(string-after-@, @Position) in currentRule
11:            if tempString = " " then                 ▷ Completer Case if NULL string
12:                call COMPLETER()                     ▷ calling completer procedure
13:            else
14:                rs ← All grammar rules with LHS = tempString
15:                if rs.next() ≠ 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 ≠LENGTH(sentence[]) then
22:                call EDITOR()
23:            end if
24:        end for
25:    end for
26:    call BUILDER()
27: end function
```

Before the execution of the internal loop, the `scannerFlag` is set to false (which is dealt within the scanner Algorithm 4) and an initial state of the chart is printed. Within the internal loop, the productions inserted in the respective charts are picked up one-by-one and are split with space to find the next processing non-terminal in `tempString` and its location `index`. If the `tempString` is found to be empty then the `COMPLETER()` will execute to complete the status of the relevant production by moving forward the '@' symbol, which is discussed in Section 4.1.7 along with its Algorithm 9. Otherwise, there are two possibilities, which are decided on the basis of further productions extracted from the grammar database into a record set `rs`. The first possibility of `PREDICTOR()` will execute if the value of `rs` is true, which means that it will contain further extracted productions and then the `PREDICTOR()` will add predicted productions into the current chart, whose further detail is presented in Section 4.1.1 with the Algorithm 3. Similarly, the second possibility of `SCANNER()` will execute if `rs` is found to be empty and then `SCANNER()` will create a new chart

https://doi.org/10.1017/S1351324915000133 Published online by Cambridge University Press

and it will then add a relevant L type production into this chart from a previous chart as illustrated in Section 4.1.2 along with its Algorithm 4.

During processing of COMPLETER(), PREDICTOR() and SCANNER(), parsing discontinuities can happen. To check these types of phenomena, an EDITOR() will come into action and it will remove all previous irrelevant entries and charts causing discontinuity up to an optimal point of parsing discovered earlier. The detailed discussion is presented in Section 4.1.6 along with its Algorithm 8. After the end of the external loop, the parsed solutions are generated and stored in the form of 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 is executed. This constructs parsed trees of solutions by manipulating the back-pointers calculated in COMPLETER() function. The BUILDER() is able to display all parsed solutions of a given sentence as discussed in Section 4.1.4 along with its relevant Algorithm 6, then the Algorithm 2 for the Urdu parser[6] is exited with complete generation of charts and bracketed parsed trees. The discussion of this main Algorithm 2 is done in this section, while the algorithms of functions called by the main algorithm are discussed in detail in the respective subsections of Section 4.1.

4.1 Extended functions

Functions called by the main Algorithm 2 are extended/introduced to meet the parsing requirement of MRL Urdu. These functions mainly include the PREDICTOR(), COMPLETER(),SCANNER(), EDITOR(), BUILDER()— and the BACKPOINTER(). The discussion of each function algorithm along with the issues is presented as follows.

4.1.1 Eliminating useless predictions

Earley's Predictor() adds useless productions in charts which cause a wastage of time during processing, and increase the chance of misleading direction toward a wrong solution or discontinuous parse if these useless productions are selected by the Earley's Completer() or Earley's Scanner() for further processing. Suppose, the current token to be parsed in an input sentence is خان 'Khan', which is a proper noun and there is a NL type production NP → @ N.PROP N.PROP residing in the current chart of the parser. Here, NP is the noun phrase, N.PROP is the proper noun and the '@' symbol is the equivalent of the bullet/dot • symbol of the Earley algorithm. 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 along with the relevant production N.PROP → @ خان from the grammar. Even the addition of irrelevant

[6] Technologies used for the development of the Urdu parser includes OpenJDK-7 (Open source Java Development Kit 7) available at http://openjdk.java.net/projects/jdk7/, Apache Tomcat V 7.0 available at http://tomcat.apache.org/index.html and MySQL available at http://dev.mysql.com/ using Eclipse IDE (Integrated Development Environment) available at https://www.eclipse.org/downloads/. The algorithms presented in Sections 4 and 4.1 provides a taste of source code written in Java environment.

https://doi.org/10.1017/S1351324915000133 Published online by Cambridge University Press

productions are not required, only the relevant production N.PROP → @ 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 a wastage of time and increase the chance of misleading direction toward a wrong solution or a discontinuous parse/state. To resolve this issue, the `PREDICTOR()` is extended to abandon the selection and insertion of useless productions from the grammar into the charts as given in Algorithm 3 and its execution is discussed next.

Algorithm 3 Predictor

```
1: function PREDICTOR(rs, i, id, chart, chartSize)
2:     if rs.getString(3) = "NL" then                ▷ Checking non-lexical production
3:         for k ←0 to chartSize[i]-1 do
4:             cRule ← chart[i].getRule(k).split(" ")
5:             if rs.getString(1) = cRule[0] & cRule[1] ="@" then
6:                 break                              ▷ production exists in chart already
7:             end if
8:         end for
9:         if k = chartSize[i] then          ▷ True, if production does not exist in chart
10:            repeat
11:                chart[i].add(id, rs.getString(1)+" @ "+rs.getString(2), i+","+i, " ", "Predictor")
12:                chart[i].print(id, Rule(id), @Position(id), BackPointer(id), Operation(id))
13:                chartSize[i] ← id = id + 1
14:            until !rs.next()
15:            rs.close()                          ▷ closing Result Set handle of grammar
16:        end if
17:    else                                          ▷ Case of lexical (L) productions
18:        for k ←0 to chartSize[i]-1 do
19:            cRule ← chart[i].getRule(k).split(" ")
20:            if rs.getString(1) = cRule[0] & cRule[1] ="@" then
21:                break                              ▷ production exists in chart already
22:            end if
23:        end for
24:        if k = chartSize[i] then          ▷ True, if production does not exist in chart
25:            repeat
26:                if rs.getString(2) = Sentence[i] then
27:                    chart[i].add(id, rs.getString(1)+" @ "+rs.getString(2), i+","+i, " ", "Predictor")
28:                    chart[i].print(id, Rule(id), @Position(id), BackPointer(id), Operation(id));
29:                    chartSize[i] ← id = id + 1
30:                end if
31:            until !rs.next()
32:            rs.close()                          ▷ closing Result Set handle of grammar
33:        end if
34:    end if
35: end function
```

https://doi.org/10.1017/S1351324915000133 Published online by Cambridge University Press

When the main parsing Algorithm 2 calls the `PREDICTOR()`, then it checks the type of production first either as NL or L using an external `if` condition. For both types, it finds the existence of predicted production in current chart `chart[i]` using a `for` loop controlled by the size of the current chart `chartSize[i]`. This loop from line three to eight is for NL type of productions and from line eighteen to twenty-three is for L type of productions. The loop will break if the predicted production is located in `chart[i]` and then it will be the case when there is no need to add this production into the chart. However, if it becomes unable to locate the predicted production then the value of `k` becomes equal to `chartSize[i]` and if it happens, then a `repeat-until` loop is used to add all extracted predicted-productions into the `chart[i]`. In Earley's algorithm, this process is same for both NL and L types productions, which causes the addition of irrelevant productions. To make it more accurate, the `PREDICTOR()` is divided into two halves to deal with NL and L type productions separately. In dealing with L type of productions, the `PREDICTOR()` is introduced with another condition displayed at line twenty-six of Algorithm 3, which enforces the predictor to add only relevant productions into the chart. It matches the token at RHS of predicted-production `rs.getString(2)` with the current token `Sentence[i]` in a given input sentence. This condition forces the `PREDICTOR()` to introduce only the relevant productions in charts and eliminates the limited possibility of misleading direction toward a wrong solution or a discontinuous parse. After removal of irrelevant productions, the wastage-time factor is reduced to optimal linear-time $O(n)$, which at least is required for processing of n number of tokens in a given input sentence.

4.1.2 Extended scanner

In Earley's parsing algorithm, the `scanner()` only matches the RHS of the L type production with the current token in a given sentence and may cause a wrong selection with an incorrect POS tag at the LHS of this L type production. 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 → ہے. It can behave as a copula verb e.g. V.COP.PRES → ہے and it can also act as a main verb e.g. V.PRES → ہے. This concept of having more than one tag is true for other lexical items as well. So, if this L type production VAUX.PRES → @ ہے is the right candidate in the current chart 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/failed state in the chart during parsing. To remove this issue, the `scanner()` is extended in the same way as was done with the `PREDICTOR()` in Section 4. This solution presented in Algorithm 4 solves the issue partially at this level, but it is completed in Section 4.1.6.

When the `SCANNER()` is called, it extracts all relevant L type productions from grammar after matching their LHS and RHS with `currentRule[index - 1]` and `tempString` of current L type production in a chart, respectively. It adds the correct L type production in a new chart `chart[i + 1]` after checking three conditions. At first, it checks that the chart number is not exceeding the length of the sentence, as the total number of charts to be produced should be equal to `LENGTH(Sentence)+1`.

https://doi.org/10.1017/S1351324915000133 Published online by Cambridge University Press

Algorithm 4 Scanner

```
1: function SCANNER(rs, i, id, tempString, Sentence, scannerFlag, fi, fj, fid, chart, chartSize)
2:     rs ← All grammar productions with LHS = currentRule[index − 1] & RHS = tempString
3:     if rs.next() ≠ false then                          ▷ Record set rs is not empty
4:         if i+1 ≠ LENGTH(Sentence)+1 then
5:             if tempString = Sentence[i] then
6:                 if !scannerFlag then
7:                     chart[i + 1] ← add("0", rs.getString(1)+" "+rs.getString(2)+
                                " @",i+","+(i+1), " ", "Scanner" )
8:                     chartSize[i + 1] = 1                ▷ new chart is initialized
9:                     scannerFlag = true
10:                    rs.close()
11:                    fi = i, fj = j, fid = id
12:                else
13:                    rs.close()
14:                end if
15:            else
16:                rs.close()
17:            end if
18:        else
19:            Print: Input Sentence ends at i
20:        end if
21:    else
22:        Print: Record set rs is empty
23:    end if
24: end function
```

Secondly, it checks that the token `tempString` in the current processing L type production is equal to the current token `Sentence[i]` in a given sentence. After that if the `scannerFlag` is false, then the new entry of matched L type production is added into the new chart. During this process, the `scannerFlag` is set to true value along with a record of some variables `fi, fj, fid`, which will be used in `EDITOR()` presented in Algorithm 8. By introducing this modification, the possibility of wrong selection of production from the grammar is abandoned and the exactly matched L type production extracted from the grammar is added to the next chart accordingly. An issue left in this solution is that the Earley's `PREDICTOR` can add all the L type productions mentioned in Section 4 into the current chart for processing of `scanner()`. In this situation, the `scanner()` will not know which one is the true candidate production for parsing. This issue is addressed and resolved in Section 4.1.6.

4.1.3 Back-pointers calculation

Earley parsing algorithm is a generator or a recognizer and hence, cannot produce parse trees or bracketed trees. The output of the Earley algorithm is the N+1

https://doi.org/10.1017/S1351324915000133 Published online by Cambridge University Press

number of charts where N is the number of words in a given sentence. To produce parse trees or bracketed trees from the output of the Earley algorithm, an idea of back-pointers was shortly discussed by Earley (1968) in his doctoral thesis. The same idea is implemented in detail to produce parse trees in our Urdu parser. To understand the calculation of back-pointers, a sentence given in Example 3 is parsed from the Urdu parser. The charts generated through it are depicted in Figure 7. Only relevant states are displayed as can be inferred from the non-sequential values of STATEID column. The column DOT-POSITION is basically the position of '@' in productions.

(3) 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()` Algorithm 9 calls the `BACKPOINTER()` Algorithm 5 to calculate the values of back-pointers. For example, during processing of `STATEID` 3 in chart 1 of Figure 7, the `COMPLETER()` calls the `BACKPOINTER()` with string type arguments `previousRule` and `dummy@Position` as K.P.POSS→P.PERS @ CM and '0,1', respectively. The final indexed value of '@' in variable `tempIndex` after subtraction obtained through the `previousRule` is 1. The string value P.PERS at this index is then stored in a variable `NT`. The value of `k` becomes 0 at this stage in the algorithm. The external loop is used to traverse the charts one-by-one from the recent state of the chart. After checking the condition of `tempIndex` at line eight, another loop for traversal of chart entries is executed. At this point, a production P.PERS→ان @ at state 0 of chart 1 is stored finally in the `cRule`. The index of '@' here in this production is stored in the `tIndex` as 2. A condition presented next becomes true and the value of the `backPointer` is calculated as '1-0'. The value in the DOT-POSITION column of this state is split with ',' and stored in a temporary variable `dummy@P`. Before breaking of internal loop, the values in `l, tempIndex` and `NT` are updated with 1, 0 and K.P.POSS, respectively. Then the control goes to the external loop in continuity with `l=1`, but now the condition of `tempIndex` becomes false and finally the external loop breaks. The calculated '1-0' value of `backPointer` is then displayed by the `COMPLETER()` in the relevant state of the chart. The rest of the back-pointers displayed in Figure 7 are calculated in the same way. These back-pointers are further used in building of bracketed parse-trees which will be discussed in Algorithm 6.

4.1.4 Building bracketed parse trees

All the back-pointers are calculated and stored in each relevant state of the chart as presented in Section 4.1.3. When all the states in the respective charts are evaluated and the external loop in Algorithm 2 is exited then another function `BUILDER()`

https://doi.org/10.1017/S1351324915000133 Published online by Cambridge University Press

Algorithm 5 Back Pointer

```
1: function BACKPOINTER(previousRule, dummy@Position, i, chartSize, chart)
2:     backPointer ← ""
3:     tempIndex ← previousRule.indexOf("@")          ▷ locating index of '@'
4:     tempIndex ← tempIndex-1                          ▷ subtracting index
5:     NT ← previousRule.get(tempIndex)          ▷ saving non-terminal before '@'
6:     k ← dummy@Position[0]          ▷ initial '@' position of previous relevant chart rule
7:     for l ← i to k step -1 do                  ▷ loop for backward backpointers
8:         if tempIndex > 0 then
9:             for m ← 0 to chartSize[l]-1 do          ▷ loop for locating relevant chart entries
10:                pString ← chart[l].getRule(m).split(" ")
11:                cRule.add(pString[])                    ▷ store pString in cRule
12:                tIndex ← cRule.indexOf("@")              ▷ getting index of '@'
13:                if (NT = cRule[0]) & (tIndex+1 = SIZE(cRule)) then
                       ▷ matching current non-terminal with non-terminals in relevant rules
14:                    backPointer ← (l + "-" + chart[l].getStateId(m) +"
                           "+backPointer)                  ▷ calculating backpointer
15:                    dummy@P = chart[l].get@Position(m).split(",")          ▷ getting '@' position
16:                    l ← dummy@P[0]                      ▷ updating loop counter l
17:                    l ← l + 1
18:                    tempIndex ← tempIndex-1
19:                    NT ← 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
```

depicted in Algorithm 6 is called. After displaying all chart entries as shown in Figure 7, the possible bracketed parse trees are evaluated and displayed by the `BUILDER()` function. Both the `BUILDER()` and the `BACKPOINTER()` contributes to shift our Algorithm 2 from a generator to a parser in contrast of the Earley's algorithm.

After displaying the chart entries in Figure 7 for the sentence given in Example 3, the `BUILDER()` sets the `num` and `chartN` to zero and the length of sentence, respectively. A loop is used to calculate the location of all `S` solution productions in the last chart and stored them in a string list `bp[]`. After evaluation, it has only

https://doi.org/10.1017/S1351324915000133 Published online by Cambridge University Press

Algorithm 6 Builder

```
1: function BUILDER(Sentence[], chartSize[], chart[])
2:     num=0, chartN = LENGTH(Sentence[])
3:     for count ← chartSize[LENGTH(Sentence[]]-1 to 0 step -1 do
4:         dummystr ←"S" and rule ← chart[chartN].getRule(count).split(" ")
5:         if rule[0] = dummystr then
6:             num = num + 1
7:             bp.add(chartN+"-"+chart[chartN].getStateId(count))   ▷ collecting relevant back-pointers
8:         end if
9:     end for
10:    tree[] ← new BTree[num]   ▷ initialized num number of bracketed-tree type tree[] objects
11:    for i ← 0 to SIZE(bp)-1 do
12:        tree[i].build(bp.get(i), chart)   ▷ building tree with pointers
13:    end for
14:    for i ←0 to SIZE(bp)-1 do
15:        tree[i].prepare(chart)   ▷ preparing bracketed tree for presentation
16:    end for
17:    for i ←0 to SIZE(bp)-1 do   ▷ loop for displaying all parsed trees
18:        bracketedSentenceLength ← tree[i].getSize() and left ← 0
19:        if bracketedSentenceLength > 0 then
20:            Print : Bracketed Parse Tree "+(i+1)+" of "+SIZE(bp)+"
21:            for j ←0 to bracketedSentenceLength-1 do
22:                if tree[i].getString(j) = "(" then
23:                    left = left + 1 and Print :  newline
24:                    for tab ←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
```

one solution production with '8-1' string value in `bp[]` for the example sentence. A bracketed-tree class type array of objects `tree[]` is initialized with the number of solutions found at line ten of the algorithm. This `tree` object is then called a user-defined method named `build(bp.get(i), chart)` with two arguments in

https://doi.org/10.1017/S1351324915000133 Published online by Cambridge University Press

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

Fig. 7. A back-pointer calculation example of the Urdu parser.

another loop. This method builds an unformatted intermediate bracketed-parse-tree with interlinked back-pointers from chart states and reveals the leaf nodes only as (8-1 (4-1 (2-2 (ان 1-0) (کا 2-0) (ذکر 3-0) ((بھی 4-0) ((یہاں 5-0) 5-1) ((ضروری 6-0) 6-1) ((ہے 7-0) 7-1) ((. 8-0))). This intermediate parse tree can be understood well by looking at the given back-pointers in the respective chart states.

A third independent loop starts and prepares the intermediate parse tree into a complete unformatted parse tree using a statement given as `tree[i].prepare(chart)`. When this loop is exited then the parse tree has obtained a form 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-PLINK (ADJ.MNR ضروری)) (VCMAIN (V.COP.PRES ہے)) (M.S .)). So, this `prepare()`

https://doi.org/10.1017/S1351324915000133 Published online by Cambridge University Press

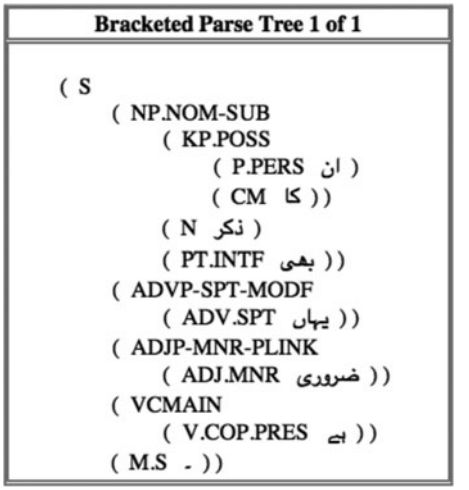


Fig. 8. An output of the *BUILDER*() method.

method only replaces the back-pointer with the LHS of relevant productions. At this point, the only thing left is the formatting of this parse tree, which is done from line seventeen to thirty-five in Algorithm 6 and the parse tree gets a final form as depicted in Figure 8.

4.1.5 Empty productions

Empty productions are divided into two categories. The first one is related to diacritic productions e.g. DIA → * and the second one is related to non-diacritic productions e.g. P.PERS → *. The POS tags DIA and P.PERS are used to represent diacritics and personal pronouns, respectively, where the '*' is used to represent absence of these categories. The diacritic is a mark or a sign that is placed over, under, or through a letter of a word in some languages to describe that the letter of a word should be pronounced in a particular way or the word with a diacritic at the last letter has some sort of connection to the following word like compound words in Urdu which will be discussed next. As tags to represent these empty productions exist in our grammar, there is a need to deal with it. It can cause discontinuity during parsing because the lexical item may or may not be present for both the categories in a given sentence. A solution was proposed for compilers by Aho *et al.* (2007), Appel and Palsberg (2007) and Leblanc and Fischer (1988). A similar solution was coded for the Earley algorithm by Aycock and Horspool (2002), which is adopted. The solution is to process the empty productions implicitly in the algorithm by moving the '@' symbol ahead. This solution causes frequent discontinuity during parsing and its property of self-decision at irrelevant places makes the things more worse. It has been found that it does not work well for Urdu.

Usually, diacritics are not present in modern Urdu writing, but the older literature makes extensive use of diacritics e.g. a compound word آبِ حَیَات *AbE h2ayAt* 'The water of life' with a diacritic at the last letter of the first word *Ab* 'water' making it a possessive entity of the following word *h2ayAt* 'life', a single word تقرِیباً *taqrIban* 'almost' with a diacritic at the end of the last letter allowing this word to be

https://doi.org/10.1017/S1351324915000133 Published online by Cambridge University Press

pronounced as *taqrIba-n* with 'n' sound and not as *taqrIba*. The lexicon can be viewed on the website designed by the Center for Language Engineering[7] under the rights protected by the Ministry of Information and Technology, Pakistan.[8] The diacritic used in the first example under the last letter بِ *b* of the first word is called *zErE-Iz3Afat* (a diacritic for addition), which is connecting the first word with the following word and thus making a compound word. This phenomenon is still in use in modern Urdu writing. The diacritic above the last letter اً *a* in second example is called *tanwin* (a diacritic for final post-nasalization), which is different from the first example and related to pronunciation of the letter. There are also other diacritics in use as well e.g., *zEr, zabar, pEsh, taSdId*, etc. In short, the problem of the modern Urdu text is this that diacritics may arise in the text or not.

During parsing, a compound word can come with the diacritic *zErE-Iz3Afat* in a given sentence 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 *E* is absent at the end of the first word *Sehr*. In such cases, its presence is by default understood by the native speakers. The production exists in the grammar to handle this compound word is NP-SPT → @ N.SPT DIA N.PROP.SPT. When the first word *Sehr*/N.SPT is processed by the `SCANNER()` then the '@' is moved ahead and the production becomes NP-SPT → N.SPT @ DIA N.PROP.SPT. The DIA is the candidate to be executed next to represent the absence of diacritic at the end of first word *Sehr*. 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 NP-SPT → N.SPT DIA N.PROP.SPT @. The problem with this solution by Aycock and Horspool (2002) is that it performs the implicit transaction silently for both the compound and non-compound words as observed and is unable to differentiate between these type of words. For example, if there is a case of compound word as discussed then the solution is perfect, but in the case of non-compound words if two independent words گھر *gHar* 'The house' and مکہ *makkah* 'Makkah' appear in a same position like discussed in compound words e.g. گھر مکہ میں ہے *gHar makkah mEN hE* 'The house is in Makkah', then this solution cannot identify the context and it applies the transaction in the same way due to the same POS tagging of the *gHar* and the *Sehr*.

The diacritics are the part of words in Urdu. In the URDU.KON-TB treebank annotation, no tag was used for the presence of diacritics. It is a wonder, why the DIA tag is used to represent absence only. After facing a problem discussed earlier, the production of DIA was removed from a sample of the grammar and achieved quite good results. It is a tested proposal that this production of DIA should be removed to achieve good parsing results on this treebank.

The absence of an argument or an optional lexical item in the sentence is the case of non-diacritic empty production. In this case, the discontinuity rate is less than the other discussed earlier. It may be due to the independent constituency structure

[7] http://www.cle.net.pk/oud/
[8] http://www.pakistan.gov.pk/

https://doi.org/10.1017/S1351324915000133 Published online by Cambridge University Press

of this category. For example, if a subject with nominative case is missing in a given sentence then the grammar has productions as NP.NOM-SUB → P.PERS and P.PERS → *. Both productions have the singletons (one terminal/non-terminal) on their RHS. This property of singletons is also existed for the other empty productions in this category. The constituent structure is different from the category of diacritic productions. It is not sure but it may be the singletons on the RHS, which are preventing this category not to show immaturity during the parsing.

Algorithm 7 Empty Productions

```
1: if rs.getString(2) = "*" then
2:     chart[i].add(id, rs.getString(1)+" "+rs.getString(2)+" @", i+","+i, " ", "Predictor"
3: else
4:     chart[i].add(id, rs.getString(1)+" @ "+rs.getString(2), i+","+i, " ", "Predictor"
5: end if
```

Due to high frequency of the DIA production in the URDU.KON-TB treebank, the proposed solution of Aycock and Horspool (2002) was implemented in `PREDICTOR()` by replacing the lines eleven and twenty-seven with the Algorithm 7 but the results found were not promising. So, an explicit method to represent empty productions has been adopted. A '*' is usually typed in the given sentence to represent absence of diacritics, arguments, lexical items, etc. Due to this explicit approach, the Urdu parser is jelling with the grammar without any issue related to empty productions.

4.1.6 Lexical dynamic behavior

The issue is related to the words which are homonyms, homographs, homophones, heteronyms and polysemes. A strict definition is considered to these attributes, that means at least the words have the same spelling. It is considered because the parser matches the tokens with the grammar lexicon or with the current word in a given sentence. The case of homonym words in a strict sense is going to be discussed here and the same concept is applicable on other attributes as well. There was an error in the Earley's algorithm, which has been removed in the Urdu parser. It was observed that when the Earley `Predictor` introduced the right candidate word along with some of its homonyms in the current chart and the Earley `Scanner` started its search for the right candidate word to be processed from the beginning of the current chart, then if homonyms of the right candidate word were present with different POS tags at the earlier positions in the current chart then the `Scanner` picked it up for processing and added it into the next chart after matching with the current token in the given input sentence. As this homonym had a different POS tag as compared to the right candidate word, so consequently the `Completer` and the `Predictor` processed further productions and finally the parser got into a wrong solution or a discontinuous state. This issue is explained with an example as follows.

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

https://doi.org/10.1017/S1351324915000133 Published online by Cambridge University Press

a possessive meaning 'of' in 4. On the other hand, it contains a meaning of 'did' in 5 as a verb.

(4) جولیا کی کتاب

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

'The book of Julia'

(5) 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'

In the grammar of the Urdu parser, this word *kI* 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 or 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 the verb V.PERF will not be completed in the next chart and the parser will go with the wrong solution or the discontinuous state. To resolve this issue, it is needed to delete all the failed charts and continue the parsing process with the other choice. To address this issue, the `Scanner` of the Earley algorithm is modified in the Urdu parser as depicted in Algorithm 4, which records the failed state in variables $fi = i, fj = j$ and $fid = id$. At line twenty-one of Algorithm 2, after checking some conditions, an `EDITOR()` presented in Algorithm 8 is called by the Urdu parser to heal this discontinuous state.

Algorithm 8 Editor

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

https://doi.org/10.1017/S1351324915000133 Published online by Cambridge University Press

When the parser is discontinued at *chart*[*i*+1], then by using recorded variables in `SCANNER()`, the `EDITOR()` first deletes and re-initializes the *chart*[*i*+1]. In a sequence, it further deletes and re-initializes the charts from *i* to *fi*+1. *fi* is basically the chart number, where homonyms are available and *fj* is the location of that wrong choice e.g. CM → کی in a chart. The wrong choice is stored in a *rule* array after splitting by space. A loop is used to traverse all productions of a chart where the homonyms are available. The chart productions are stored temporarily in a *temprule* array. When the RHS of both the arrays are found to be equal then the LHS are checked. If LHS are found to be equal then those productions are skipped and recorded otherwise. 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 moves toward an optimal solution.

4.1.7 Completer's limitations

This issue of wrong solution or a discontinuous/failed parse obtained during parsing is related to subordinate clauses, when the number of subordinate clauses in a sentence is more than one. The issue does not appear often and is related to NL type productions, specially the subordinate clauses denoted by SBAR in an annotated or a parsed sentence. The wrong production caused the parsing discontinuity can lie in any chart but one thing is sure that the right choice of production also exists in the same chart but not selected by the `Completer` due to the order of productions. In the Earley algorithm, after finalizing its choice of production, the `Completer` never comes back to seek the other choice even made a wrong choice in its first attempt. Due to this restriction, it was observed that the parser produced a discontinuous parse in case of the nested subordinate clauses and the discontinuity occurred during the processing of the last chart when the production of subordinate clause SBAR was being manipulated.

(6) *[In laRAyI jHagRON=kI vajah=sE Ap=kO*
[these.Pl fight.Fem dispute.Obl=Poss reason.Fem.Sg=Inst he=Dat
moqa2h nah mil sakA [kEh Ap
opportunity.Masc.Sg not find.Root can.Perf [**that/SBAR** he=Nom
[jEsA dil cAhtA tHA us t2arah
[**as/SBAR** heart want.Imperf.Masc.Sg be.Past.Masc.Sg that like
is3lAh2 farmAEN .]]]
reforms do.Subj full-stop]]]

'Due to these conflicts, he could not find an opportunity that he could do the reforms like that as his heart wanted '

A sentence of twenty-three tokens in 6 with two subordinate clauses highlighted with SBAR is also an evidence of this issue among the other examples. After the parsing of productions in the last (twenty-third) chart for this sentence, the order of the complete productions should be as follows. The '@' at the end of each production represents the complete status of productions.

https://doi.org/10.1017/S1351324915000133 Published online by Cambridge University Press

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

Fig. 9. Incomplete solution of the Urdu paser.

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 without any solution during processing of the last chart with the productions given in Figure 9. This means that the parser did not find a complete solution production 'S' with '@' at its end. As it was the last chart processed for the given sentence and another chart could not be added. So, the parser discontinued finally with the incomplete productions in the last chart without any solution.

In Figure 9, the parser performed well up to the completion of first SBAR → ... @ production. Then the `COMPLETER()` went back in charts to search a related incomplete production with the properties like SBAR on its RHS and the '@' symbol before the SBAR, as @ SBAR. The `COMPLETER()` made a fault in chart 12 in the presence of a wrong choice with the similar properties at higher precedence. It found an incomplete SBAR production as SBAR → C.SBORD NP.NOM-SUB @ **SBAR** ADVP-MNR-MODF NP.NOM-MNR-OBJ VCMAIN M.S. As the '@' symbol was before the SBAR, so the `COMPLETER()` picked it up. After moving the '@' forward, it added the updated production in the last chart as can be seen in Figure 9. The `PREDICTOR()` then checked the '@' before the ADVP-MNR-MODF and the parser went into a wrong direction. It happened because the `COMPLETER()` picked up the wrong production in chart 12, even the right choice S → KP-INST-MODF KP.DAT-SUB NP.NOM-OBJ VCMAIN @ SBAR was also there in the early rows of that chart. To resolve this issue, it is needed to allow the `COMPLETER()` to go back further until a successful parse has been obtained as can be seen in Algorithm 9.

When the Urdu parser called the `COMPLETER()`, it first sets the `completerCheck` flag to false. This is the flag which is used to allow the `COMPLETER()` to back track the right choice among the NL type productions. The `COMPLETER()` has two main loops. The external loop is used to handle the previous charts $chart[pc]$ from $i-1$ to 0. The internal loop is used to traverse the chart states from 0 to $chartSize[pc]-1$. With the execution of internal loop, each candidate chart production $chart[pc]$.getRule(c) is stored first in the *previousRule* array through the *dummyString*. An already processed position of the '@' symbol is recorded in the *dummy@Position*. After getting index of the '@' from the *previousRule*, a condition is checked that the production in the *previousRule* is not a complete production. If it is the case, then after matching the LHS of the current chart production *currentRule*[0] with the non-terminal after the

https://doi.org/10.1017/S1351324915000133 Published online by Cambridge University Press

Algorithm 9 Completer

```
1: function COMPLETER(i, j, chartSize, chart, currentRule, id, Sentence)
2:     completerCheck ← false                          ▷ completerCheck flag set to false
3:     for pc ← i-1 To 0 do                            ▷ loop to handle previous charts
4:         for c ← 0 To chartSize[pc]-1 do             ▷ for traversing chart entries
5:             dummyString ← chart[pc].getRule(c).split(" ")      ▷ splitting with space
6:             dummy@Position ← chart[pc].get@Position(c).split(",")   ▷ splitting with ','
7:             previousRule.add(dummyString[])         ▷ storing dummyString in a list
8:             pIndex ← previousRule.indexOf("@")      ▷ locating '@' position
9:             if pIndex+1 ≠ SIZE(previousRule) then
10:                if currentRule[0] = previousRule.get(pIndex+1) then
11:                    move '@' forward and format previousRule, then store it in pStr
12:                    recursion ← false
13:                    Check pStr in chart[i], if exist then recursion ← true
14:                    if !recursion then
15:                        call BACKPOINTER()
16:                        chart[i].add("id", pStr, k+","+i, backPointer, "Completer" )
17:                        completerCheck ← true
18:                        chart[i].print(StateId, Rule, @Position, BackPointer, Operation)
19:                        id = id + 1                 ▷ increment StateId
20:                        chartSize[i] ← id           ▷ chartSize is updated
21:                        previousRule.clear()
22:                    end if
23:                else
24:                    previousRule.clear()
25:                end if
26:            else
27:                previousRule.clear()
28:            end if
29:        end for
30:        if completerCheck = true & i < LENGTH(Sentence[]) then
31:            break          ▷ pc loop exit because completer() has finished and no back track
32:        else if completerCheck = true & i ≥ LENGTH(Sentence[]) then
33:            completerCheck ← false
34:        end if
35:    end for
36: end function
```

'@' in the previous chart production *previousRule*.get(*pIndex*+1), the '@' symbol is moved forward. The updated and formatted *previousRule* is stored in another array *pStr* for further processing. The existence of this updated production in *chart*[*i*] is checked via a *recursion* Boolean variable. If the *recursion* is found to be false then

https://doi.org/10.1017/S1351324915000133 Published online by Cambridge University Press

back-pointers are calculated first as explained in Section 4.1.3. After calculating the back-pointers, the updated production entry is then added and printed into a *chart*[*i*] by setting the `completerCheck` to true. After updating some variable, the *previousRule* is again cleared and the internal loop continues. However, if the *previousRule* is found to be a complete state, then the array *previousRule* is made empty and the internal loop continues. At the end of internal loop, there are some conditions which deal with the discontinuity issues of the `COMPLETER()` discussed in this section. If the *completerCheck* is found to be true and the chart number is less than the length of the sentence then the external loop breaks, which means that the `COMPLETER()` has found the solution and there is no need to go back. On the other hand, if the *completerCheck* is found to be true and the chart number is greater or equal to the length of the sentence then the `COMPLETER()` is allowed to back track by setting its flag to its default value.

5 A run example

To see the effect of the Urdu parser, we examine Example 7. The length of the sentence is five tokens, for which the parser generates 0 to 5 charts. All irrelevant NL type productions are removed from the output charts to save the document space. Due to this removal, the values in the STATID column of Figure 10 are not consecutive. The charts generated through the parser without the bracketed parse tree are displayed in Figure 10. After displaying these charts, the parser then displays the bracketed parse tree of this sentence, which can be seen in Figure 11.

(7) قبلائی خان منگول تھا ۔

QublAI	*xAn*	*mangOl*	*tHA*	-
N.PROP	N.PROP	N.PROP	V.COP.PAST	

'Kublai Khan was a Mangol.'

After the removal of issues discussed in Section 4.1, the parser has got the following benefits. The output of the parser is so directed, speedy and refined in the sense that no extra or irrelevant L type productions are introduced by the `PREDICTOR()`. It is really hard now that the `SCANNER()` will select a wrong choice of production. If it happens then the `SCANNER()` has a tendency to correct itself. The `COMPLETER()` is boosted up to find a solution if the solution is available in any chart at any position. Empty productions are handled but it is better to remove them from the grammar completely. The calculation of back-pointers in the `COMPLETER()` is helpful to the `BUILDER()`, which builds bracketed parse trees of the given sentence. The `EDITOR()` provides a mechanism through which failed parsed output is edited and corrected. These all features make up an Urdu parser with rich morphological, syntactical and functional information in the form of a grammar in it.

6 Experimentation and results

The URDU.KON-TB treebank contains 1,400 annotated sentences. The sentences are divided into 80% training data (for the development of grammar) and 20% test

https://doi.org/10.1017/S1351324915000133 Published online by Cambridge University Press

CHART(0)				
STATEID	RULE	DOT-POSITON	BACK-POINTER	OPERATION
0	ROOT @ S	0,0		Seed
17	S @ NP.NOM-SUB NP.NOM-PLINK VCMAIN M.S	0,0		Predictor
57	NP.NOM-SUB @ N.PROP N.PROP	0,0		Predictor
119	N.PROP @ قبلائی	0,0		Predictor
CHART(1)=قبلائی				
STATEID	RULE	DOT-POSITON	BACK-POINTER	OPERATION
0	N.PROP قبلائی @	0,1		Scanner
1	NP.NOM-SUB N.PROP @ N.PROP	0,1	1-0	Completer
4	N.PROP @ خان	1,1		Predictor
CHART(2)=خان				
STATEID	RULE	DOT-POSITON	BACK-POINTER	OPERATION
0	N.PROP خان @	1,2		Scanner
1	NP.NOM-SUB N.PROP N.PROP @	0,2	1-0 2-0	Completer
5	S NP.NOM-SUB @ NP.NOM-PLINK VCMAIN M.S	0,2	2-1	Completer
35	NP.NOM-PLINK @ N.PROP	2,2		Predictor
90	N.PROP @ منگول	2,2		Predictor
CHART(3)=منگول				
STATEID	RULE	DOT-POSITON	BACK-POINTER	OPERATION
0	N.PROP منگول @	2,3		Scanner
1	NP.NOM-PLINK N.PROP @	2,3	3-0	Completer
5	S NP.NOM-SUB NP.NOM-PLINK @ VCMAIN M.S	0,3	2-1 3-1	Completer
17	VCMAIN @ V.COP.PAST	3,3		Predictor
46	V.COP.PAST @ تھا	3,3		Predictor
CHART(4)=تھا				
STATEID	RULE	DOT-POSITON	BACK-POINTER	OPERATION
0	V.COP.PAST تھا @	3,4		Scanner
1	VCMAIN V.COP.PAST @	3,4	4-0	Completer
2	S NP.NOM-SUB NP.NOM-PLINK VCMAIN @ M.S	0,4	2-1 3-1 4-1	Completer
4	M.S @ -	4,4		Predictor
CHART(5)=-				
STATEID	RULE	DOT-POSITON	BACK-POINTER	OPERATION
0	M.S - @	4,5		Scanner
1	S NP.NOM-SUB NP.NOM-PLINK VCMAIN M.S @	0,5	2-1 3-1 4-1 5-0	Completer
2	ROOT S @	0,5	5-1	Completer

Fig. 10. Charts generated by the Urdu parser for a sentence 7.

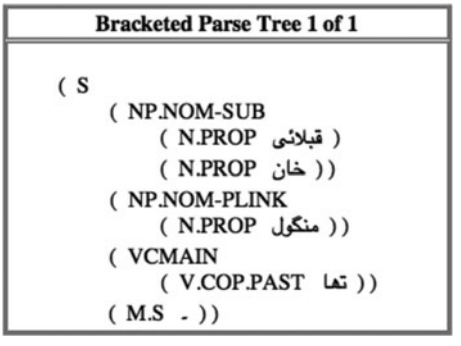


Fig. 11. A bracketed parse tree generated by the Urdu parser for a sentence 7.

data (for the development of test suite). First, a CFG is extracted computationally from the training data which is categorized into L and NL type productions. The test data is further divided into 10% held-out data and 10% test data. The held-out data is used in the development of the Urdu parser, while the test data is used for evaluation of results after the completion of the Urdu parser. To make the test data more valuable and reliable for results, ten sentences from each 100 of 1,400 sentences are selected. The test data so contains 140 sentences in all. In the test data, the average length is 13.73 words per sentence, the minimum length of

https://doi.org/10.1017/S1351324915000133 Published online by Cambridge University Press

Table 1. *Evaluation results of the Urdu parser*

	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	1,922	2,449	3,107	2,531	0.968	0.788	0.869	329
CBP	220	...	...	...	...	0.680	0.663	...	...
SP	200	...	...	...	...	0.759	0.568	...	...
MPSRP	100	...	...	...	...	...	0.74	...	...

a sentence found is five words and the maximum length of a sentence is forty-six words. All items which can exist in a normal text are considered e.g., punctuation, null elements, diacritics, headings, regard titles, Hadees (statements of the prophets), anaphora within a sentence, and others except the unknown[9] words which will be dealt statistically when this parser will be shifted toward a probabilistic one.

The PARSEVAL measures are used to evaluate the results of the Urdu parser computationally[10]. The evaluation results of the Urdu parser depicted in Table 1 includes the number of sentences in test, gold and matched set, precision and recall percentages, f-score, and crossing brackets. The PARSEVAL measures are calculated in two ways on the basis of constituents and their results are presented in rows as A-1 and T-2. The results in row A-1 of the table are calculated on the average basis and the results in row T-2 are calculated on the basis of totality, which is a standard method of calculation. The results given in Table 1 generally describe that for the *Length* of a sentence in words, the parser produces a number of constituents in a *Test* parse. If *Test* parse contains such number of constituents then such number of constituents are *Matched* with the constituents available in the *Gold* data. *Precision* is calculated by dividing the *Matched* constituents with the constituents available in the *Test* parse, while the *Recall* is calculated by dividing the *Matched* constituents with the constituents available in the *Gold* data. The *F-score* is the harmonic mean of *Precision* and *Recall.*

A two stage constraint based dependency parser for Hindi (Bharati *et al.* 2008) was trained on 1,185 sentences from the Hindi treebank (Begum *et al.* 2008). The test data contained 220 sentences but their average length was not reported. The evaluation results in the form of unlabeled attachment (UA), labeled (L) and labeled attachment (LA) are provided by Bharati *et al.* (2008). From which only the percentage values for LA can be compared with the results of the Urdu parser because LA counts only the correct number of heads and dependency arcs with its labels. The percentage value of LA reported without recall is 75%. However, the LA with recall (LA-R) percentages are also provided for twelve different labels, whose average is 66.32%. As the LA counts only the correct number of heads and dependency arcs with its labels or constituents produced in the parse, so according to this, the Urdu parser

[9] Any word which is not listed in the lexicon of the Urdu parser

[10] The results are calculated through a module developed in the Perl programming environment.

https://doi.org/10.1017/S1351324915000133 Published online by Cambridge University Press

has the constituent accuracy or LA in T-2 as 96.76%[11], which is 21.76% better than the two stage Hindi dependency parser. On the other hand, for comparison of the LA-R, we can compare the calculated average percentage 66.32% with the recall value in A-1 of the Table 1 logically, but being on the safe side after comparing this value with the recall percentage in T-2, it can be concluded that the Urdu parser outperforms the two stage Hindi dependency parser with 12.48% in recall.

The Urdu parser outperforms the simple Hindi dependency parser (SP) by Bharati *et al.* (2009) with an additional recall of 22%. In (Bharati *et al.* 2009), only precision and recall percentages are given. That is why only the precision and recall percentages of 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 both the parsers concludes that the Urdu parser outperforms the simple Hindi dependency parser with 22% increase in recall.

The MPSRP (Mukhtar *et al.* 2012) for Urdu parsed seventy-four 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 *Gold* data. As recall percentage in our case is 78.8%, so we can say that the Urdu parser beats the MPSRP with a 4.8% increase in recall. On the other hand, from the first 100 sentences of the test data, the Urdu parser provides eighty-nine sentences with parsed solutions. Comparatively, the Urdu parser has 15% more accuracy than the MPSRP, but the parsed solutions are not compared with the *Gold* data. So, by considering the safe side, we can repeat our argument that the Urdu parser beats the MPSRP with a 4.8% increase in recall.

7 Conclusion

To fulfill the encoding requirements of rich information for MRL Urdu, a grammar having rich morphological, POS, syntactical and functional information was extracted from the URDU.KON-TB treebank. The existing dynamic programming algorithm known as the Earley algorithm was extended to accomplish the Urdu parsing needs. Several Urdu parsing issues have been resolved through this extension. After removal of issues, the output of the Urdu parser is so directed, speedy and refined in a sense that no extra or irrelevant L type productions are introduced by the parser. It is really hard now that the Urdu parser will select a wrong choice of

[11] Getting percentage by dividing the correct/matched constituents with the constituents produced in the test parse by the Urdu parser as ((2449x100)/2531).

https://doi.org/10.1017/S1351324915000133 Published online by Cambridge University Press

production because it has a tendency to correct itself. Urdu parser is boosted up to find a solution if the solution is available in any chart at any position. Empty productions are handled partially. The calculation of back pointers is helpful in building the bracketed parse trees of a given sentence. The parser has a mechanism through which failed parsed output is automatically edited and corrected. These all features concludes an Urdu parser with rich morphological, POS, syntactical and functional information in the form of a grammar in it. The parser with rich morphological information is comparable or better than the state-of-the-art in Urdu/Hindi domain. The evaluation results were compared with three parsers, which included the MPSRP for Urdu, a two stage constraint based dependency parser for Hindi and the simple Hindi dependency parser. The Urdu parser has better evaluation results as compared to these three parsers presented. It 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 size of the treebank can also be increased. At present, the treebank contains only 1,400 annotated sentences, which can 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. The issues discussed about the handling of empty productions will be addressed further, when this parser will be shifted toward a statistical one.

Acknowledgements

I would like to thank Prof. Dr. Miriam Butt, University of Konstanz, Germany, who invested her full effort in guiding me to achieve this goal of parsing for morphologically rich language Urdu. Moreover, special thanks goes to unknown reviewers of this article because the quality and presentation of this article is sufficiently improved with their valuable comments, critics, suggestions and support.

References

Abbas, Q. 2012. Building a hierarchical annotated corpus of urdu: the URDU.KON-TB Treebank. *Lecture Notes in Computer Science* **7181**(1): 66–79.

Abbas, Q. 2014a. *Building Computational Resources : The URDU.KON-TB Treebank and the Urdu Parser*. PhD thesis, Germany: KOPS, University of Konstanz.

Abbas, Q. 2014b. Exploiting language variants via grammar parsing having morphologically rich information. In *Proceedings of the EMNLP Workshop on Language Technology for Closely Related Languages and Language Variants*, Association for Computational Linguistics, Doha, Qatar, pp. 35–45.

Abbas, Q. 2014c. Semi-semantic part of speech annotation and evaluation. In *Proceedings of ACL 8th Linguistic Annotation Workshop held in conjunction with COLING*, Association for Computational Linguistics, Dublin, Ireland, pp. 75–81.

Abbas, Q., Karamat, N., and Niazi, S. 2009. Development of tree-bank based probabilistic grammar for Urdu language. *International Journal of Electrical & Computer Science* **9**(09): 231–235.

Abbas, Q., and Nabi Khan, A. 2009. Lexical functional grammar for Urdu modal verbs. In *IEEE International Conference on Emerging Technologies (ICET)*, IEEE, Islamabad, Pakistan, pp. 7–12.

https://doi.org/10.1017/S1351324915000133 Published online by Cambridge University Press

Abbas, Q., and Raza, G. 2014. A computational classification of Urdu dynamic copula verb. *International Journal of Computer Applications* **85**(10): 1–12.

Abbas, Q., Zia, T., and Khan, A. N. 2015. Syntactic and semantic analysis of Urdu modal verbs using XLE parser. *International Journal of Computer Applications* **107**(10): 39–46.

Agrawal, B., Agarwal, R., Husain, S., and Sharma, D. M. 2013. An automatic approach to treebank error detection using a dependency parser. In *Computational Linguistics and Intelligent Text Processing,* pp. 294–303. Samos, Greece. Springer-Verlag.

Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. 2007. *Compilers: Principles, Techniques, & Tools*, vol. 1009. USA: Pearson/Addison Wesley.

Ali, W., and Hussain, S. 2010. Urdu dependency parser: a data-driven approach. In *Proceedings of Conference on Language and Technology (CLT10)*, SNLP, Lahore, Pakistan.

Appel, A. W., and Palsberg, J. 2007. *Modern Compiler Implementation in Java*. New York: Cambridge University Press.

Arun, A., and Keller, F. 2005. Lexicalization in crosslinguistic probabilistic parsing: the case of French. In *Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics*, Association for Computational Linguistics, pp. 306–313. Ann Arbor, Michigan, United States.

Aycock, J., and Horspool, R. N. 2002. Practical earley parsing. *The Computer Journal* **45**(6): 620–630.

Begum, R., Husain, S., Dhwaj, A., Sharma, D. M., Bai, L., and Sangal, R. 2008. Dependency annotation scheme for Indian languages. In *Proceedings of The 3rd International Joint Conference on Natural Language Processing (IJCNLP)*, IIIT, Hyderabad, India, pp. 721–726.

Bharati, A., Bhatia, M., Chaitanya, V., and Sangal, R. 1996. Paninian grammar framework applied to english. Technical Report, TRCS-96-238, CSE, IIIT, Kanpur, India.

Bharati, A., Chaitanya, V., Sangal, R., and Ramakrishnamacharyulu, K. 1995. *Natural Language Processing: A Paninian Perspective*. New Delhi: Prentice-Hall of India.

Bharati, A., Gupta, M., Yadav, V., Gali, K., and Sharma, D. M. 2009. Simple parser for Indian languages in a dependency framework. In *Proceedings of the 3rd Linguistic Annotation Workshop*, Association for Computational Linguistics, Singapore, pp. 162–165.

Bharati, A., Husain, S., Sharma, D. M., and Sangal, R. 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)*, COLIPS, Thailand.

Bhat, R. A., Jain, S., and Sharma, D. M. 2012. Experiments on dependency parsing of Urdu. In *Proceedings of the 11th International Workshop on Treebanks and Linguistic Theories (TLT11)*, edi-colibri, Portugal.

Butt, M. 1993. Hindi-Urdu infinitives as NPs. In Y. Kachru (ed.), *South Asian Language Review: Special Issue on Studies in Hindi-Urdu*, vol. 3(1), pp. 51–72. New Delhi: Creative Publishers.

Butt, M. 2003. The light verb jungle. In *Harvard Working Papers in Linguistics*, Harvard University, USA.

Butt, M. 2010. The light verb jungle: still hacking away. In M. Amberber, M. Harvey, and B. Baker (eds.), *Complex Predicates in Cross-Linguistic Perspective*, pp. 48–78. USA: Cambridge University Press.

Butt, M., and King, T. H. 2007. Urdu in a parallel grammar development environment. In T. Takenobu, and C. -R. Huang (eds.), *Language Resources and Evaluation: Special Issue on Asian Language Processing: State of the Art Resources and Processing*, vol. 41, pp. 191–207. Netherlands: Kluwer Academic Publishers.

Butt, M., and Ramchand, G. 2001. Complex aspectual structure in Hindi/Urdu. In M. Liakata, B. Jensen, and D. Maillat (eds.), *Oxford Working Papers in Linguistics, Philology and Phonetics*, pp. 1–30. UK: Oxford University.

Butt, M., and Rizvi, J. 2010. Tense and aspect in Urdu. In P. Cabredo-Hofherr, and B. Laca (eds.), *Layers of Aspect*. Stanford: CSLI Publications.

https://doi.org/10.1017/S1351324915000133 Published online by Cambridge University Press

Chomsky, N. 1956. Three models for the description of language. *IRE Trans. Inform. Theory* **2**(3): 113–124.

Collins, M., Ramshaw, L., Hajič, J., and Tillmann, C. 1999. A statistical parser for Czech. In *Proceedings of the 37th Annual Meeting of the Association for Computational Linguistics*, Association for Computational Linguistics, Maryland, USA, pp. 505–512.

Corazza, A., Lavelli, A., Satta, G., and Zanoli, R. 2004. Analyzing an Italian treebank with state-of-the-art statistical parsers. In *Proceedings of the 3rd Workshop on Treebanks and Linguistic Theories (TLT 2004)*, Kluwer Academic Publishers, Tuebingen, Germany.

Dubey, A., and Keller, F. 2003. Probabilistic parsing for German using sister-head dependencies. In *Proceedings of the 41st Annual Meeting on Association for Computational Linguistics-Volume 1*, Association for Computational Linguistics, Sapporo, Japan, pp. 96–103.

Earley, J. 1970. An efficient context-free parsing algorithm. *Communications of the ACM* **13**(2): 94–102.

Earley, J. C. 1968. *An Efficient Context-Free Parsing Algorithm*. PhD thesis, PA, USA: Carnegie Mellon University, Pittsburgh.

Hopcroft, J. E., Motwani, R., and Ullman, J. D. 2001. *Introduction to Automata Theory, Languages, and Computation*. USA: Addison-Wesley.

Jiang, W., Xiong, H., and Liu, Q. 2009. Mutipath shift-reduce parsing with online training. In *Proceedings of 1st Workshop on Chinese Syntactic Parsing Evaluation*, CIPS ParsEval, Beijing.

Khan, A. J. 2006. *Urdu/Hindi: An Artificial Divide: African Heritage, Mesopotamian Roots, Indian Culture & Britiah Colonialism*. USA: Algora Publishers.

Kulick, S., Gabbard, R., and Marcus, M. 2006. Parsing the Arabic treebank: analysis and improvements. In: J. Hajič, and J. Nivre (eds.), *Proceedings of the TLT06*, Institute of Formal and Applied Linguistics, Prague, Czech Republic, pp. 31–42.

Leblanc, R., and Fischer, C. N. 1988. *Crafting a Compiler*. USA: Benjamin-Cummings Publishing Company.

Lewis, P. M., Simons, G. F., and Fennig, C. D. 2013. *Ethnologue: Languages of the World,* 17th ed. Dallas: SIL International.

McDonald, R., Pereira, F., Ribarov, K., and Hajič, J. 2005. Non-projective dependency parsing using spanning tree algorithms. In *Proceedings of the Conference on Human Language Technology and Empirical Methods in Natural Language Processing*, Association for Computational Linguistics, pp. 523–530. Vancouver, B.C., Canada.

McLane, J. R. 1970. *The Political Awakening in India*. New Jersey, US: Prentice Hall.

Mukhtar, N., Khan, M. A., and Zuhra, F. T. 2011. Probabilistic context free grammar for Urdu. *Linguistic and Literature Review* **1**(1): 86–94.

Mukhtar, N., Khan, M. A., and Zuhra, F. T. 2012. Algorithm for developing Urdu Probabilistic Parser. *International journal of Electrical and Computer Sciences* **12**(3): 57–66.

Mukhtar, N., Khan, M. A., Zuhra, F. T., and Chiragh, N. 2012. Implementation of Urdu probabilistic parser. *International Journal of Computational Linguistics (IJCL)* **3**(1): 12–20.

Nivre, J., Hall, J., Nilsson, J., Chanev, A., Eryigit, G., Kübler, S., Marinov, S., and Marsi, E. 2007. MaltParser: a language-independent system for data-driven dependency parsing. *Natural Language Engineering* **13**(2): 95–135.

Sikkel, K., and Nijholt, A. 1997. *Parsing of Context-Free Languages*. Berlin/Heidelberg, Germany: Springer Verlag.

Tesnière, L., and Fourquet, J. 1959. *Eléments de Syntaxe Structurale*, vol. 1965. Paris: Klincksieck.

Tsarfaty, R., Seddah, D., Goldberg, Y., Kuebler, S., Candito, M., Foster, J., Versley, Y., Rehbein, I., and Tounsi, L. 2010. Statistical parsing of morphologically rich languages (SPMRL): what, how and whither. In *Proceedings of the NAACL HLT 2010 1st Workshop on Statistical Parsing of Morphologically-Rich Languages*, Association for Computational Linguistics, Los Angeles, CA.

https://doi.org/10.1017/S1351324915000133 Published online by Cambridge University Press

Tsarfaty, R., Seddah, D., Kübler, S., and Nivre, J. 2013. Parsing morphologically rich languages: introduction to the special issue. *Computational Linguistics* **39**(1): 15–22.

Tsarfaty, R., and Sima'an, K. 2007. Three-dimensional parametrization for parsing morphologically rich languages. In *Proceedings of the 10th International Conference on Parsing Technologies*, Association for Computational Linguistics, Prague, Czech Republic, pp. 156–167.

https://doi.org/10.1017/S1351324915000133 Published online by Cambridge University Press