REVERSIBLE PUSHDOWN AUTOMATA AND BIDIRECTIONAL PARSING

MARK-JAN NEDERHOF
Humanities Computing, University of Groningen
P.O. Box 716, 9700 AS Groningen, The Netherlands
e-mail: markjan@let.rug.nl

ABSTRACT
We introduce a new kind of pushdown automaton, called *reversible* pushdown automaton, which has the property that it can process input from left to right as well as from right to left. We show that such automata constitute a new approach to *bidirectional* parsing, which means that parsing in both directions may be done alternately.

1 Introduction

Traditionally, parsing algorithms which are studied in formal language theory process input from left to right. The motivation for this monodirectionality may be found in psycho-linguistic arguments, both for programming languages and for natural languages. However, some arguments can be found in favour of bidirectional processing. By *bidirectional* processing of input we mean that the order in which the individual input symbols are read is neither strictly from left to right nor strictly from right to left, but alternately in both directions.

For example, for error handling in the case of programming languages, it may be advantageous to read the input backwards starting at an error position after having reached that position by normal left-to-right parsing [4]. Another form of bidirectional parsing is *island-driven parsing* [14] in the area of speech recognition: processing of input starts at those positions within the input where acoustic perception could be done most reliably. Related to the island-driven algorithms are the *head-driven algorithms* [11], which process the parts of the input in an order which allows some particular parts of an underlying grammar to be handled before other parts are. Such head-driven algorithms also lead to bidirectional processing of the input.

Incremental parsing algorithms [6] are similar to island-driven parsing: modified input is processed starting at the substrings within the input that were not modified, making use of subparses already found for those substrings. Conceptually or actually, the order in which modified substrings are processed may be in both directions.

A number of approaches to bidirectional parsing have been proposed. The oldest approach deals with two-way pushdown automata [1]. These automata differ from ordinary pushdown automata in that each transition may either cause an increment or a decrement of the input pointer. The order in which the input symbols are processed is however completely fixed by the definition of each such automaton.

The island-driven approach in [14] allows more flexibility in the order of processing. Regrettably, the algorithms are very complicated and not systematically derived. The same holds for the algorithm in [13] (see also [6]), which proposes a bidirectional algorithm consisting of an LR parser and a reverse LR parser, which collaborate to process the input both from left to right and from right to left in different substrings of the input.

In this paper we introduce a new approach to bidirectional parsing. The approach is based on the existence of pushdown automata which are *reversible*, which means that they can process input both from left to right and from right to left. This is accomplished by choosing a symmetrical definition for the allowable transitions, so that they are interpretable in both directions. Such automata can be simulated by fixed tabular techniques in a way that yields different bidirectional algorithms for different automata.

2 Context-free grammars and reversible PDAs

A context-free grammar $G = (T, N, S, P)$ consists of two finite disjoint sets T and N of terminals and nonterminals, respectively, a start symbol $S \in N$, and a finite set of rules P. Every rule has the form $A \rightarrow \alpha$, where $A \in N$ and $\alpha \in (T \cup N)^*$.

We generally use symbols $A, B, C, \ldots$ to range over N, symbols $a, b, c, \ldots$ to range over T, and symbols $\alpha, \beta, \gamma, \ldots$ to range over $(T \cup N)^*$.

A pushdown automaton (PDA) is a 5-tuple $(\Sigma, \Delta, X_{initial}, X_{final}, \mathcal{T})$, where Σ, Δ and $\mathcal{T}$ are finite sets of input symbols, stack symbols and transitions, respectively.[1] Both $X_{initial}$ and X_{final} are distinguished stack symbols in Δ.

We consider a fixed input string $a_1 \ldots a_n \in \Sigma^*$, $0 \leq n$. A *configuration* of the PDA is a pair (δ, i) consisting of a stack $\delta \in \Delta^*$ and an input position i, with $0 \leq i \leq n$. In our notation, the right-most element of δ represents the top of the stack. The *initial* configuration is of the form $(X_{initial}, 0)$; the *final* configuration is of the form (X_{final}, n).

The transitions in $\mathcal{T}$ are objects of the form $\delta_1 \mapsto \delta_2$ or of the form $\delta_1 \overset{a}{\mapsto} \delta_2$, where $\delta_1, \delta_2 \in \Delta^*$ and $a \in \Sigma$. For a fixed PDA and input string $a_1 \ldots a_n$ we define the binary relation $\vdash$ on configurations as the least relation satisfying, for all i and δ, $(\delta\delta_1, i) \vdash (\delta\delta_2, i)$ if there is a transition $\delta_1 \mapsto \delta_2$, and $(\delta\delta_1, i-1) \vdash (\delta\delta_2, i)$ if there is a transition $\delta_1 \overset{a}{\mapsto} \delta_2$ such that $a = a_i$.

The input string $a_1 \ldots a_n$ is recognized by a PDA if $(X_{initial}, 0) \vdash^* (X_{final}, n)$. An input which can be recognized is called a *sentence*. For a certain PDA $\mathcal{A}$, the set of all such sentences is called the *language accepted by* $\mathcal{A}$. A PDA is called *deterministic* if for all possible configurations (δ, i) for some input there is at most one configuration (δ', i') such that $(\delta, i) \vdash (\delta', i')$.[2]

[1] Note that we have no notion of *state* as some definitions of PDAs have. This does not affect the descriptive power, since states can be encoded into the stack symbols and the transitions.

[2] According to the above definition, deterministic PDAs cannot describe all deterministic (i.e. LR(k)) languages since our definition requires the stack to contain a single element upon recognition

stack	input
$[S \to \bullet\ aAe]$	$\mid abcde$
$[S \to a \bullet Ae]$	$a \mid bcde$
$[S \to a\underline{A}e][A \to \bullet\ bAd]$	$a \mid bcde$
$[S \to a\underline{A}e][A \to b \bullet Ad]$	$ab \mid cde$
$[S \to a\underline{A}e][A \to b\underline{A}d][A \to \bullet\ c]$	$ab \mid cde$
$[S \to a\underline{A}e][A \to b\underline{A}d][A \to c\ \bullet]$	$abc \mid de$
$[S \to a\underline{A}e][A \to bA \bullet d]$	$abc \mid de$
$[S \to a\underline{A}e][A \to bAd\ \bullet]$	$abcd \mid e$
$[S \to aA \bullet e]$	$abcd \mid e$
$[S \to aAe\ \bullet]$	$abcde \mid$

The position of the input pointer i is indicated by a vertical line just after the i-th input symbol. Note that this sequence of configurations may be seen as a top-down recognition process starting with the initial configuration, but equally well as a top-down recognition process starting with the *final* configuration, by reading this table from the last line upwards.

Figure 1: Behaviour of an RPDA resulting from Construction 1.

The transitions in $\mathcal{T}$ of a *reversible* PDA (RPDA) are of one of the forms $XY \mapsto Z$, $Z \mapsto XY$ or $X \overset{a}{\mapsto} Y$, where $X, Y, Z \in \Delta$ and $a \in \Sigma$. The first kind is called *popping transition*, the second *pushing transition*, and the third *reading transition*.

What makes this kind of PDA reversible is that a pushing transition may also be seen as a popping transition performed backwards, and vice versa. This prevents any undesirable bias towards either left-to-right or right-to-left processing of the input.

It is important to note that reversibility does not add to the descriptive power: the languages accepted by RPDAs are the context-free languages. That reversibility does also not restrict the descriptive power is witnessed by the following example of a construction of an RPDA from a context-free grammar.

Construction 1 Consider a context-free grammar $G = (T, N, S, P)$. Without loss of generality, assume that there is only one rule of the form $S \to \sigma$. Construct the RPDA with the transitions below. The stack symbols are of the form $[A \to \alpha \bullet \beta]$, where $A \to \alpha\beta \in P$, or $[A \to \alpha\underline{B}\beta]$, where $A \to \alpha B\beta \in P$; as $X_{initial}$ we take $[S \to \bullet\ \sigma]$, as X_{final} we take $[S \to \sigma\ \bullet]$.

$$\begin{array}{llll}
[A \to \alpha \bullet B\beta] & \mapsto & [A \to \alpha\underline{B}\beta]\ [B \to \bullet\ \gamma] & \text{for all } A \to \alpha B\beta, B \to \gamma \in P \\
[A \to \alpha\underline{B}\beta]\ [B \to \gamma\ \bullet] & \mapsto & [A \to \alpha B \bullet \beta] & \text{for all } A \to \alpha B\beta, B \to \gamma \in P \\
[A \to \alpha \bullet a\beta] & \overset{a}{\mapsto} & [A \to \alpha a \bullet \beta] & \text{for all } A \to \alpha a\beta \in P
\end{array}$$

Consider this construction of top-down recognizers applied to the grammar with the rules: $S \to aAe$, $A \to bAd$, $A \to c$. That the input *abcde* is recognized by the resulting RPDA is shown in Figure 1.

3 Tabular simulation of RPDAs

The task of determining recognition of some input by some PDA is complicated by possible nondeterminism of that PDA. A naive way to solve this task is to compute all

[7]. This can however easily be solved by introducing endmarkers. We will not assume determinism of PDAs here and therefore this issue is outside the scope of this paper.

sequences of steps $(X_{initial}, 0) \vdash^* (\delta, i)$ that the PDA may perform. Regrettably, there may be exponentially (and for some PDAs even infinitely) many of such sequences.

However, it has been shown that all such sequences may be simulated in polynomial time by *tabular* algorithms. For example, the RPDAs resulting from Construction 1 can be simulated by a tabular algorithm similar to Earley's algorithm [5] (see also [10, Chapter 1]). Simulation of arbitrary PDAs, generalizing the results by Earley, was presented in [2]. We will present here a variant of that idea to RPDAs.

The main data structure we will need is a table (set) of *items* of the form (X, j, Y, i), where $X, Y \in \Delta$ and $0 \leq j \leq i \leq n$. Such an item represents a piece of tabulated computation by an RPDA, namely that if X occurs on top of the stack at input position j, then by reading the input up to position i this stack element is replaced by Y.[3] Because of the reversible nature of our RDPAs, this can of course also be seen the other way around: the input is read from right to left, from i to j, and Y is replaced by X.

The following tabular algorithm simulates an RPDA by processing the input from left to right.

Algorithm 1 (On-line) Consider a fixed RPDA. Assume the input is $a_1 \ldots a_n$. Let the set of items U be $\{(X_{initial}, 0, X_{initial}, 0)\}$. Perform one of the following three steps as long as one of them is applicable.

push Choose a pair, not considered before, consisting of a transition $Z \mapsto XY$ and an item $(W, j, Z, i) \in U$. Add to U the item (Y, i, Y, i) (if it is not already there).

pop Choose a quadruple, not considered before, consisting of two transitions $Z_1 \mapsto XY_1$ and $XY_2 \mapsto Z_2$ and items $(W, k, Z_1, j), (Y_1, j, Y_2, i) \in U$. Add to U the item (W, k, Z_2, i).

read Choose a pair, not considered before, consisting of a transition $X \overset{a}{\mapsto} Y$ and an item $(W, j, X, i-1) \in U$ such that $a_i = a$. Add to U the item (W, j, Y, i).

The input is recognized if $(X_{initial}, 0, X_{final}, n) \in U$.

This algorithm performs *on-line* processing of the input. Informally, this means that no steps of the RPDA are simulated which could not actually be performed by the RPDA processing the input from left to right. One of the consequences of this on-line property is that if no sequence of steps of the RPDA would lead past the i-th input symbol, then Algorithm 1 would also not compute any items (X, j, Y, i'), with $i' > i$. The usefulness of such on-line processing for the location of syntax errors is demonstrated in [9].

Formally, we have that an item (X, j, Y, i) is eventually added to U by Algorithm 1 if and only if

1. either $X = X_{initial} \wedge j = 0$ or
   $(X_{initial}, 0) \vdash^* (\delta Z, j) \vdash (\delta W X, j)$, for some δ, Z and W; and

[3]Such items bear some resemblance to the table elements from [3, 8, 12].

2. $(X, j) \vdash^* (Y, i)$.

From this characterization of the table U resulting from Algorithm 1, its correctness as simulation of RPDAs immediately follows. Note that the first of these two conditions represents the on-line property. An off-line algorithm, where this first condition is absent, is investigated shortly.

As an example, we consider an RPDA with the transitions $X \overset{a}{\mapsto} P$, $P \mapsto YX$, $X \overset{c}{\mapsto} Z$, $Z \overset{a}{\mapsto} Q$, $YQ \mapsto Z$, where $X_{\mathit{initial}} = X$ and $X_{\mathit{final}} = Z$. The table resulting from Algorithm 1 on the input *aacaa* is given in Figure 2a. We use a pictorial representation for a set of items U: we have one column of nodes for each input position; the nodes in each column represent stack symbols that may be on top of the stack at the corresponding input position. Items (X, i, Y, j) are represented by an arc from a node labelled X in the i-th column to a node labelled Y in the j-th column.

An alternative to the on-line simulation of PDAs is such that the simulation on each substring of the input is performed irrespective of the preceding input. This is called *off-line* simulation. This kind of simulation was already presented in [1], be it for a more general type of automaton, viz. for the *two-way* PDAs.

Algorithm 1 can be adapted to perform off-line simulation:

Algorithm 2 (Off-line) Consider a fixed RPDA and input as usual. Let the set of items U be $\{(X, i, X, i) \mid X \in \Delta \wedge 0 \leq i \leq n\}$. Perform one of the popping or reading steps from Algorithm 1 as long as one of them is applicable. The pushing step is no longer required.

The input is recognized if $(X_{\mathit{initial}}, 0, X_{\mathit{final}}, n) \in U$.

An item (X, j, Y, i) is eventually added to U by Algorithm 2 if and only if $(X, j) \vdash^* (Y, i)$. This implies that more items will be added to the table U than in the case of Algorithm 1. This can be informally explained by the fact that Algorithm 1 only adds items to the table if they are useful with regard to the previously processed input, whereas Algorithm 2 has no such mechanism for filtering out useless items.

Because we restricted ourselves to reversible PDAs, the on-line and off-line algorithms above can be put in mirrored forms such that they are oriented towards right-to-left processing. For the off-line algorithm this does not change the final form of the table. The on-line algorithm on the other hand changes its behaviour after mirroring so that it filters out items with regard to input to the right instead of input to the left:

Algorithm 3 (On-line, mirrored) Consider a fixed RPDA and input as usual. Let the set of items U be $\{(X_{\mathit{final}}, n, X_{\mathit{final}}, n)\}$. Perform one of the following three steps as long as one of them is applicable.

push Choose a pair, not considered before, consisting of a transition $XY \mapsto Z$ and an item $(Z, i, W, j) \in U$. Add to U the item (Y, i, Y, i).

pop Choose a quadruple, not considered before, consisting of two transitions $XY_1 \mapsto Z_1$ and $Z_2 \mapsto XY_2$ and items $(Z_1, j, W, k), (Y_2, i, Y_1, j) \in U$. Add to U the item (Z_2, i, W, k).

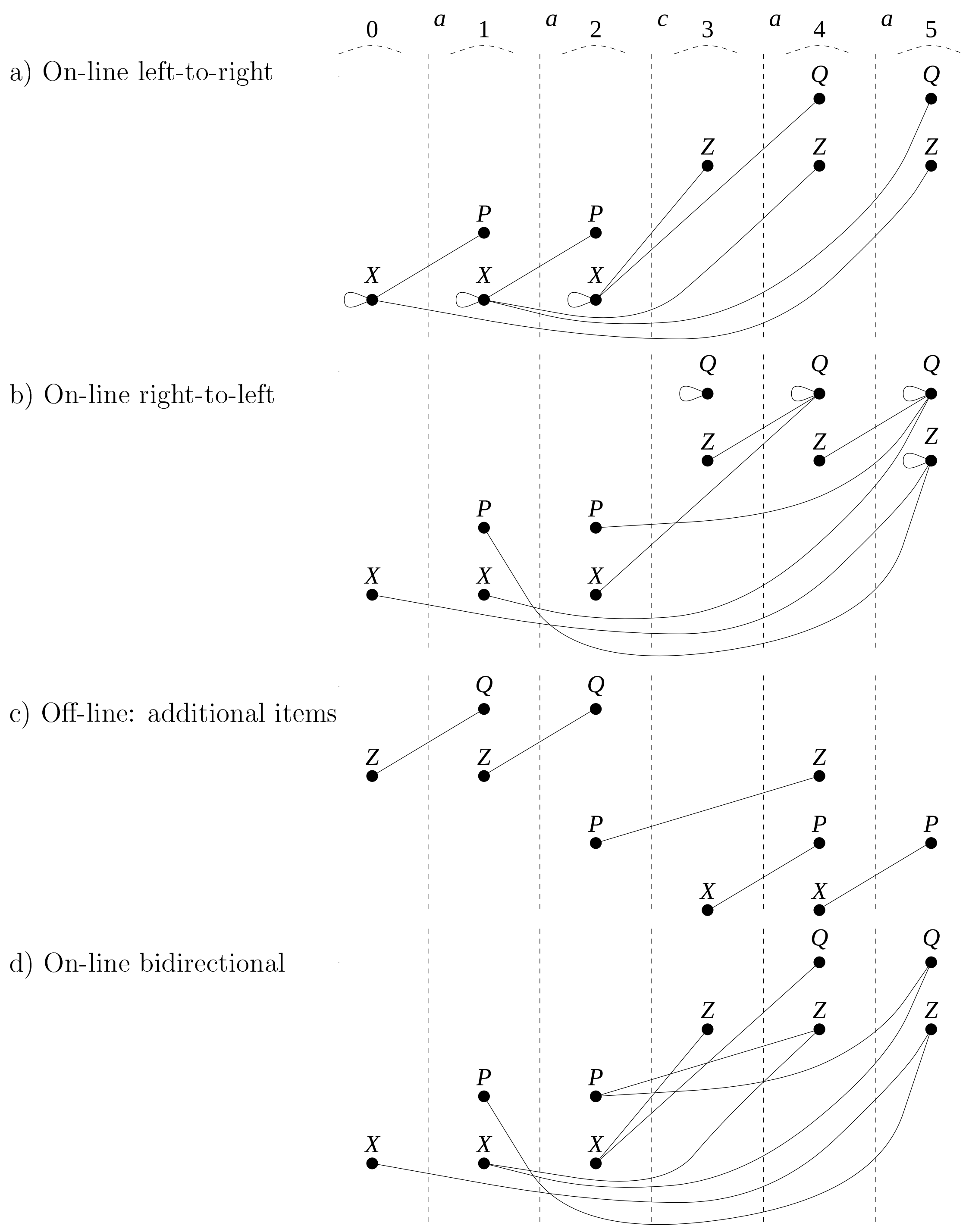

Figure 2: The tables U resulting from various tabular algorithms.

read Choose a pair, not considered before, consisting of a transition $Y \overset{a}{\mapsto} X$ and an item $(X, i+1, W, j) \in U$ such that $a_{i+1} = a$. Add to U the item (Y, i, W, j).

The input is recognized if $(X_{initial}, 0, X_{final}, n) \in U$.

An item (X, j, Y, i) is eventually added to U by Algorithm 3 if and only if

1. either $Y = X_{final} \wedge i = n$ or
   $(\delta W Y, i) \vdash (\delta Z, i) \vdash^* (X_{final}, n)$, for some δ, Z and W; and
2. $(X, j) \vdash^* (Y, i)$.

From this characterization, we conclude that the mirrored on-line algorithm filters out useless items (X, j, Y, i) with regard to the input from position i to position n, whereas Algorithm 1 filters out such items with regard to the input from position 0 to position j.

For the running example, the table U resulting from Algorithm 3 is given in Figure 2b. The table U resulting from the off-line algorithm (Algorithm 2) contains those in Figure 2a and Figure 2b plus some additional items indicated in Figure 2c; we have omitted however the items (X, i, X, i), since such items are in U for all X and i.

4 Bidirectional parsing

In the previous section we have shown that reversible PDAs can be simulated on-line from left to right or from right to left. We have also presented off-line simulation, which has no inherent directionality. In this section we address *bidirectional* parsing, which means that left-to-right and right-to-left processing of input proceed simultaneously and in collaboration.

As the simplest case of bidirectionality we consider the problem of recognizing some input string starting from a distinguished input position, which is not necessarily the left-most or right-most position. From this position we scan the input to the right up to the right-most position, and also to the left down to the left-most position. These two activities collaborate so that items are filtered out which are useless with regard to the already scanned input (a substring of the complete input including the distinguished position).

The on-line algorithms we saw in the previous section both simulated RPDAs starting from a completely known configuration, viz. either the initial or the final configuration. A bidirectional algorithm which operates starting from an arbitrary input position however must allow for any stack that may exist at that input position. In particular, some special arrangements are needed to simulate a popping transition when information concerning the element immediately below the top-of-stack is not available.

We propose to solve this issue by choosing an on-line bidirectional algorithm which allows the left-to-right and right-to-left processing of the input to collaborate by making joint assumptions about the unknown stack elements upon popping transitions.

Intuitively, if the left-to-right activity can proceed if a certain stack element is assumed just below the top-of-stack, then this assumption is made only if the right-to-left activity can proceed under the same assumption; and vice versa.

The complete bidirectional on-line algorithm is given by the following.

Algorithm 4 (Bidirectional) Consider a fixed RPDA and input as usual. Choose some distinguished input position m, where $1 \leq m \leq n$. Let the set of items U be $\{(X, m-1, Y, m) \mid X \overset{a}{\mapsto} Y \in \mathcal{T} \wedge a = a_m\}$. Perform one of the following seven steps as long as one of them is applicable.

(left-to-right) push For $m \leq i$, as in Algorithm 1.

(left-to-right) pop For $m \leq i$, as in Algorithm 1.

(left-to-right) read For $m+1 \leq i$, as in Algorithm 1.

(right-to-left) push For $i < m$, as in Algorithm 3.

(right-to-left) pop For $i < m$, as in Algorithm 3.

(right-to-left) read For $i < m-1$, as in Algorithm 3.

bidirectional pop Choose a triple, not considered before, consisting of two transitions $Z_1 \mapsto XY_1$ and $XY_2 \mapsto Z_2$ and item $(Y_1, j, Y_2, i) \in U$, $j < m \leq i$. Add to U the item (Z_1, j, Z_2, i).

The input is recognized if $(X_{initial}, 0, X_{final}, n) \in U$.

In this algorithm, the bidirectional pop is what constitutes the collaboration between the left-to-right and the right-to-left processes. Both processes are constrained to make the same assumption concerning the stack element X which occurs below the top-most stack elements Y_1 and Y_2.

For a characterization of the table U constructed by the algorithm, we need to distinguish between three kinds of item. An item (X, j, Y, i), $j < m \leq i$, is eventually added to U by Algorithm 4 if and only if

1. $(X, j) \vdash^* (Y, i)$.

An item (X, j, Y, i), $m \leq j$, is eventually added to U if and only if

1. $(Z_1, k) \vdash^* (\delta Z_2, j) \vdash (\delta W X, j)$, for some δ, Z_1, Z_2, $k < m$ and W; and
2. $(X, j) \vdash^* (Y, i)$.

An item (X, j, Y, i), $i < m$, is eventually added to U if and only if

1. $(\delta W Y, i) \vdash (\delta Z_2, i) \vdash^* (Z_1, k)$, for some δ, Z_1, Z_2, $k \geq m$ and W; and
2. $(X, j) \vdash^* (Y, i)$.

By choosing $m = 3$ for the running example, Algorithm 4 results in the table in Figure 2d.

A generalization of Algorithm 4 to more than one distinguished input position is straightforward, and is called *island-driven parsing*. We need one extra step which computes items (Z, k, X, i) from pairs of items $(Z, k, Y, j), (Y, j, X, i)$. We omit the complete description of such an algorithm because of space limitations. See [14] for related ideas.[4]

All tabular algorithms described above have a time complexity of $\mathcal{O}(n^3)$ for general PDAs. For specific PDAs however, the time-complexities may vary significantly between the different algorithms, with regard to constant factors, but also with regard to the *order* of the time-complexities.

5 Input/output reversibility

PDAs may be extended to be pushdown *transducers* [2]: we add *writing* transitions that write output symbols. The tabular algorithms in this paper may be extended to simulate pushdown transducers.

With careful definition, writing transitions are the mirror images of reading transitions, and transduction may be reversed. This input/output reversibility is orthogonal to the left-to-right/right-to-left reversibility that we have discussed above.

6 Conclusions

Reversible pushdown automata constitute a flexible approach to bidirectional parsing: a parsing algorithm may be described by a combination of a PDA construction and the application of a certain tabular simulation algorithm for PDAs. This is in contrast to the algorithms in e.g. [6, 11, 13, 14] that are fixed for a chosen context-free grammar.

Because of the high level of flexibility, our approach is a suitable theoretical framework for studying bidirectional parsing, in that it may clarify existing algorithms and lead to new algorithms.

Acknowledgements

Intensive correspondence with Giorgio Satta has helped me bring the ideas in this paper into being. An anonymous reviewer has given me a useful reference, now included in this paper.

The author is supported by the Dutch Organization for Scientific Research (NWO), under grant 305-00-802.

[4]In [14] the problem of *analysis redundancy* is treated: a single analysis may be computed in various ways, causing spurious ambiguity. There does not seem to be an easy solution to this problem, neither for our framework, nor for the one in [14].

References

[1] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. Time and tape complexity of pushdown automaton languages. *Information and Control*, 13:186–206, 1968.

[2] S. Billot and B. Lang. The structure of shared forests in ambiguous parsing. In *27th Annual Meeting of the Association for Computational Linguistics, Proceedings of the Conference*, pages 143–151, Vancouver, British Columbia, Canada, June 1989.

[3] S.A. Cook. Linear time simulation of deterministic two-way pushdown automata. In *Proceedings of IFIP Congress 71*, volume 1 – Foundations and Systems, pages 75–80, Ljubljana, Yugoslavia, August 1971.

[4] G.V. Cormack. An LR substring parser for noncorrecting syntax error recovery. *SIGPLAN Notices*, 24(7):161–169, 1989.

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

[6] C. Ghezzi and D. Mandrioli. Incremental parsing. *ACM Transactions on Programming Languages and Systems*, 1(1):58–70, July 1979.

[7] M.A. Harrison. *Introduction to Formal Language Theory.* Addison-Wesley, 1978.

[8] N.D. Jones. A note on linear time simulation of deterministic two-way pushdown automata. *Information Processing Letters*, 6(4):110–112, August 1977.

[9] M.-J. Nederhof and E. Bertsch. Linear-time suffix recognition for deterministic languages. Technical Report CSI-R9409, University of Nijmegen, Department of Computer Science, August 1994. To appear in Journal of the ACM.

[10] M.J. Nederhof. *Linguistic Parsing and Program Transformations.* PhD thesis, University of Nijmegen, 1994.

[11] M.J. Nederhof and G. Satta. An extended theory of head-driven parsing. In *32nd Annual Meeting of the Association for Computational Linguistics, Proceedings of the Conference*, pages 210–217, Las Cruces, New Mexico, USA, June 1994.

[12] W. Rytter. Fast recognition of pushdown automaton and context-free languages. *Information and Control*, 67:12–22, 1985.

[13] H. Saito. Bi-directional LR parsing from an anchor word for speech recognition. In *Papers presented to the 13th International Conference on Computational Linguistics*, volume 3, pages 237–242, 1990.

[14] G. Satta and O. Stock. Bidirectional context-free grammar parsing for natural language processing. *Artificial Intelligence*, 69:123–164, 1994.