

Linear-Time Suffix Parsing for Deterministic Languages

MARK-JAN NEDERHOF

University of Groningen, Groningen, The Netherlands

AND

EBERHARD BERTSCH

Ruhr University, Bochum, Germany

Abstract. We present a linear-time algorithm to decide for any fixed deterministic context-free language L and input string w whether w is a suffix of some string in L . In contrast to a previously published technique, the decision procedure may be extended to produce syntactic structures (parses) without an increase in time complexity. We also show how this algorithm may be applied to process incorrect input in linear time.

Categories and Subject Descriptors: D.3.4 [Programming Languages]: Processors—*parsing*; F.4.2 [Mathematical Logic and Formal Languages]: Grammars and Other Rewriting Systems—*parsing*

General Terms: Languages, Theory, Verification

1. Introduction

Efficient procedures for recognition and parsing of context-free languages have been a recurrent topic in the computer science literature for several decades. With regard to the full language class, the fastest generally applicable recognition algorithms exceed quadratic bounds [Valiant 1975], and the best practical algorithms are cubic.¹ Deterministic languages permit linear-time recognition and parsing [Sippu and Soisalon-Soininen 1990].

These results presuppose that a particular input string is to be tested and analyzed for the property of being a *complete* sentence of the language under consideration. Much less is known about the complexity of testing and analysis in cases of *incomplete* input [Lang 1988; 1991; Nederhof 1994].

M.-J. Nederhof was supported by the Dutch Organization for Scientific Research (NWO), under grant 305-00-802.

Authors addresses: M.-J. Nederhof, University of Groningen, Faculty of Arts, P.O. Box 716, 9700 AS Groningen, The Netherlands, e-mail: markjan@let.rug.al; E. Bertsch, Ruhr University, Bochum, Germany, e-mail: eberhard.bertsch@lpi.ruhr-uni-bochum.de.

Permission to make digital/hard copy of part or all of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery (ACM), Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.

© 1996 ACM 0004-5411/96/0500-0524 \$03.50

¹ See, for example, Earley [1970], Graham et al. [1980], Leiss [1990], and Nederhof [1994].

The problems addressed in this article refer to incomplete data. They can be stated as follows. Given an arbitrary deterministic context-free language (defined by an $LR(k)$ grammar, a deterministic pushdown automaton, or otherwise) and a terminal string of length n , determine in a linear number of steps whether that string is a suffix of some word in the language, and if so, describe the structure of the string in terms of parse trees or similar concepts.

The practical relevance of efficient algorithms in this area is immediately evident from previous work on noncorrecting error recovery in compilers.² If errors are not to be “corrected” by modifying the input program in some way or other, a parser must continue without dependable information on its stack. Although there are several alternatives as to how this can be done, Richter argues convincingly for the very general approach, called *suffix analysis* in his paper: “Disregard all previous input, and examine whether the remaining input is a syntactically permissible suffix of some program.”

While all of these publications mention the necessity of using time-efficient algorithms for suffix analysis, the first widely accessible published result giving a linear bound for the class of $LR(1)$ languages appeared in 1994.

That article [Bates and Lavie 1994] provides a linear-time algorithm for suffix *recognition* in the above sense. The requirement of *suffix parsing* is, however, not met by Bates and Lavie [1994]. Furthermore, the technique presented in Bates and Lavie [1994] is intrinsically dependent on left-to-right processing of the input. In this regard, the new algorithm to be described in this paper is strictly more general: Without losing linear-time efficiency of the entire task, we can process the input string in any desired order, producing structural descriptions of segments at arbitrary positions.

Our notational and conceptual framework is not entirely new. In particular, the linear time complexity of our algorithms relies on the concept of *tabulation* of the parsing process, which is based on Earley’s algorithm [Earley 1970] in the case of $LL(1)$ languages and on the dynamic programming algorithm of Lang [Lang 1974; Billot and Lang 1989; Nederhof 1994] for all other deterministic languages including, of course, the $SLR(k)$ languages, the $LALR(k)$ languages, and so forth.

An important observation will be that essentially the *same* collection of parsing steps can be used either for ordinary context-free parsing or for suffix parsing. The data contained initially in the parsing table constitutes the only difference. In the former case, a *single* table element represents the available information that a complete parse will need to be found. In the latter case, *all* table elements that could possibly be constructed by any number of parsing steps for the unknown prefix (preceding the given suffix) are needed for proper initialization.

This paper has the following outline: We begin in the next section by defining some notation and terminology for context-free grammars and pushdown automata. In Section 3, we discuss linear-time suffix recognition for a subset of the deterministic languages, viz. the class of $LL(1)$ languages. Because treatment of $LL(1)$ languages is very simple, this section is relatively easy to understand and therefore prepares the reader for Section 4, which solves the full problem of

² See, for example, Richter [1985], Cormack [1989], van Deudekom and Kooiman [1993], and Grune and Jacobs [1990].

linear-time suffix recognition for deterministic languages. How the special case of LL(1) languages fits into the more general framework of deterministic languages is explained in Section 5.

In Section 6, we discuss how the recognition procedures can be extended to produce all parse trees for a suffix.

Section 7 explains how incorrect input can be processed by repeatedly applying substring parsing, which is derived from suffix parsing. The practical usefulness of our algorithms is argued in Section 8, which discusses realizations using the recursive-descent method. That our ideas are also applicable to substrings of general context-free languages is explained in Section 9.

In Section 10, we compare our approach to those in the existing literature, most notably [Bates and Lavie 1994].

2. Notation

A context-free grammar $G = (T, N, P, S)$ 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 the left-hand side A is an element from N and the right-hand side α is an element from V^* , where V denotes $(N \cup T)$. P can also be seen as a relation on $N \times V^*$.

We generally use symbols A, B, C, \dots to range over N , symbols a, b, c, \dots to range over T , symbols $\alpha, \beta, \gamma, \dots$ to range over V^* , and symbols v, w, x, \dots to range over T^* . We let ϵ denote the empty string.

The relation P is extended to a relation \rightarrow on $V^* \times V^*$ as usual. The reflexive and transitive closure of \rightarrow is denoted by \rightarrow^* .

A pushdown automaton (PDA) operates on a stack, while reading an input string $a_1 \cdots a_n \#$ from left to right. We assume that the *endmarker* $\#$ only occurs as last symbol of the input string. A *configuration* of the automaton is a pair (δ, v) consisting of a stack δ and the remaining input v , which is a suffix of the original input string $a_1 \cdots a_n \#$. We will sometimes however also consider v to be a string not ending in $\#$, especially if we are only interested in recognition of a part of the input. We use symbols X, Y, Z to denote stack symbols.

The *initial configuration* is of the form $(X_{init}, a_1 \cdots a_n \#)$, where the stack is formed by one fixed stack symbol X_{init} . The *final configuration* is of the form (X_{fin}, ϵ) , where the stack is formed by one fixed stack symbol X_{fin} .

We will assume that the allowable steps of a PDA are described by a finite set of *transitions* of the form

$$\delta_1 \xrightarrow{z} \delta_2$$

where δ_1 and δ_2 represent zero or more stack symbols, and z represents the empty string ϵ or a single terminal a .

The application of such a transition $\delta_1 \xrightarrow{z} \delta_2$ is described as follows. If the top-most symbols on the stack are δ_1 , then these may be replaced by δ_2 , provided either $z = \epsilon$, or $z = a$ and a is the first symbol of the remaining input. If $z = a$, then furthermore a is removed from the remaining input.

Formally, for a fixed PDA we define the binary relation \vdash on configurations as the least relation satisfying $(\delta\delta_1, v) \vdash (\delta\delta_2, v)$ if there is a transition $\delta_1 \xrightarrow{\epsilon} \delta_2$, and $(\delta\delta_1, av) \vdash (\delta\delta_2, v)$ if there is a transition $\delta_1 \xrightarrow{a} \delta_2$. The relation \vdash represents the steps that the PDA can make (cf. the notion “parsing step”).

In the case that we consider more than one PDA at the same time, we use symbols $\mapsto_{\mathcal{A}}$ and $\vdash_{\mathcal{A}}$ instead of \mapsto and \vdash if these refer to one particular PDA \mathcal{A} .

The recognition of a certain input v is obtained if starting from the initial configuration for that input we can reach the final configuration by repeated application of transitions, or, formally, if $(X_{init}, v\#) \vdash^* (X_{fin}, \epsilon)$, where \vdash^* denotes the reflexive and transitive closure of \vdash (and \vdash^+ denotes the transitive closure of \vdash). An input v that can be recognized is called a *sentence*. For a certain PDA \mathcal{A} , the set of all such sentences v is called the language *accepted by \mathcal{A}* . A PDA is called *deterministic* if for all possible configurations at most one transition is applicable. The languages accepted by deterministic PDAs are called *deterministic languages*.

3. Suffix Recognition for LL(1) Languages

In this section, we restrict ourselves to a relatively simple subset of the deterministic languages, namely the LL(1) languages.

For a grammar $G = (T, N, P, S)$, we define the lookahead set $\mathcal{LA}(A \rightarrow \alpha)$ for each rule $A \rightarrow \alpha$ as $\{a \mid \exists v, w, x [S\# \rightarrow^* vAw\# \rightarrow^* v\alpha w\# \rightarrow^* vax]\}$.

We define LL(1) recognition by the construction below of pushdown automata from context-free grammars. The PDAs use special transitions of the form $X \xrightarrow{a} XY$, which have the same meaning as transitions of the form $X \xrightarrow{a} XY$, except that the a is not removed from the input; formally, a transition $X \xrightarrow{a} XY$ gives rise to a step $(\delta X, av) \vdash (\delta XY, av)$. The effect of such a transition can also be described by a set of ordinary transitions as defined in Section 2, and therefore this extension adds nothing to the descriptive power of PDAs. However, we will use these special transitions here since they allow more elegant description of LL(1) recognizers:

Construction: **LL-PDA** (PDA construction of LL(1) recognizers)

Consider a context-free grammar $G = (T, N, P, S)$. Without loss of generality, assume that there is only one rule of the form $S \rightarrow \sigma$. Construct the PDA with the transitions below. The stack symbols are of the form $[A \rightarrow \alpha \bullet \beta]$, where $A \rightarrow \alpha\beta \in P$.³ As X_{init} we take $[S \rightarrow \bullet \sigma]$, as X_{fin} we take $[S \rightarrow \sigma \bullet]$.

$$\begin{aligned}
 [A \rightarrow \alpha \bullet B\beta] & \xrightarrow{a} [A \rightarrow \alpha \bullet B\beta][B \rightarrow \bullet \gamma] & \text{for } A \rightarrow \alpha B\beta, B \rightarrow \gamma, \\
 & & a \in \mathcal{LA}(B \rightarrow \gamma) \\
 [A \rightarrow \alpha \bullet a\beta] & \xrightarrow{a} [A \rightarrow \alpha a \bullet \beta] & \text{for } A \rightarrow \alpha a\beta \\
 [A \rightarrow \alpha \bullet B\beta][B \rightarrow \gamma \bullet] & \xrightarrow{\epsilon} [A \rightarrow \alpha B \bullet \beta] & \text{for } A \rightarrow \alpha B\beta, B \rightarrow \gamma
 \end{aligned}$$

The final configuration is to be of the form $([S \rightarrow \sigma \bullet], \#)$.

A grammar for which the above construction yields a deterministic PDA is said to be LL(1). The language accepted by such a PDA is called an LL(1) language.

Example 3.1. Consider the following LL(1) grammar, generating the language of all palindromes over $\{a, b\}$ with a center marker c .

³ Note that rules of the form $A \rightarrow \epsilon \in P$ give rise to a single stack element, viz. $[A \rightarrow \bullet]$, or written in full, $[A \rightarrow \epsilon \bullet \epsilon]$.

stack	input
$[S \rightarrow \bullet A]$	$babcbab\#$
$[S \rightarrow \bullet A][A \rightarrow \bullet bAb]$	$babcbab\#$
$[S \rightarrow \bullet A][A \rightarrow b \bullet Ab]$	$abcbab\#$
$[S \rightarrow \bullet A][A \rightarrow b \bullet Ab][A \rightarrow \bullet aAa]$	$abcbab\#$
$[S \rightarrow \bullet A][A \rightarrow b \bullet Ab][A \rightarrow a \bullet Aa]$	$bcbab\#$
$[S \rightarrow \bullet A][A \rightarrow b \bullet Ab][A \rightarrow a \bullet Aa][A \rightarrow \bullet bAb]$	$bcbab\#$
$[S \rightarrow \bullet A][A \rightarrow b \bullet Ab][A \rightarrow a \bullet Aa][A \rightarrow b \bullet Ab]$	$cbab\#$
$[S \rightarrow \bullet A][A \rightarrow b \bullet Ab][A \rightarrow a \bullet Aa][A \rightarrow b \bullet Ab][A \rightarrow \bullet c]$	$cbab\#$
$[S \rightarrow \bullet A][A \rightarrow b \bullet Ab][A \rightarrow a \bullet Aa][A \rightarrow b \bullet Ab][A \rightarrow c \bullet]$	$bab\#$
$[S \rightarrow \bullet A][A \rightarrow b \bullet Ab][A \rightarrow a \bullet Aa][A \rightarrow bA \bullet b]$	$bab\#$
$[S \rightarrow \bullet A][A \rightarrow b \bullet Ab][A \rightarrow a \bullet Aa][A \rightarrow bAb \bullet]$	$ab\#$
$[S \rightarrow \bullet A][A \rightarrow b \bullet Ab][A \rightarrow aA \bullet a]$	$ab\#$
$[S \rightarrow \bullet A][A \rightarrow b \bullet Ab][A \rightarrow aAa \bullet]$	$b\#$
$[S \rightarrow \bullet A][A \rightarrow bA \bullet b]$	$b\#$
$[S \rightarrow \bullet A][A \rightarrow bAb \bullet]$	$\#$
$[S \rightarrow A \bullet]$	$\#$

FIG. 1. Behavior of a PDA yielded by Construction **LL-PDA**.

$$\begin{aligned}
S &\rightarrow A \\
A &\rightarrow aAa \\
A &\rightarrow bAb \\
A &\rightarrow c
\end{aligned}$$

The PDA from Construction **LL-PDA** recognizes the input $babcbab$ in the sequence of configurations given in Figure 1.

LL(1) recognition can also be done using a tabular algorithm that is a particular variant of Earley's algorithm [Earley 1970; Bouckaert et al. 1975]. This tabular algorithm is given by the following.

Algorithm: **Earley** (Earley's recognizer)

Consider a context-free grammar $G = (T, N, P, S)$. Without loss of generality, assume that there is only one rule of the form $S \rightarrow \sigma$. Assume the input is $a_1 \cdots a_n$. Let $a_{n+1} = \#$. The table U is a set of items of the form $(j, [A \rightarrow \alpha \bullet \beta], i)$, where $A \rightarrow \alpha\beta \in P$ and $0 \leq j \leq i \leq n$. Initialize U to be $\{(0, [S \rightarrow \bullet \sigma], 0)\}$. Perform one of the following steps as long as one of them is applicable.

predictor. For some $(j, [A \rightarrow \alpha \bullet B\beta], i) \in U$ not considered before, add $(i, [B \rightarrow \bullet \gamma], i)$ to U (provided it is not already in U) for all $B \rightarrow \gamma$ such that $a_{i+1} \in \mathcal{LA}(B \rightarrow \gamma)$.

scanner. For some $(j, [A \rightarrow \alpha \bullet a\beta], i-1) \in U$ not considered before, add $(j, [A \rightarrow \alpha a \bullet \beta], i)$ to U , provided $a = a_i$.

completer. For some pair $(h, [A \rightarrow \alpha \bullet B\beta], j), (j, [B \rightarrow \gamma \bullet], i) \in U$ not considered before, add $(h, [A \rightarrow \alpha B \bullet \beta], i)$ to U .

The input is recognized if $(0, [S \rightarrow \sigma \bullet], n) \in U$.

The invariants usually presented for Earley's algorithm are in terms of the grammar. Such an invariant would state, for example, that an item $(j, [A \rightarrow \alpha \cdot \beta], i) \in U$ implies $\alpha \rightarrow^* a_{j+1} \cdots a_i$. Here we are interested however in an invariant in terms of the PDAs resulting from Construction **LL-PDA**. We do this in anticipation of results in the following sections, where we will show that the relation between PDAs resulting from Construction **LL-PDA** and Algorithm **Earley** can be generalized.

The desired invariant now is that eventually $(j, [A \rightarrow \alpha \cdot \beta], i) \in U$ if and only if for the PDA yielded by Construction **LL-PDA**:

- (1) $([S \rightarrow \cdot \sigma], a_1 \cdots a_n \#) \vdash^* (\delta[A \rightarrow \cdot \alpha\beta], a_{j+1} \cdots a_n \#)$, for some δ , and
- (2) $([A \rightarrow \cdot \alpha\beta], a_{j+1} \cdots a_n \#) \vdash^* ([A \rightarrow \alpha \cdot \beta], a_{i+1} \cdots a_n \#)$.

The table U in Earley's algorithm can be partitioned according to the third components i of items $(j, [A \rightarrow \alpha \cdot \beta], i)$. The subset of U consisting of all such items for fixed i is called a *column* of the table, and consists of the items that can be created just after reading the i th input symbol. Note that in Algorithm **Earley** the initialization consists of assigning to column 0 the set $\{(0, [S \rightarrow \cdot \sigma], 0)\}$, and to all other columns the empty set.

For suffix recognition, we need a different initialization, namely one where column 0 is assigned the set I_0^0 , which is defined to be the smallest set satisfying:

- $(0, [S \rightarrow \cdot \sigma], 0) \in I_0^0$.
- For each item $(0, [A \rightarrow \alpha \cdot B\beta], 0) \in I_0^0$ and rule $B \rightarrow \gamma$ we have $(0, [B \rightarrow \cdot \gamma], 0) \in I_0^0$.
- For each item $(0, [A \rightarrow \alpha \cdot a\beta], 0) \in I_0^0$ we have $(0, [A \rightarrow \alpha a \cdot \beta], 0) \in I_0^0$.
- For each pair of items $(0, [A \rightarrow \alpha \cdot B\beta], 0), (0, [B \rightarrow \gamma \cdot], 0) \in I_0^0$ we have $(0, [A \rightarrow \alpha B \cdot \beta], 0) \in I_0^0$.

If the grammar is reduced (i.e., for all $A \in N$, we have $\exists v, w, x[S \rightarrow^* vAw \rightarrow^* x]$), then

$$I_0^0 = \{(0, [A \rightarrow \alpha \cdot \beta], 0) \mid A \rightarrow \alpha\beta\}.$$

By applying the rest of Algorithm **Earley** for computation of columns 1, 2, ... after the initialization of column 0 with I_0^0 , we can decide whether the input $a_1 \cdots a_n$ is a suffix of a complete sentence. This can be explained as follows. For a complete sentence $va_1 \cdots a_n$ we have that $([S \rightarrow \cdot \sigma], va_1 \cdots a_n \#) \vdash^* (\delta, a_1 \cdots a_n \#) \vdash^* ([S \rightarrow \sigma \cdot], \#)$, some δ . By initializing U to be I_0^0 we have a finite representation for all possible stacks δ that can be in some configuration of the PDA after reading some prefix v . Earley's algorithm can now simulate computations of $(\delta, a_1 \cdots a_n \#) \vdash^* ([S \rightarrow \sigma \cdot], \#)$, without having to consider an individual δ .

After changing the initialization, Earley's algorithm is given by the following:

Algorithm: Earley-Suf (Earley's recognizer adapted to suffix recognition)

Consider a context-free grammar $G = (T, N, P, S)$. Without loss of generality, assume that there is only one rule of the form $S \rightarrow \sigma$. Assume the input is $a_1 \cdots a_n$. Let $a_{n+1} = \#$. Initialize table U to be I_0^0 . Perform one of the three

	0	1	2	3	4	5	6	7
0	$[S \rightarrow \bullet A]$ $[A \rightarrow \bullet bAb]$	$[A \rightarrow b\bullet Ab]$					$[A \rightarrow bAb\bullet]$	$[S \rightarrow A\bullet]$ $[A \rightarrow bAb\bullet]$
1		$[A \rightarrow \bullet aAa]$	$[A \rightarrow a\bullet Aa]$			$[A \rightarrow aA\bullet a]$	$[A \rightarrow aAa\bullet]$	
2			$[A \rightarrow \bullet bAb]$	$[A \rightarrow b\bullet Ab]$	$[A \rightarrow bAb\bullet]$	$[A \rightarrow bAb\bullet]$		
3				$[A \rightarrow \bullet c]$	$[A \rightarrow c\bullet]$			
4								
5								
6								
7								

FIG. 2. The set U produced by Algorithm **Earley**, with input $babcbab$.

	0	1	2	3	4	5	6
0	I_0^0 $[A \rightarrow \bullet aAa]$ $[A \rightarrow \bullet aAa\bullet]$ $[A \rightarrow \bullet aA\bullet a]$ $[A \rightarrow \bullet bAb]$ $[S \rightarrow \bullet A]$	$[A \rightarrow a\bullet Aa]$ $[A \rightarrow aA\bullet a]$ $[A \rightarrow aA\bullet a]$ $[A \rightarrow bAb\bullet]$ $[S \rightarrow A\bullet]$	$[A \rightarrow b\bullet Ab]$ $[A \rightarrow bAb\bullet]$ $[S \rightarrow A\bullet]$		$[A \rightarrow aA\bullet a]$ $[A \rightarrow aA\bullet a]$	$[A \rightarrow aA\bullet a]$ $[A \rightarrow aA\bullet a]$ $[A \rightarrow bAb\bullet]$ $[S \rightarrow A\bullet]$	$[A \rightarrow bAb\bullet]$ $[A \rightarrow bAb\bullet]$ $[S \rightarrow A\bullet]$
1		$[A \rightarrow \bullet bAb]$	$[A \rightarrow b\bullet Ab]$	$[A \rightarrow bAb\bullet]$	$[A \rightarrow bAb\bullet]$		
2			$[A \rightarrow \bullet c]$	$[A \rightarrow c\bullet]$			
3							
4							
5							
6							

FIG. 3. The set U produced by Algorithm **Earley-Suf**, with input $abcbab$.

steps predictor, scanner or completer, as long as one of them is applicable. (This is only useful for $0 < i \leq n$, because only then new items may be added.)

The input is recognized as suffix if $(0, [S \rightarrow \sigma \bullet], n) \in U$.

Note that an item $(0, [A \rightarrow \alpha \bullet \beta], i)$ computed by this algorithm represents the fact that $\alpha \rightarrow^* va_1 \cdots a_i$ for some v , and not $\alpha \rightarrow^* a_1 \cdots a_i$ as was the case for Algorithm **Earley**.

Example 3.2. Consider again the grammar and input from Example 3.1. The set U produced by Algorithm **Earley** is given in Figure 2. An item $(j, [A \rightarrow \alpha \bullet \beta], i) \in U$ is indicated by an occurrence of $[A \rightarrow \alpha \bullet \beta]$ in the i th column and j th row. Note that each item in U corresponds to some configuration in Figure 1.

If we apply Algorithm **Earley-Suf** to the suffix $abcbab$, then the set U given in Figure 3 will be produced. Most of the items in Figure 3 correspond with items in Figure 2. Of the remaining items (those marked with an asterisk) the ones in columns 1 and 2 represent the idea that the algorithm does not have enough information at that point to decide whether it is processing the part of a palindrome before or after the center marker c ; the ones in columns 5 and 6 represent the idea that the algorithm must consider all strings which may make a prefix of some palindrome if they are put before the actual input up to the current input position.

It is obvious that Algorithm **Earley** has a linear time complexity if the grammar is LL(1). This is because it directly simulates the behavior of the PDA resulting from Construction **LL-PDA**, which is (by definition) deterministic for an LL(1) grammar, and deterministic PDAs process the input in linear time (provided they terminate, which they do in this case).

We can however prove that also Algorithm **Earley-Suf** has a linear time complexity if the grammar is LL(1). The central observation is this:

LEMMA 3.3. *Let U be computed using Algorithm **Earley-Suf**, for an LL(1) grammar and certain input. There can be at most one item $(j, [A \rightarrow \alpha \cdot \beta], i) \in U$ for each $j > 0$ and $[A \rightarrow \alpha \cdot \beta]$.*

PROOF. We define the relation \angle between nonterminals as: $B \angle A$ if and only if $A \rightarrow \alpha B \beta$ for some α and β satisfying $\alpha \rightarrow^* \epsilon$. It is well-known that \angle^+ , the transitive closure of this relation, is irreflexive for LL(1) grammars. A consequence is that for any nonempty subset N' of the nonterminals from an LL(1) grammar there is some $A \in N'$ that satisfies $\neg \exists B \in N' [B \angle A]$. We call such a nonterminal A in a set N' of nonterminals \angle -minimal.

Suppose that for an LL(1) grammar and certain input the resulting table U is such that there are pairs $(j, [A \rightarrow \alpha \cdot \beta], i_1), (j, [A \rightarrow \alpha \cdot \beta], i_2) \in U$ with $i_1 \neq i_2$. Consider the set of such pairs for which j is maximal. Reduce this set to those for which A is \angle -minimal. Of the resulting set, choose a pair for which the length of α , denoted $|\alpha|$, is minimal.

Stated more succinctly, we consider a pair $(j, [A \rightarrow \alpha \cdot \beta], i_1), (j, [A \rightarrow \alpha \cdot \beta], i_2) \in U$ with $i_1 \neq i_2$, such that first j is maximal, then A is \angle -minimal, and then $|\alpha|$ is minimal.

We must have that α is of the form $\alpha' B$ where $(j, [A \rightarrow \alpha' \cdot B \beta], h) \in U$ for some h such that $(h, [B \rightarrow \gamma_1 \cdot], i_1), (h, [B \rightarrow \gamma_2 \cdot], i_2) \in U$, for some γ_1 and γ_2 . (Note that our assumption of minimality of $|\alpha|$, after maximality of j and \angle -minimality of A , contradicts that α is of the form $\alpha' a$, because then $(j, [A \rightarrow \alpha' \cdot a \beta], i_1 - 1), (j, [A \rightarrow \alpha' \cdot a \beta], i_2 - 1) \in U$, and $|\alpha'| < |\alpha|$.)

The possibility $\gamma_1 = \gamma_2$ is contradicted by the assumption that the pair $(j, [A \rightarrow \alpha \cdot \beta], i_1), (j, [A \rightarrow \alpha \cdot \beta], i_2) \in U$ has been chosen with first j is maximal and then A is \angle -minimal. This is because $\gamma_1 = \gamma_2$ and the maximality of j imply that $h = j$, which means that $\alpha' \rightarrow^* \epsilon$, and thereby $B \angle A$. This is in contradiction with subsequent \angle -minimality of A .

On the other hand, the possibility $\gamma_1 \neq \gamma_2$ together with $(h, [B \rightarrow \cdot \gamma_1], h), (h, [B \rightarrow \cdot \gamma_2], h) \in U$ is contradicted by the fact that the grammar is LL(1).

This proves that our initial assumption is false. \square

Let $|G|$ denote the size of grammar G , measured in the number of stack symbols $[A \rightarrow \alpha \cdot \beta]$. We can now prove

LEMMA 3.4. *Algorithm **Earley-Suf** has a linear time complexity for LL(1) grammars, measured in the length of the input.*

PROOF. Let the input be $a_1 \cdots a_n$. We investigate how many times the three steps can be performed.

predictor. There are $\mathcal{O}(|G| \cdot n)$ items of the form $(0, [A \rightarrow \alpha \cdot B \beta], i) \in U$ for $i > 0$, and, because of Lemma 3.3, $\mathcal{O}(|G| \cdot n)$ items of the form $(j, [A \rightarrow \alpha \cdot B \beta], i) \in U$ for $j > 0$. Each of these items gives rise to at most one item $(i, [B \rightarrow \cdot \gamma], i) \in U$ because the grammar is LL(1). Therefore, the predictor step is applied $\mathcal{O}(|G| \cdot n)$ times.

scanner. The scanner step is applied $\mathcal{O}(|G| \cdot n)$ times following a similar reasoning.

completer. We distinguish between two cases.

- There are $\mathcal{O}(|G| \cdot n)$ items of the form $(j, [A \rightarrow \alpha \cdot B\beta], i) \in U$, for $i > 0$. For each $i > 0$ and B there is only one item $(i, [B \rightarrow \gamma \cdot], h) \in U$, for some γ , because of Lemma 3.3 and because there can be only one item $(i, [B \rightarrow \cdot \gamma], i) \in U$ since the grammar is LL(1). (The lookahead symbol a_{i+1} determines which rule with left-hand side B is applied.) For this case, the completer step is therefore applied $\mathcal{O}(|G| \cdot n)$ times.
- There are $\mathcal{O}(|G|)$ items of the form $(0, [A \rightarrow \alpha \cdot B\beta], 0) \in U$. There are $\mathcal{O}(|G| \cdot n)$ items of the form $(0, [B \rightarrow \gamma \cdot], h) \in U$. For this case, the completer step is therefore applied $\mathcal{O}(|G|^2 \cdot n)$ times.

Together this yields $\mathcal{O}(|G|^2 \cdot n)$ steps. \square

We conclude:

THEOREM 3.5. *Suffix recognition can be performed in linear time for all LL(1) languages.*

The time complexity $\mathcal{O}(|G|^2 \cdot n)$ we computed in the proof of Lemma 3.4 can be improved to $\mathcal{O}(|G| \cdot n)$, by applying a trivial optimization. For details see Section 1.2.9 of Nederhof [1994].

4. Suffix Recognition for Deterministic Languages

In this section, we show that tabular suffix recognizers can be constructed not just for the LL(1) languages, but for all deterministic languages. We will only consider PDAs with restricted types of transition, in order to simplify the discussion. This restriction requires that all transitions are of the form $X \xrightarrow{z} XY$, where $z = \epsilon$ or $z = a$, or of the form $XY \xrightarrow{\epsilon} Z$. It is important to realize that the restriction does not affect the descriptive power of PDAs, that is, all context-free languages can be described by PDAs that use only the restricted types of transition. Again without loss of generality, we assume that there are no transitions of the form $X_{fin} \xrightarrow{z} X_{fin}Y$.

We define a subrelation \models^+ of \vdash^+ as: $(\delta, vw) \models^+ (\delta\delta', w)$ if and only if $(\delta, vw) = (\delta, z_1z_2 \cdots z_mw) \vdash (\delta\delta_1, z_2 \cdots z_mw) \vdash \cdots \vdash (\delta\delta_m, w) = (\delta\delta', w)$, for some $m \geq 1$, where $|\delta_k| > 0$ for all k , $1 \leq k \leq m$. Informally, we have $(\delta, vw) \models^+ (\delta', w)$ if configuration (δ', w) can be reached from (δ, vw) without the bottom-most part δ of the intermediate stacks being affected by any of the transitions; furthermore, at least one element is pushed on top of δ . Note that $(\delta_1X, vw) \models^+ (\delta_1X\delta', w)$ implies $(\delta_2X, vw') \models^+ (\delta_2X\delta', w')$ for any δ_2 and any w' , since the transitions do not address the part of the stack below X , nor read the input following v .

Below we reformulate an algorithm from Lang [1974] and Billot and Lang [1989], which accepts a language accepted by a PDA, using a parsing table U . This algorithm generalizes Algorithm **Earley**. For technical reasons, we have to assume that the stack always consists of at least two elements. This is accomplished by assuming that a fresh stack symbol \perp occurs below the bottom of the actual stack, and by assuming that the actual initial configuration is created by an imaginary extra step $(\perp, v\#) \vdash (\perp X_{init}, v\#)$.

Algorithm: **Dyna** (Dynamic programming)

Consider a fixed PDA. Assume the input is $a_1 \cdots a_n$. Let $a_{n+1} = \#$. Let the set U be $\{(\perp, 0, X_{init}, 0)\}$. Perform one of the following two steps as long as one of them is applicable.

- push** (1) Choose a pair, not considered before, consisting of a transition $X \xrightarrow{z} XY$ and an input position j , such that there is an item $(W, h, X, j) \in U$, for some W and h , and such that $z = \epsilon \vee z = a_{j+1}$.
 (2) If $z = \epsilon$, then let $i = j$, else let $i = j + 1$.
 (3) Add item (X, j, Y, i) to U if it is not already there.
- pop** (1) Choose a triple, not considered before, consisting of a transition $XY \xrightarrow{\epsilon} Z$ and items $(W, h, X, j), (X, j, Y, i) \in U$.
 (2) Add item (W, h, Z, i) to U if it is not already there.

The input is recognized if $(\perp, 0, X_{fin}, n + 1) \in U$.

It can be proved that Algorithm **Dyna** eventually adds an item (X, j, Y, i) to U if and only if

- (1) $(\perp, a_1 \cdots a_j) \vdash^* (\delta X, \epsilon)$, for some δ , and
- (2) $(X, a_{j+1} \cdots a_i) \models^+ (XY, \epsilon)$.

The first condition states that some configuration can be reached from the initial configuration by reading the input up to position j , and in this configuration, an element labeled X is on top of the stack.

The second condition states that, if a stack has an element labeled X on top, then the pushdown automaton can, by reading the input between j and i and without ever popping X , obtain a stack with one more element, labeled Y , which is on top of X .

That eventually $(\perp, 0, X_{fin}, n + 1) \in U$ is equivalent to $(\perp, a_1 \cdots a_n \#) \vdash (\perp X_{init}, a_1 \cdots a_n \#) \vdash^* (\perp X_{fin}, \epsilon)$, which proves the correctness of Algorithm **Dyna**.

We can also apply the above algorithm for suffix recognition. This requires a different initialization of the first column of the table (cf. Algorithms **Earley** and **Earley-Suf**). For a set of items I , we define the set $\text{closure}(I)$ as the smallest set satisfying:

- $I \subseteq \text{closure}(I)$.
- For each item $(W, h, X, j) \in \text{closure}(I)$ and transition $X \xrightarrow{z} XY$ we have $(X, j, Y, j) \in \text{closure}(I)$.
- For each pair of items $(W, h, X, j), (X, j, Y, i) \in \text{closure}(I)$ and transition $XY \xrightarrow{\epsilon} Z$ we have $(W, h, Z, i) \in \text{closure}(I)$.

The intuition behind closure is that it simulates the PDA without taking notice of any input. If $I = \{(\perp, 0, X_{init}, 0)\}$, then $\text{closure}(I)$ represents in a finite way all stacks that can be reached by the PDA from an initial configuration by reading some input. By comparing the definition of closure with the definition of I_0^0 for Earley's algorithm, one may see that $\text{closure}(\{(\perp, 0, X_{init}, 0)\})$ is a generalization of I_0^0 .

We have the following algorithm, which is a special case of a more general algorithm defined in Lang [1988].

Algorithm: **Dyna-Suf** (Dynamic programming, adapted to suffix recognition)

Consider a fixed PDA. Assume the input is $a_1 \cdots a_n$. Let $a_{n+1} = \#$. Let the set U be $\text{closure}(\{(\perp, 0, X_{\text{init}}, 0)\})$. Perform one of the pushing or popping steps (as in Algorithm **Dyna**) as long as one of them is applicable. (The steps are only useful for $i > 0$.)

The input is recognized as suffix if $(\perp, 0, X_{\text{fin}}, n + 1) \in U$.

This algorithm eventually adds an item (X, j, Y, i) , $j > 0$, to U if and only if

- (1) $(\perp, va_1 \cdots a_j) \vdash^* (\delta X, \epsilon)$, for some v and δ , and
- (2) $(X, a_{j+1} \cdots a_i) \models^+ (XY, \epsilon)$

and an item $(X, 0, Y, i)$ to U if and only if

- (1) $(\perp, v) \vdash^* (\delta X, \epsilon)$, for some v and δ , and
- (2) $(X, wa_1 \cdots a_i) \models^+ (XY, \epsilon)$, for some w .

Algorithm **Dyna** has a linear time complexity for deterministic PDAs, for the same reason that Algorithm **Earley** has a linear time complexity for LL(1) grammars. However, for suffix recognition such a generalization is not immediately possible; that is, although Algorithm **Earley-Suf** has a linear time complexity for LL(1) grammars, Algorithm **Dyna-Suf**, which is its generalization, can have a quadratic time complexity for deterministic PDAs. This fact can be explained as follows:

We have seen that the central observation needed to prove that Algorithm **Earley-Suf** has a linear complexity is that there is at most one item $(j, [A \rightarrow \alpha \cdot \beta], i) \in U$ for each $j > 0$ and $[A \rightarrow \alpha \cdot \beta]$ (Lemma 3.3). No similar fact holds for Algorithm **Dyna-Suf**: depending on the nature of the PDA, it is possible that items $(X, j, Y, i) \in U$ are added for several i , with fixed X, j and Y . This may happen if $(\perp, va_1 \cdots a_j v_1 v_2 \cdots v_m) \vdash^* (\delta X, v_1 v_2 \cdots v_m) \models^+ (\delta XY, v_2 \cdots v_m)$ and $(Y, v_2 \cdots v_m) \vdash^+ (Y, v_3 \cdots v_m) \vdash^+ \cdots \vdash^+ (Y, v_m)$. Such a situation can in the most trivial case be caused by a pair of transitions $X \xrightarrow{\delta} XY$ and $XY \xrightarrow{\epsilon} X$; the general case is more complex however.

The following definition identifies the problem with obtaining a linear complexity. We define a PDA to be *loop-free* if $(X, v) \vdash^+ (X, \epsilon)$ does not hold for any X and v . The intuition is that reading some input must be reflected by a change in the stack.

Our solution to linear-time suffix recognition for deterministic PDAs that are not loop-free is the following: we define a transformation from one deterministic PDA to another that accepts the same language and is loop-free. Intuitively, this is done by pushing extra elements \bar{X} on the stack so that we have $(X, v) \vdash^+ (\bar{X} X, \epsilon)$ instead of $(X, v) \vdash^+ (X, \epsilon)$, where \bar{X} is a special stack symbol to be defined shortly.

As a first step we remark that for a deterministic PDA we can divide the stack symbols into two sets *PUSH* and *POP*, defined by

$$\text{PUSH} = \{X \mid \text{there is a transition } X \xrightarrow{\delta} XY\}$$

$$\text{POP} = \{Y \mid \text{there is a transition } XY \xrightarrow{\epsilon} Z\} \cup \{X_{\text{fin}}\}.$$

It is straightforward to see that determinism of the PDA requires that *PUSH* and *POP* are disjoint. We may further assume that each stack symbol belongs to

either *PUSH* or *POP*, provided we assume that the PDA is *reduced*, meaning that there are no transitions which are useless for obtaining the final configuration from an initial configuration.⁴

Construction: τ (Transformation to loop-free PDAs)

Consider a deterministic PDA \mathcal{A} of which the stack symbols are partitioned into *PUSH* and *POP*, as explained above. From this PDA, a new PDA $\tau(\mathcal{A})$ is constructed, of which the stack symbols are those of \mathcal{A} plus stack symbols of the form X , with $X \in \text{PUSH}$. The transitions of $\tau(\mathcal{A})$ are given by

$$\begin{array}{lll} XY \xrightarrow{\epsilon}_{\tau(\mathcal{A})} Z & \text{for } XY \xrightarrow{\epsilon}_{\mathcal{A}} Z \text{ with } Z \in \text{POP} \\ XY \xrightarrow{\epsilon}_{\tau(\mathcal{A})} \bar{Z} & \text{for } XY \xrightarrow{\epsilon}_{\mathcal{A}} Z \text{ with } Z \in \text{PUSH} \\ X \xrightarrow{\epsilon}_{\tau(\mathcal{A})} XX & \text{for } X \in \text{PUSH} \\ XY \xrightarrow{\epsilon}_{\tau(\mathcal{A})} Y & \text{for } X \in \text{PUSH}, Y \in \text{POP} \\ X \xrightarrow{z}_{\tau(\mathcal{A})} XY & \text{for } X \xrightarrow{z}_{\mathcal{A}} XY \end{array}$$

The special stack symbols X_{init} and X_{fin} for $\tau(\mathcal{A})$ are the same as those for \mathcal{A} .

Example 4.1. Let T be a set of terminals. For a string $v \in T^*$ and a terminal $a \in T$, let v_a denote the number of occurrences of a in v . Consider the language

$$L = \{vw \mid v \in L_1 \wedge w \in L_2\}$$

where

$$L_1 = \{v \in \{a, b\}^+ \mid v_a = v_b \wedge$$

for all nonempty proper prefixes v' of v

we have $v'_a > v'_b\}$

$$L_2 = \{b\}^*$$

A deterministic PDA \mathcal{A} accepting this language L is given in Figure 4, together with the transformed PDA $\tau(\mathcal{A})$. For \mathcal{A} we have

$$\text{PUSH} = \{X_{init}, a, X_{inter}, d\}$$

$$\text{POP} = \{b, c, e, \#, X_{fin}\}.$$

Consider the string $aabbbb \in L$; note that $aabb \in L_1$ and $bb \in L_2$. Figure 5 demonstrates how \mathcal{A} and $\tau(\mathcal{A})$ recognize this string.

We now set out to prove that τ has the required properties.

LEMMA 4.2. *If \mathcal{A} is a deterministic PDA, then $\tau(\mathcal{A})$ is deterministic.*

PROOF. This can be proved easily by comparing the transitions in \mathcal{A} and $\tau(\mathcal{A})$. Note that for each symbol \bar{X} on top of the stack exactly one pushing transition

⁴ Note that each PDA may be turned into a reduced PDA accepting the same language by just omitting the useless transitions.

\mathcal{A}			$\tau(\mathcal{A})$	
X_{init}	\xrightarrow{a}	$X_{init} a$	ditto	
a	\xrightarrow{a}	$a a$	ditto	
a	\xrightarrow{b}	$a b$	ditto	
$a b$	\xrightarrow{c}	c	ditto	
$a c$	\xrightarrow{c}	a	$a c \xrightarrow{c} \bar{a}$	
$X_{init} c$	\xrightarrow{c}	X_{inter}	$X_{init} c \xrightarrow{c} \bar{X}_{inter}$	
X_{inter}	\xrightarrow{c}	$X_{inter} d$	ditto	
d	\xrightarrow{b}	$d e$	ditto	
$d e$	\xrightarrow{c}	d	$d e \xrightarrow{c} \bar{d}$	
d	$\xrightarrow{\#}$	$d \#$	ditto	
$d \#$	\xrightarrow{c}	$\#$	ditto	
$X_{inter} \#$	\xrightarrow{c}	X_{fin}	ditto	
			$\bar{X} \xrightarrow{c} \bar{X}X$ for all $X \in PUSH$	
			$\bar{X}Y \xrightarrow{c} Y$ for all $X \in PUSH, Y \in POP$	

FIG. 4. The transformation τ applied to a deterministic PDA \mathcal{A} . For $\tau(\mathcal{A})$ only those transitions are given which differ from corresponding transitions of \mathcal{A} .

may be applied, while for each pair of symbols $\bar{X}Y$ on top of the stack, with $Y \in POP$, exactly one popping transition may be applied. \square

LEMMA 4.3. *If \mathcal{A} is a deterministic PDA, then \mathcal{A} and $\tau(\mathcal{A})$ accept the same language.*

PROOF. We first prove that for all stack symbols X_1, \dots, X_m from \mathcal{A} we have $(X_{init}, v) \vdash_{\mathcal{A}}^* (X_1 \cdots X_m, \epsilon)$ if and only if $(X_{init}, v) \vdash_{\tau(\mathcal{A})}^* (\alpha_1 X_1 \alpha_2 \cdots \alpha_m X_m, \epsilon)$ where for all i , $1 \leq i \leq m$, we have $\alpha_i = \bar{Y}_{i,1} \bar{Y}_{i,2} \cdots \bar{Y}_{i,r_i}$ for some $r_i \geq 0$.

“only if”. The proof is given by induction on the number of steps used in $(X_{init}, v) \vdash_{\mathcal{A}}^* (X_1 \cdots X_m, \epsilon)$.

- (1) If zero steps are involved, then we have $(X_{init}, \epsilon) \vdash_{\mathcal{A}}^* (X_{init}, \epsilon)$. By definition, we have also $(X_{init}, \epsilon) \vdash_{\tau(\mathcal{A})}^* (X_{init}, \epsilon)$.
- (2) Suppose that the last step is a push, then we have $(X_{init}, v) \vdash_{\mathcal{A}}^* (X_1 \cdots X_{m-1}, z) \vdash_{\mathcal{A}} (X_1 \cdots X_m, \epsilon)$, where the last transition used is $X_{m-1} \xrightarrow{z} X_{m-1} X_m$. The induction hypothesis informs us that $(X_{init}, v) \vdash_{\tau(\mathcal{A})}^* (\alpha_1 X_1 \alpha_2 \cdots \alpha_{m-1} X_{m-1}, z)$. Because also $X_{m-1} \xrightarrow{z} X_{m-1} X_m$ we have $(X_{init}, v) \vdash_{\tau(\mathcal{A})}^* (\alpha_1 X_1 \alpha_2 \cdots \alpha_{m-1} X_{m-1} X_m, \epsilon)$.
- (3) Suppose that the last step is a pop, then we have $(X_{init}, v) \vdash_{\mathcal{A}}^* (X_1 \cdots X_{m-1} X'_m X_{m+1}, \epsilon) \vdash_{\mathcal{A}} (X_1 \cdots X_{m-1} X_m, \epsilon)$, where the last transition used is $X'_m X_{m+1} \xrightarrow{c} X_m$. The induction hypothesis informs us that $(X_{init}, v) \vdash_{\tau(\mathcal{A})}^* (\alpha_1 X_1 \alpha_2 \cdots \alpha_m X'_m \alpha_{m+1} X_{m+1}, \epsilon)$, with $\alpha_{m+1} = \bar{Y}_1 \cdots \bar{Y}_r$, for some $r > 0$. We first have $(\alpha_1 X_1 \alpha_2 \cdots \alpha_m X'_m \bar{Y}_1 \cdots \bar{Y}_r X_{m+1}, \epsilon) \vdash_{\tau(\mathcal{A})} (\alpha_1 X_1 \alpha_2 \cdots \alpha_m X'_m \bar{Y}_1 \cdots \bar{Y}_{r-1} X_{m+1}, \epsilon) \vdash_{\tau(\mathcal{A})} (\alpha_1 X_1 \alpha_2 \cdots \alpha_m X'_m X_{m+1}, \epsilon)$, using the transitions $\bar{Y}_j X_{m+1} \xrightarrow{c} X_{m+1}, 1 \leq j \leq r$, which exist since $X_{m+1} \in POP$. Subsequently, there are two possibilities:

\mathcal{A}		$\tau(\mathcal{A})$	
X_{init}	aabbbb#	X_{init}	aabbbb#
$X_{init} a$	abbbb#	$X_{init} a$	abbbb#
$X_{init} a a$	bbbb#	$X_{init} a a$	bbbb#
$X_{init} a a b$	bbb#	$X_{init} a a b$	bbb#
$X_{init} a c$	bbb#	$X_{init} a c$	bbb#
$X_{init} a$	bbb#	$X_{init} \bar{a}$	bbb#
		$X_{init} \bar{a} a$	bbb#
$X_{init} a b$	bb#	$X_{init} \bar{a} a b$	bb#
$X_{init} c$	bb#	$X_{init} \bar{a} c$	bb#
		$X_{init} c$	bb#
X_{inter}	bb#	$\overline{X_{inter}}$	bb#
		$\overline{X_{inter}} X_{inter}$	bb#
$X_{inter} d$	bb#	$\overline{X_{inter}} X_{inter} d$	bb#
$X_{inter} d e$	b#	$\overline{X_{inter}} X_{inter} d e$	b#
$X_{inter} d$	b#	$\overline{X_{inter}} X_{inter} \bar{d}$	b#
		$\overline{X_{inter}} X_{inter} \bar{d} d$	b#
$X_{inter} d e$	#	$\overline{X_{inter}} X_{inter} \bar{d} d e$	#
$X_{inter} d$	#	$\overline{X_{inter}} X_{inter} \bar{d} \bar{d}$	#
		$\overline{X_{inter}} X_{inter} \bar{d} \bar{d} d$	#
$X_{inter} d \#$	ϵ	$\overline{X_{inter}} X_{inter} \bar{d} \bar{d} d \#$	ϵ
$X_{inter} \#$	ϵ	$\overline{X_{inter}} X_{inter} \bar{d} \bar{d} \#$	ϵ
		$\overline{X_{inter}} X_{inter} \bar{d} \#$	ϵ
		$\overline{X_{inter}} X_{inter} \#$	ϵ
X_{fin}	ϵ	$\overline{X_{inter}} X_{fin}$	ϵ
		X_{fin}	ϵ

FIG. 5. The sequences of configurations for input aabbbb, using \mathcal{A} and $\tau(\mathcal{A})$.

- If $X_m \in PUSH$ then $X'_m X_{m+1} \xrightarrow{\epsilon}_{\tau(\mathcal{A})} \overline{X_m}$ and $(\alpha_1 X_1 \alpha_2 \cdots \alpha_m X'_m X_{m+1}, \epsilon) \vdash_{\tau(\mathcal{A})}^* (\alpha_1 X_1 \alpha_2 \cdots \alpha_m \overline{X_m}, \epsilon) \vdash_{\tau(\mathcal{A})}^* (\alpha_1 X_1 \alpha_2 \cdots \alpha_m \overline{X_m} X_m, \epsilon)$. In the last configuration, $\alpha_m \overline{X_m}$ is a sequence of “barred” symbols as desired.
- If $X_m \in POP$, then $X'_m X_{m+1} \xrightarrow{\epsilon}_{\tau(\mathcal{A})} X_m$ and $(\alpha_1 X_1 \alpha_2 \cdots \alpha_m X'_m X_{m+1}, \epsilon) \vdash_{\tau(\mathcal{A})}^* (\alpha_1 X_1 \alpha_2 \cdots \alpha_m X_m, \epsilon)$.

“if”. Analogously to the “only if” part, the proof is given by induction on the number of steps used in $(X_{init}, v) \vdash_{\tau(\mathcal{A})}^* (\alpha_1 X_1 \alpha_2 \cdots \alpha_m X_m, \epsilon)$. Below we only give the most interesting cases.

- (1) Suppose the last transition used was $\overline{X_m} \xrightarrow{\epsilon}_{\tau(\mathcal{A})} \overline{X_m} X_m$, then we have $(X_{init}, v) \vdash_{\tau(\mathcal{A})}^* (\alpha_1 X_1 \alpha_2 \cdots X_{m-1} \alpha'_m XY, \epsilon) \vdash_{\tau(\mathcal{A})}^* (\alpha_1 X_1 \alpha_2 \cdots X_{m-1} \alpha'_m \overline{X_m}, \epsilon) \vdash_{\tau(\mathcal{A})}^* (\alpha_1 X_1 \alpha_2 \cdots X_{m-1} \alpha'_m \overline{X_m} X_m, \epsilon)$, where $\alpha_m = \alpha'_m \overline{X_m}$, and the second-last transition used was $XY \xrightarrow{\epsilon}_{\tau(\mathcal{A})} \overline{X_m}$ for some X and Y . The induction hypothesis informs us that $(X_{init}, v) \vdash_{\mathcal{A}}^* (X_1 \cdots X_{m-1} XY, \epsilon)$. From $XY \xrightarrow{\epsilon}_{\tau(\mathcal{A})} \overline{X_m}$, we conclude the existence of $XY \xrightarrow{\epsilon}_{\mathcal{A}} X_m$. Therefore, $(X_1 \cdots X_{m-1} XY, \epsilon) \vdash_{\mathcal{A}}^* (X_1 \cdots X_{m-1} X_m, \epsilon)$.
- (2) Suppose the last transition used was $\overline{X} X_m \xrightarrow{\epsilon}_{\tau(\mathcal{A})} X_m$, then we have $(X_{init}, v) \vdash_{\tau(\mathcal{A})}^* (\alpha_1 X_1 \alpha_2 \cdots \alpha_m \overline{X} X_m, \epsilon) \vdash_{\tau(\mathcal{A})}^* (\alpha_1 X_1 \alpha_2 \cdots \alpha_m X_m, \epsilon)$. Since $\alpha_m \overline{X_m}$ is a sequence of barred symbols, the induction hypothesis informs us that $(X_{init}, v) \vdash_{\mathcal{A}}^* (X_1 \cdots X_m, \epsilon)$.

From the “if” part we conclude that $(X_{init}, v) \vdash_{\tau(\mathcal{A})}^* (X_{fin}, \epsilon)$ implies $(X_{init}, v) \vdash_{\mathcal{A}}^* (X_{fin}, \epsilon)$. From the “only if” part we conclude that $(X_{init}, v) \vdash_{\mathcal{A}}^* (X_{fin}, \epsilon)$ implies $(X_{init}, v) \vdash_{\tau(\mathcal{A})}^* (\alpha X_{fin}, \epsilon) \vdash_{\tau(\mathcal{A})}^* (X_{fin}, \epsilon)$, for some $\alpha = \bar{Y}_1 \cdots \bar{Y}_r$, using the transitions $\bar{Y}_j X_{fin} \xrightarrow{\epsilon}_{\tau(\mathcal{A})} X_{fin}$, $1 \leq j \leq r$, which exist since $X_{fin} \in POP$.

This proves the equivalence of the two accepted languages. \square

LEMMA 4.4. *If \mathcal{A} is a deterministic PDA, then $\tau(\mathcal{A})$ is loop-free.*

PROOF. Consider the set of stack symbols of $\tau(\mathcal{A})$. We define an ordering $<$ on these symbols as the least ordering satisfying:

$$\begin{aligned} X < \bar{Y} & \quad \text{for all} \quad X \in POP, Y \in PUSH \\ \bar{Y} < Z & \quad \text{for all} \quad Y, Z \in PUSH \\ X < Z & \quad \text{for all} \quad X \in POP, Z \in PUSH. \end{aligned}$$

Note that this relation is transitive and irreflexive. Below we prove that if $(X, v) \vdash_{\tau(\mathcal{A})}^+ (Z, \epsilon)$ then $Z < X$. This is sufficient to prove that $\tau(\mathcal{A})$ is loop-free, since $<$ is irreflexive.

Consider $(X, v) \vdash_{\tau(\mathcal{A})}^+ (XY, \epsilon) \vdash_{\tau(\mathcal{A})}^+ (Z, \epsilon)$, using some transition $XY \xrightarrow{\epsilon}_{\tau(\mathcal{A})} Z$ for the last step. It is obvious that $X \notin POP$ since otherwise Y could not have been on top of X . There are two remaining cases:

- If $X \in PUSH$, then either $Z \in POP$ or Z is of the form \bar{Z}' , with $Z' \in PUSH$, according to the definition of τ . Therefore, in either case, $Z < X$.
- If X is of the form \bar{X}' , with $X' \in PUSH$, then the transition $XY \xrightarrow{\epsilon}_{\tau(\mathcal{A})} Z$ must be of the form $\bar{X}'Y \xrightarrow{\epsilon}_{\tau(\mathcal{A})} Y$, with $Y = Z \in POP$. Therefore, $Z < X$.

Since each sequence $(X, v) \vdash_{\tau(\mathcal{A})}^+ (Z, \epsilon)$ can be split up into smaller sequences (in this case at most two, leading from a symbol in $PUSH$ to a barred symbol and then to a symbol in POP) of the form $(X, v) \vdash_{\tau(\mathcal{A})}^+ (XY, \epsilon) \vdash_{\tau(\mathcal{A})}^+ (Z, \epsilon)$, and since $<$ is transitive, the required result follows. \square

From the above, we conclude:

COROLLARY 4.5. *Any deterministic language is accepted by some deterministic and loop-free PDA.*

We now return to the issue of the time complexity of tabular suffix recognition. We start with a minor result, which is a generalization of Lemma 3.3.

LEMMA 4.6. *Let U be computed using Algorithm Dyna-Suf, for a deterministic and loop-free PDA and certain input. There can be at most one item of the form $(X, j, Y, i) \in U$ for each X, Y and $j > 0$.*

PROOF. The existence of an item $(X, j, Y, i) \in U$ requires that $(X, a_{j+1} \cdots a_i) \models^+ (XY, \epsilon)$. In the case that the PDA is deterministic, the existence of two items $(X, j, Y, i_1), (X, j, Y, i_2) \in U$ (say $i_1 < i_2$) requires that $(X, a_{j+1} \cdots a_{i_1}) \models^+ (XY, \epsilon)$ and $(Y, a_{i_1+1} \cdots a_{i_2}) \vdash^+ (Y, \epsilon)$, because of the definition of \models^+ . However, $(Y, a_{i_1+1} \cdots a_{i_2}) \vdash^+ (Y, \epsilon)$ is not possible if the PDA is loop-free. \square

LEMMA 4.7. *For a deterministic and loop-free PDA \mathcal{A} , Algorithm **Dyna-Suf** has a linear time complexity, measured in the length of the input.*

PROOF. Let the input be $a_1 \cdots a_n$. Let $|\mathcal{A}|$ denote the number of stack symbols of PDA \mathcal{A} . We investigate how many steps are applied.

push Since the PDA is deterministic, there are $\mathcal{O}(|\mathcal{A}| \cdot n)$ combinations of a stack symbol X and an input position $i > 0$ such that there is a transition $X \xrightarrow{z} XY$ with $z = \epsilon \vee z = a_i$. Therefore, the pushing step is applied $\mathcal{O}(|\mathcal{A}| \cdot n)$ times.

pop We distinguish between two cases.

—There are $\mathcal{O}(|\mathcal{A}|^2 \cdot n)$ items of the form $(W, h, X, j) \in U$, for $j > 0$, because of Lemma 4.6. For each of these, there are $\mathcal{O}(|\mathcal{A}|)$ items of the form $(X, j, Y, i) \in U$, again because of Lemma 4.6. For this case, the popping step is therefore applied $\mathcal{O}(|\mathcal{A}|^3 \cdot n)$ times.

—There are $\mathcal{O}(|\mathcal{A}|^2)$ items of the form $(W, 0, X, 0) \in U$, and for each of these there are $\mathcal{O}(|\mathcal{A}| \cdot n)$ items of the form $(X, 0, Y, i) \in U$. For this case, the popping step is therefore applied $\mathcal{O}(|\mathcal{A}|^3 \cdot n)$ times.

Together, this yields $\mathcal{O}(|\mathcal{A}|^3 \cdot n)$ steps. \square

We can now prove the main result of this section.

THEOREM 4.8. *Suffix recognition can be performed in linear time for all deterministic languages.*

PROOF. This follows directly from Lemma 4.7 and Corollary 4.5. \square

It is important to realize that the time complexity we computed in the proof of Lemma 4.7 is rather pessimistic. First, Lemma 4.6 states that for deterministic and loop-free PDAs, Algorithm **Dyna-Suf** can add at most one item of the form (X, j, Y, i) to U for each X, Y and $j > 0$. If however the PDA results from transformation τ , then Algorithm **Dyna-Suf** can add at most three items of the form (X, j, Y, i) to U for each X and $j > 0$, because of the relation $<$ from Lemma 4.4, which does not allow any sequences $Y_1 < Y_2 < \cdots < Y_m$ with $m > 3$. This means that there are $\mathcal{O}(|\mathcal{A}| \cdot n)$ items of the form $(W, h, X, j) \in U$, for $h > 0$, and for each of these, there are $\mathcal{O}(1)$ items of the form $(X, j, Y, i) \in U$. Therefore there are only $\mathcal{O}(|\mathcal{A}| \cdot n)$ applications of popping transitions, for $h > 0$. For $h = 0$ it is important to note that the number of triples W, X, Y such that $(W, v) \models^+ (WX, \epsilon)$ and $(X, w) \models^+ (XY, \epsilon)$, for some v and w , is much less than $|\mathcal{A}|^3$ for typical PDAs.

Example 4.9. Consider the PDAs from Example 4.1. It is clear that \mathcal{A} is not loop-free: we have $(d, b) \vdash (de, \epsilon) \vdash (d, \epsilon)$.

Consider some input $v \in \{b\}^*$. Let $b_1 \cdots b_n = v$. For any prefix $b_1 \cdots b_j$ of v , we have $(\perp, wb_1 \cdots b_j) \vdash^* (X_{inter}, \epsilon)$, for some w : if $j \neq 0$, take $w \in \{a\}^+$ such that $|w| = j$ and if $j = 0$ take $w = ab$; then $wb_1 \cdots b_j \in L_1$.

We further have

$$\begin{array}{ll}
(X_{inter} & , \quad b_{j+1}b_{j+2}b_{j+3} \cdots b_i) \vdash \\
(X_{inter} d & , \quad b_{j+1}b_{j+2}b_{j+3} \cdots b_i) \vdash \\
(X_{inter} d e & , \quad b_{j+2}b_{j+3} \cdots b_i) \vdash \\
(X_{inter} d & , \quad b_{j+2}b_{j+3} \cdots b_i) \vdash \\
(X_{inter} d e & , \quad b_{j+3} \cdots b_i) \vdash \\
\vdots & \vdots \\
(X_{inter} d e & , \quad \epsilon) \vdash \\
(X_{inter} d & , \quad \epsilon)
\end{array}$$

or in other words, $(X_{inter}, b_{j+1} \cdots b_i) \models^+ (X_{inter} d, \epsilon)$, for all i such that $j \leq i \leq n$. Relying on the discussion we gave just after Algorithm **Dyna-Suf**, we conclude that Algorithm **Dyna-Suf** will add one item (X_{inter}, j, d, i) for each combination of j and i such that $0 \leq j \leq i \leq n$. In other words, Algorithm **Dyna-Suf** adds at least $\mathcal{O}(n^2)$ items to U , which demonstrates the quadratic behavior for PDAs that are not loop-free.

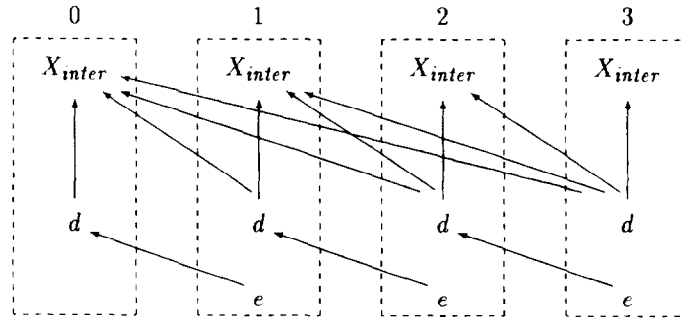
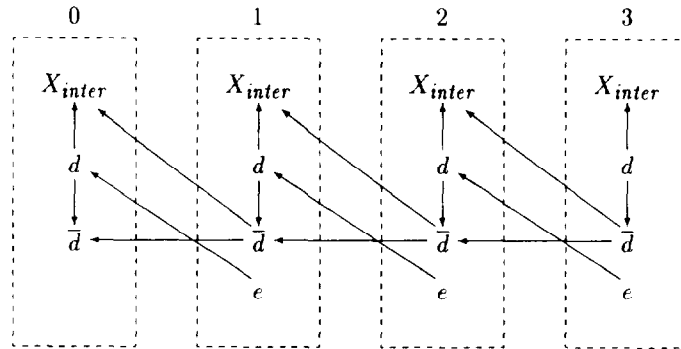
The transformed automaton $\tau(\mathcal{A})$ however is loop-free and therefore the quadratic behavior does not occur. We demonstrate this by giving the relevant items in U produced by Algorithm **Dyna-Suf**, both for \mathcal{A} and for $\tau(\mathcal{A})$. For presentational reasons, we give the set of items in a pictorial form, called a *graph-structured stack* [Tomita 1986; Nederhof 1994]. A graph-structured stack is a graph with $n + 2$ sets of nodes (assuming the input has length n as usual), each set containing one node for each stack symbol. An arrow from a node representing symbol Y in the i th set to a node representing symbol X in the j th set then represents an item $(X, j, Y, i) \in U$.

A selection of the arrows in the graph-structured stacks for input bbb is given in Figure 6. Note the quadratic number of arrows (items) in the case of \mathcal{A} , where $\tau(\mathcal{A})$ gives rise to only a linear number of arrows.

5. The Special Case of LL(1) Languages

At this point, it may be clarifying to investigate how the algorithms from Section 3 fit into the more general framework of Section 4. Two observations are in place. First, the special transitions of the form $X \xrightarrow{a} XY$ add nothing to the descriptive power of PDAs and can be encoded as transitions of the forms used in Section 4. Since a PDA for LL(1) recognition which then results is loop-free, the transformation τ is not needed in order to obtain a linear-time suffix recognition algorithm. Typically, PDAs realizing LR parsing or left-corner parsing for left-recursive grammars are *not* loop-free, and therefore require a solution similar to the application of τ in order to allow linear-time suffix recognition.

Second, in Section 3, we used 3-tuple items $(j, [A \rightarrow \alpha \cdot \beta], i)$, instead of the 4-tuple items $([B \rightarrow \gamma \cdot \delta], j, [A \rightarrow \alpha \cdot \beta], i)$ that one may expect from Section 4. The reason that the first component of such a 4-tuple can be omitted is that the PDAs resulting from Construction **LL-PDA** satisfy a special property

Graph-structured stack for automaton \mathcal{A} :Graph-structured stack for automaton $\tau(\mathcal{A})$:FIG. 6. Some of the relevant items for \mathcal{A} and for $\tau(\mathcal{A})$, given in graph-structured stacks. The input is bbb in both cases.

called *context-independence*. Most naturally occurring PDAs satisfy this property. For more details, see Nederhof [1994].

6. Producing Parse Trees

We have shown that suffix recognition can be done efficiently, especially for deterministic languages. The next step is to investigate how the recognition algorithms can be extended to be parsing algorithms. The approach to parsing in Lang [1974] and Billot and Lang [1989] is to start with pushdown transducers instead of with pushdown automata. A pushdown transducer can be seen as a PDA of which the transitions produce certain *output symbols* when they are applied. The *output string*, which is a list of all output symbols which are produced while successfully recognizing a sentence, is then seen as a representation of the parse.

We will use the notation $\delta_1 \xrightarrow{z|t} \delta_2$ to indicate a transition of some pushdown transducer, where t is ϵ or some output symbol b which is produced when the transition is applied, and where δ_1 , δ_2 , z are as before. Configurations are now triples consisting of a stack, a remaining input and the preliminary output string. We may define the new binary relation \vdash on these configurations as the least relation satisfying $(\delta\delta_1, zv, w) \vdash (\delta\delta_2, v, wt)$ if there is a transition $\delta_1 \xrightarrow{z|t} \delta_2$.

If such pushdown transducers are to be realized using a tabular algorithm such as Algorithm **Dyna** then we may apply the following to compute all output strings without deteriorating the time complexity of the recognition algorithm. The idea is that a context-free grammar, the *output grammar*, is constructed as a side-effect of recognition. For each item (X, j, Y, i) added to the table the grammar contains a nonterminal $A_{(X,j,Y,i)}$. This nonterminal is to generate all lists of output symbols w which the pushdown transducer produces while computing $(X, a_{j+1} \cdots a_i, \epsilon) \models^+ (XY, \epsilon, w)$. The rules of the output grammar are created when items are computed from others. For example, if we compute an item (W, h, Z, i) from two items $(W, h, X, j), (X, j, Y, i) \in U$, using a popping transition $XY \xrightarrow{\epsilon|b} Z$ which produces output symbol b , then the output grammar is extended with rule $A_{(W,h,Z,i)} \rightarrow A_{(W,h,X,j)} A_{(X,j,Y,i)} b$. The start symbol of the output grammar is $A_{(\perp, 0, X_{\text{fin}}, n+1)}$. The language generated by the output grammar consists of all output strings that may be produced by the pushdown transducer while successfully recognizing the input.

An output grammar is a particular representation of all parse trees for a certain input, and is therefore sometimes called a *parse forest*. For more details see Lang [1974; 1988], Billot and Lang [1989], and Nederhof [1994]. Some additional remarks concerning the form of parse forests in the context of incomplete input can be found in Rekers and Koorn [1991].

Transformation τ from Section 4 can be extended to work on deterministic pushdown transducers, in such a way that the output strings are not affected. A consequence is that the complexity results from the previous sections hold for suffix recognition as well as for suffix parsing.

For a pushdown transducer \mathcal{A} , the transitions of $\tau(\mathcal{A})$ are given by:

$$\begin{array}{lll}
 XY \xrightarrow{\epsilon|t}_{\tau(\mathcal{A})} Z & \text{for } XY \xrightarrow{\epsilon|t}_{\mathcal{A}} Z \text{ with } Z \in POP \\
 XY \xrightarrow{\epsilon|t}_{\tau(\mathcal{A})} \bar{Z} & \text{for } XY \xrightarrow{\epsilon|t}_{\mathcal{A}} Z \text{ with } Z \in PUSH \\
 \bar{X} \xrightarrow{\epsilon|\epsilon}_{\tau(\mathcal{A})} \bar{X}X & \text{for } X \in PUSH \\
 \bar{X}Y \xrightarrow{\epsilon|\epsilon}_{\tau(\mathcal{A})} Y & \text{for } X \in PUSH, Y \in POP \\
 X \xrightarrow{z|t}_{\tau(\mathcal{A})} XY & \text{for } X \xrightarrow{z|t}_{\mathcal{A}} XY.
 \end{array}$$

By straightforwardly adapting the proofs from Section 4, it can be shown that this extended transformation τ not only preserves the language accepted by an automaton, but also preserves the output string generated for each input that is recognized.

It is important to realize that the method described here for producing parse forests can be refined if a particular structured representation of all parses is required, or if parsing consists in evaluation of attributes in the case of attribute grammars. The method above is however the most straightforward way to yield all parses of a sentence without deteriorating the time complexities of the recognition algorithms.

7. Parsing of Incorrect Input

A natural property for PDAs is that they do not read past the first incorrect character of the input, which is called the *correct-prefix property*. Formally, a PDA

satisfies the correct-prefix property if for all δ and v such that $(X_{init}, v) \vdash^* (\delta, \epsilon)$ we have some string w (vw ends in $\#$) such that $(X_{init}, vw) \vdash^* (X_{fin}, \epsilon)$.

Let us investigate more closely the pushing step of Algorithms **Dyna** and **Dyna-Suf**:

- push** (1) Choose a pair, not considered before, consisting of a transition $X \xrightarrow{z} XY$ and an input position j , such that there is an item $(W, h, X, j) \in U$, for some W and h , and such that $z = \epsilon \vee z = a_{j+1}$.
 (2) If $z = \epsilon$, then let $i = j$, else let $i = j + 1$.
 (3) Add item (X, j, Y, i) to U , if it is not already there.

This step causes a left-to-right dependency on the processing of input, or in other words, first some entries of column i have to be present in the table before any entries in column $(i + 1)$ can be computed. This left-to-right dependency is avoided by simplifying this step to

- push** (1) Choose a pair, not considered before, consisting of a transition $X \xrightarrow{z} XY$ and an input position j , such that $z = \epsilon \vee z = a_{j+1}$.
 (2) If $z = \epsilon$, then let $i = j$, else let $i = j + 1$.
 (3) Add item (X, j, Y, i) to U , if it is not already there.

The version of Algorithm **Dyna** that then results is reminiscent of an algorithm in Aho et al. [1968], which predates the dynamic programming algorithm from Lang [1974].

With the above simplification, both Algorithm **Dyna** and Algorithm **Dyna-Suf** eventually add an item (X, j, Y, i) , $j > 0$, to U if and only if⁵

- (1) $(X, a_{j+1} \dots a_i) \models^+ (XY, \epsilon)$.

This means that the simplified algorithms add more entries to the table, which makes them less efficient. It is surprising that the (worst-case) time complexity as computed in Lemma 4.7 is not affected. The simplification has a disadvantage however which is unrelated to the time complexity: without the simplification we have that Algorithm **Dyna-Suf** may only add an item (X, j, Y, i) , $j > 0$, to U if $(\perp, va_1 \dots a_i) \vdash^* (\delta X, a_{j+1} \dots a_i) \vdash^* (\delta XY, \epsilon)$, for some v and δ . This means that the existence of an item (X, j, Y, i) indicates that the input up to position i may be the prefix of a suffix of a sentence. In fact, the least $i \leq n$ such that no entries (X, j, Y, i) are added to U (if such an i exists) indicates that i is maximal such that $a_1 \dots a_{i-1}$ is a prefix of a suffix of a sentence, provided we may assume the correct-prefix property.

For applications in parsing of incorrect input, this idea is useful for finding multiple errors in an input string. First, Algorithm **Dyna** is started on the complete input $a_1 \dots a_n$. Suppose that column i_1 is the first one in the table to remain empty. Position i_1 must be seen as the first error in the input. Then we take the suffix $a_{i_1} \dots a_n$ of the complete input and we try to parse this new input using Algorithm **Dyna-Suf**. Suppose that column i_2 is now the first to remain empty (we number the columns starting from $i_1 - 1$ instead of starting from 0). Position i_2 can now be seen as the second error in the input. Algorithm **Dyna-Suf** is then repeated with input $a_{i_2} \dots a_n$, etc. This method not only works for deterministic PDAs, but for any PDA. It provides us with good heuristics how to locate the errors in an incorrect input.

⁵ For $j = 0$ the following discussion is equally relevant.

All these possibilities are lost when **Dyna** and **Dyna-Suf** are simplified as described above, because then all columns will contain some items, so that emptiness of columns is no longer a criterion for locating errors.

A more sophisticated way to parse incorrect input is described below. It is called *subsequence recognition*, since here the input is not a presumed suffix of a sentence, but a subsequence.

First note that algorithms such as Earley's algorithm and the dynamic programming algorithm are usually implemented as *synchronous* algorithms, which means that the columns of the table are computed strictly from left to right; first column 0, then column 1, etc.

The difference between Algorithms **Dyna** and **Dyna-Suf** can for synchronous computation be described as follows. Algorithm **Dyna** initializes column 0 with $\{(\perp, 0, X_{init}, 0)\}$ and then computes columns $0, 1, \dots, n + 1$. Algorithm **Dyna-Suf** does the same, except that it replaces the first column by its closure before computing columns $1, \dots, n + 1$. The application of *closure* on column 0 has the effect that Algorithm **Dyna-Suf** simulates processing of all prefixes of sentences by the PDA.

We can generalize this idea by applying *closure* on any other column except column 0. This means that first columns $0, 1, \dots, i$, for some i , are computed as usual; then column i is replaced by its closure, and then columns $i + 1, i + 2, \dots, n + 1$ are computed as usual. The result is that the input is recognized (i.e., eventually $(\perp, 0, X_{fin}, n + 1) \in U$) if and only if $a_1 \cdots a_i v a_{i+1} \cdots a_n$ is a sentence, for some v . This idea can be generalized to application of *closure* to any number of columns of U . If we apply *closure* to every column before computing the next column, then the input is recognized if and only if $v_0 a_1 v_1 \cdots v_{n-1} a_n v_n$ is a sentence for some $v_0 v_1 \cdots v_n$.

An important application of this idea is known as *insert-only error recovery* [Fischer and Manney 1992]: if we know that the input itself is not a sentence and if we conjecture that the error consists in some missing substrings, then this assumption may be verified by applying the above method.⁶

The algorithm in Lang [1988] for parsing incomplete input, from which our Algorithm **Dyna-Suf** is derived, assumes that the input is annotated with markers indicating where *closure* is to be applied.

Regrettably, linear-time subsequence recognition is not always possible for deterministic languages, even if the number of applications of *closure* is bounded.

8. Recursive-Descent Parsing

Tabular algorithms such as Earley's algorithm operate by manipulating tables U . These algorithms are therefore "interpretative": they are driven by the data in the table instead of by code of a program. One may argue that the time complexity suffers from the interpretative nature of the tabular algorithms, and that it is therefore better to change the structure of the algorithms so that some control information is put into a program.

⁶ Treating the input as a subsequence of a sentence is one extreme. The other extreme is to find the subsequences of the input which are sentences; see, for example, Lavie and Tomita [1993] and Nederhof [1994].

For top-down parsing, this is accomplished by using the *recursive-descent* method [Leermakers 1992]. The idea is that one procedure (or function) is assigned to each nonterminal, each rule or each “dotted rule” $[A \rightarrow \alpha \cdot \beta]$. In order to stress the similarity to Earley’s algorithm, we will give an example where each procedure corresponds to a dotted rule.

Construction: **Func-LL** (Functional LL(1) recognition)

Consider an LL(1) grammar $G = (T, N, P, S)$. Without loss of generality, assume that the grammar is reduced and that there is only one rule of the form $S \rightarrow \sigma$. Assume the input is $a_1 \cdots a_n$. Let $a_{n+1} = \#$. For each $[A \rightarrow \alpha \cdot \beta]$, where $A \rightarrow \alpha\beta \in P$, we construct one procedure. The procedures have two arguments, which are input positions, and yield a result that is again an input position, or the value “**failure**” to indicate failure of recognition. For the definition of the procedures we distinguish between three cases:⁷

```
[A → α · Bβ](j, i):  if there is no B → γ with ai+1 ∈ ℒA(B → γ)
                      then return “failure”
                      else let B → γ be such that ai+1 ∈ ℒA(B → γ);
                        let h = [B → · γ](i, i);
                        if h = “failure”
                        then return “failure”
                        else return [A → αB · β](j, h)
                      end
                    end.
```

```
[A → α · aβ](j, i):  if a = ai+1
                      then return [A → αa · β](j, i + 1)
                      else return “failure”
                      end.
```

```
[A → α ·](j, i):    return i.
```

The main procedure is:

```
main:  if [S → · σ](0, 0) = n
      then report “success: input is sentence”
      fi.
```

The correspondence between the program resulting from the above construction and Algorithm **Earley** is that $(j, [A \rightarrow \alpha \cdot \beta], i) \in U$ if and only if there is a call $[A \rightarrow \alpha \cdot \beta](j, i)$.

Algorithm **Earley** starts the parsing process by adding an item $(0, [S \rightarrow \cdot \sigma], 0)$ to U . In the same way, the program constructed above has all its control emanating from the call $[S \rightarrow \cdot \sigma](0, 0)$. This poses some difficulty when we try to realize Algorithm **Earley-Suf**, for suffix parsing, using the recursive-descent method. The reason is that in I_0^0 there is no single item from which the control emanates. In particular, there may be cyclic dependencies among the items in I_0^0 . Such cyclic dependencies cannot be realized using the recursive-descent method and therefore the items $(0, [A \rightarrow \alpha \cdot \beta], i)$ still need to be computed by an interpretative method:

⁷ The definitions of the procedures can be much simplified. For example, the first arguments can be removed since they are redundant, and the tail recursion can be turned into a loop. All this is however beyond the scope of this paper.

Construction: **Func-LL-Suf** (Functional LL(1) suffix recognition)

Consider an LL(1) grammar as before, and construct the procedures $[A \rightarrow \alpha \cdot \beta]$ as in Construction **Func-LL**. Assume the input is $a_1 \cdots a_n$, $n > 0$.⁸ Let $a_{n+1} = \#$. The main procedure is now given by:

```

main: let  $W = \{([A \rightarrow \alpha a \cdot \beta], 1) \mid A \rightarrow \alpha a \beta \in P \wedge a = a_1\}$ ;
  for all  $([A \rightarrow \alpha \cdot \beta], i) \in W$  not considered before
  do let  $h = [A \rightarrow \alpha \cdot \beta](0, i)$ ;
    if  $h \neq \text{"failure"}$ 
    then let  $W = W \cup \{([C \rightarrow \gamma B \cdot \delta], h) \mid C \rightarrow \gamma B \delta \in P \wedge B = A\}$ ;
      if  $A = S \wedge h = n$ 
      then report "success: input is suffix"
    end
  end
end.

```

That the behavior of procedure **main** is similar to the behavior of Algorithm **Earley-Suf** is witnessed by the fact that for $i > 0$, $(j, [A \rightarrow \alpha \cdot \beta], i) \in U$ if and only if there is a call $[A \rightarrow \alpha \cdot \beta](j, i)$. In fact, if the procedures are implemented as memo functions, the behavior of the program is almost identical to the behavior of Algorithm **Earley-Suf**.

Without going into details, we just mention that the above recognition procedures can easily be extended to be parsing procedures.

The procedure **main** can be considered to represent a finite state automaton over the set of grammar symbols V . Bertsch [1994] proposes to determinize this automaton to obtain a better run-time efficiency, although this increases the size of the parser.

For LR parsing, procedures similar to the ones above can be constructed according to the recursive-ascent method [Leermakers 1992].

9. Generalizations to Nondeterministic Automata

Algorithms **Earley**, **Earley-Suf**, **Dyna**, and **Dyna-Suf** all derive from general context-free parsing algorithms. In fact, they correctly simulate deterministic as well as nondeterministic PDAs. This means that the techniques from this paper can be used also for suffix parsing and parsing of incorrect input for *arbitrary* context-free languages, although here a linear time complexity is not guaranteed.

This is a useful property for error recovery in compilers: Often programming languages are "not quite" LL(k) or LR(k) and the nondeterminism during normal parsing is circumvented by special conflict resolvers. When a syntactic error occurs, these conflict resolvers may not be available for some reason. The applicability of our algorithms to recovery of the error, however, is not hindered by the nondeterminism that may then ensue.

The PDA transformation τ requires special attention when we allow nondeterminism. On the one hand, application of τ can without many consequences be omitted in this case, since its only use was to ensure a linear time complexity for tabular simulation of *deterministic* PDAs, whereas a linear time complexity cannot be ensured for *nondeterministic* PDAs, regardless of any language-preserving PDA transformation.

⁸ Without loss of generality, we restrict ourselves to nonempty input.

On the other hand, it seems reasonable to suggest that PDAs that are “almost” deterministic may be simulated by Algorithm **Dyna-Suf** in “almost” linear time, provided some generalized transformation τ is applied first.

Not much effort is needed to generalize the definition of τ to nondeterministic PDAs. In fact, the proof that it preserves the language accepted by a deterministic PDA (see Lemma 4.3) does not even make use of the determinism. Therefore, τ may also be applied to nondeterministic PDAs without affecting the accepted language. Regrettably, for nondeterministic PDAs, the sets *PUSH* and *POP* may not be disjoint, which causes τ to produce spurious nondeterminism. This problem is avoided by marking stack symbols if they should be interpreted as an element in *POP*, blocking any transition that would push an element on top of it. The result is the following:

Construction: τ' (Generalized transformation to loop-free PDAs)

Consider a PDA \mathcal{A} . We define these two sets which are not necessarily disjoint:

$$PUSH = \{X \mid \text{there is a transition } X \xrightarrow{\cdot}_{\mathcal{A}} XY\}$$

$$POP = \{Y \mid \text{there is a transition } XY \xrightarrow{\epsilon}_{\mathcal{A}} Z\} \cup \{X_{fin}\}.$$

Without loss of generality we assume that $X_{init} \notin POP$. A new PDA $\tau'(\mathcal{A})$ is constructed, of which the stack symbols are those in *PUSH* plus stack symbols of the form \hat{X} , with $X \in PUSH$, plus stack symbols of the form \bar{X} , with $X \in POP$. The transitions of $\tau'(\mathcal{A})$ are given by

$$\begin{array}{llll} X\hat{Y} & \xrightarrow{\epsilon}_{\tau'(\mathcal{A})} & \hat{Z} & \text{for } XY \xrightarrow{\epsilon}_{\mathcal{A}} Z \text{ with } Z \in POP \\ X\hat{Y} & \xrightarrow{\epsilon}_{\tau'(\mathcal{A})} & Z & \text{for } XY \xrightarrow{\epsilon}_{\mathcal{A}} Z \text{ with } Z \in PUSH \\ \bar{X} & \xrightarrow{\epsilon}_{\tau'(\mathcal{A})} & XX & \text{for } X \in PUSH \\ \bar{X}\hat{Y} & \xrightarrow{\epsilon}_{\tau'(\mathcal{A})} & \hat{Y} & \text{for } X \in PUSH, Y \in POP \\ X & \xrightarrow{\cdot}_{\tau'(\mathcal{A})} & X\hat{Y} & \text{for } X \xrightarrow{\cdot}_{\mathcal{A}} XY \text{ with } Y \in POP \\ X & \xrightarrow{\cdot}_{\tau'(\mathcal{A})} & XY & \text{for } X \xrightarrow{\cdot}_{\mathcal{A}} XY \text{ with } Y \in PUSH. \end{array}$$

The initial and final stack symbols for $\tau'(\mathcal{A})$ are X_{init} and $\widehat{X_{fin}}$ respectively.

Note that if the PDA \mathcal{A} is deterministic and therefore *PUSH* and *POP* are disjoint, then τ' simplifies to τ , apart from the renaming of stack symbols $X \in POP$ in $\tau(\mathcal{A})$ to stack symbols \bar{X} in $\tau'(\mathcal{A})$.

That the time complexity of Algorithm **Dyna-Suf** is “almost” linear for PDAs that are “almost” deterministic and that have been transformed by $\tau'(\mathcal{A})$ can obviously not be proved by formal arguments. This raises the question whether τ' may also have a detrimental effect on the time complexity. The answer,

regrettably, is affirmative. For example, consider a nondeterministic PDA having (among others) the following transitions:

$$\begin{array}{lll} Q & \xrightarrow{\epsilon} & QX \\ X & \xrightarrow{a} & XY \quad XY \xrightarrow{\epsilon} X \\ X & \xrightarrow{\epsilon} & XZ \quad XZ \xrightarrow{\epsilon} R \\ QR & \xrightarrow{\epsilon} & R \end{array}$$

Let us consider the time complexity of simulation of these transitions by Algorithm **Dyna-Suf** if we assume Q occurs on top of the stack at a single input position, say position 1, and if the input is $a_1 \cdots a_n$, with $a_i = a$, for $1 \leq i \leq n$. We have sequences of configurations of the form

$$\begin{array}{ll} (Q, a_1 a_2 a_3 \cdots a_n) \vdash & \\ (QX, a_1 a_2 a_3 \cdots a_n) \vdash & (QXY, a_2 a_3 \cdots a_n) \vdash \\ (QX, a_2 a_3 \cdots a_n) \vdash & \\ \vdots & \\ (QX, a_i a_{i+1} \cdots a_n) \vdash & (QXY, a_{i+1} \cdots a_n) \vdash \\ (QX, a_{i+1} \cdots a_n) \vdash & (QXZ, a_{i+1} \cdots a_n) \vdash \\ (QR, a_{i+1} \cdots a_n) \vdash & \\ (R, a_{i+1} \cdots a_n) & \end{array}$$

for different i , with $1 \leq i \leq n$. Of such a sequence, only the last three steps are unique and are not simulated together with steps in other sequences by Algorithm **Dyna-Suf**. This means that Algorithm **Dyna-Suf** simulates these $\mathcal{O}(n)$ sequences in linear time.

For the transformed automaton we have

$$\begin{array}{ll} (Q, a_1 a_2 a_3 \cdots a_n) \vdash & \\ (QX, a_1 a_2 a_3 \cdots a_n) \vdash & (QX\hat{Y}, a_2 a_3 \cdots a_n) \vdash (Q\bar{X}, a_2 a_3 \cdots a_n) \vdash \\ (Q\bar{X}X, a_2 a_3 \cdots a_n) \vdash & \\ \vdots & \\ (Q\bar{X}^{i-1}X, a_i a_{i+1} \cdots a_n) \vdash & (Q\bar{X}^{i-1}X\hat{Y}, a_{i+1} \cdots a_n) \vdash (Q\bar{X}^i, a_{i+1} \cdots a_n) \vdash \\ (Q\bar{X}^i X, a_{i+1} \cdots a_n) \vdash & (Q\bar{X}^i X\hat{Z}, a_{i+1} \cdots a_n) \vdash \\ (Q\bar{X}^i \hat{R}, a_{i+1} \cdots a_n) \vdash & \\ (Q\bar{X}^{i-1} \hat{R}, a_{i+1} \cdots a_n) \vdash & \\ \vdots & \\ (Q\bar{X} \hat{R}, a_{i+1} \cdots a_n) \vdash & \\ (Q\hat{R}, a_{i+1} \cdots a_n) \vdash & \\ (\hat{R}, a_{i+1} \cdots a_n) & \end{array}$$

Algorithm **Dyna-Suf** does not simulate the final $i + 2$ steps of such a sequence together with steps in other such sequences. In other words, $\mathcal{O}(n)$ steps are simulated individually for each of the $\mathcal{O}(n)$ sequences of the form above, which implies that Algorithm **Dyna-Suf** has a quadratic time complexity in this case.

We conclude that τ' is not useful in general for improving the time complexity of Algorithm **Dyna-Suf** for nondeterministic PDAs.

10. Related Research for LR Recognition

The class of deterministic languages is equal to the class of LR(1) languages. Correspondingly, some algorithms from the existing literature which perform suffix recognition or suffix parsing based on the LR technique are comparable to the algorithms in this paper.

The algorithm in Rekers and Koorn [1991], for example, is very similar to Algorithm **Dyna-Suf** for a PDA realizing LR parsing. A superficial difference is that the *closure* operation is realized implicitly (as in Construction **Func-LL-Suf**), which has practical advantages but little theoretical significance.

The bidirectional LR parsing algorithm in Saito [1990] is also related: LR parsing may start at any input position i , and process the input from i to the right. The difference with Rekers and Koorn [1991] is that simultaneously also the input from i to the left is processed by a reverse LR algorithm. Both parsing processes, one working from i to the right and the other from i to the left, co-operate to compute nonterminals that generate substrings of the input including the position i . Further generalizations described in Saito [1990] allow parsing processes to start from a number of input positions simultaneously; this is beyond the scope of this paper however.

Neither of the algorithms in Rekers and Koorn [1991] and Saito [1990] is linear in the length of the input for all LR grammars. The explanation is that LR automata are not loop-free for left-recursive grammars (for the reverse LR automata from Saito [1991], the obstacle is of course *right* recursion).

The solution presented in Bates and Lavie [1994], developed independently from our own, can be described as follows. In Section 4, we argued that a nonlinear time complexity may result if there is no linear bound on the number of items $(X, j, Y, i) \in U$ for fixed X and Y . We solved this problem by transforming the PDAs such that only one such item can exist for each j . The alternative solution in Bates and Lavie [1994] is to alter the tabular algorithm itself, so that different items $(X, j, Y, i) \in U$ for fixed X , Y and i are represented by a single object in the parsing table. Shared representation of such items for different j is made possible by choosing a kind of parsing table that allows the distinction between different input positions to be partly neglected. Such a data structure is the concept of graph-structured stacks mentioned in Example 4.9, with the difference that nodes do not have to be uniquely related to input positions. This variant of a graph-structured stack is called a *forest-structured stack*.

We take a closer look at the relationship to our approach. For Algorithm **Dyna-Suf**, a list of items in the table U of the form $(X_0, i_0, X_1, i_1), (X_1, i_1, X_2, i_2), \dots, (X_{m-1}, i_{m-1}, X_m, i_m)$, with $i_0 = 0$, indicates the existence of a stack $\delta X_0 X_1 \dots X_m$, some δ , which results from an initial configuration by reading some unknown prefix followed by the input symbols up to position i_m . Such lists

of items correspond to paths in the graph-structured stack, which at this point is just a pictorial representation for the set of items U . Such a path consists of nodes s_m, \dots, s_0 labeled X_m, \dots, X_0 , respectively, and arrows connecting s_k to s_{k-1} , $0 < k \leq m$. The nodes are divided into subsets according to the input position where they are created, that is, node s_k belongs to the i_k th set.

The approach in Bates and Lavie [1994] relies on the fact that for a tabular realization of *deterministic* PDAs the input positions in the paths are not essential, and consequently the nodes in the paths do not need to be divided into subsets according to input positions. In fact, some paths starting at the same node may be partly merged if they have some labels in common in an initial subpath, even if the corresponding input positions are different.

In order to explain the difference between our Algorithm **Dyna-Suf** and the algorithm in Bates and Lavie [1994], we consider again the quadratic behavior that we have discussed in Section 4. We do this by taking a collection of PDA transitions similar to the one we presented in Figure 4:

$$\begin{array}{llll}
 P & \xrightarrow{a} & PQ & Q \xrightarrow{a} QX \\
 X & \xrightarrow{a} & XY & XY \xrightarrow{\epsilon} X \\
 X & \xrightarrow{\#} & XZ & XZ \xrightarrow{\epsilon} R \\
 QR & \xrightarrow{\epsilon} & R &
 \end{array}$$

In the worst case, Algorithm **Dyna-Suf** simulates $\mathcal{O}(n)$ sequences of transitions $(P, a_j \cdots a_n \#) \vdash^* (PR, \epsilon)$, for different j . For three of these sequences, viz. for $j = n - 3, n - 2, n - 1$, the subsets of U relevant to simulation of these sequences are given as separate subgraphs of the graph-structured stack in Figure 7(a). Items of the form (X, j, Y, i) and (X, j, Z, i) have been omitted to simplify the pictures. The latent quadratic behavior reminiscent of Example 4.9 is also present in this case: we may have quadratically many items of the form (Q, j, X, i) ; linearly many for each subset of U simulating one of the linearly many sequences of transitions $(P, a_j \cdots a_n \#) \vdash^* (PR, \epsilon)$, for different j .

Figure 7(b) gives the corresponding three subsets of U resulting after the PDA is transformed by τ . The quadratic behavior disappears. Note that, for example, the item $(\bar{X}, n, R, n + 1)$ occurs in both the first and the second subgraph. The three subgraphs are given together in Figure 7(c).

Figure 7(d) shows how the algorithm in Bates and Lavie [1994] achieves a linear time complexity. The dashed lines indicate arrows in the graph that are at some point in time constructed but later merged with other arrows. For example, at some point there will be two paths from the node labeled X created at position $n - 1$. One is along the node labeled Q created at $n - 3$, and the node labeled P created at $n - 4$. The other path is also along two nodes with the labels Q and P , in this order, but these two nodes were created at positions $n - 2$ and $n - 3$, respectively. The algorithm merges this latter path with the former, thus avoiding quadratically many arrows from nodes labeled X to nodes labeled Q . Merging of paths takes place recursively: first the two nodes labeled Q at $n - 3$ and $n - 2$ are merged together with the two incoming arrows from the node labeled X at $n - 1$. Then the two nodes both labeled P further down the two paths are merged, together with the two connecting arrows. This process continues further

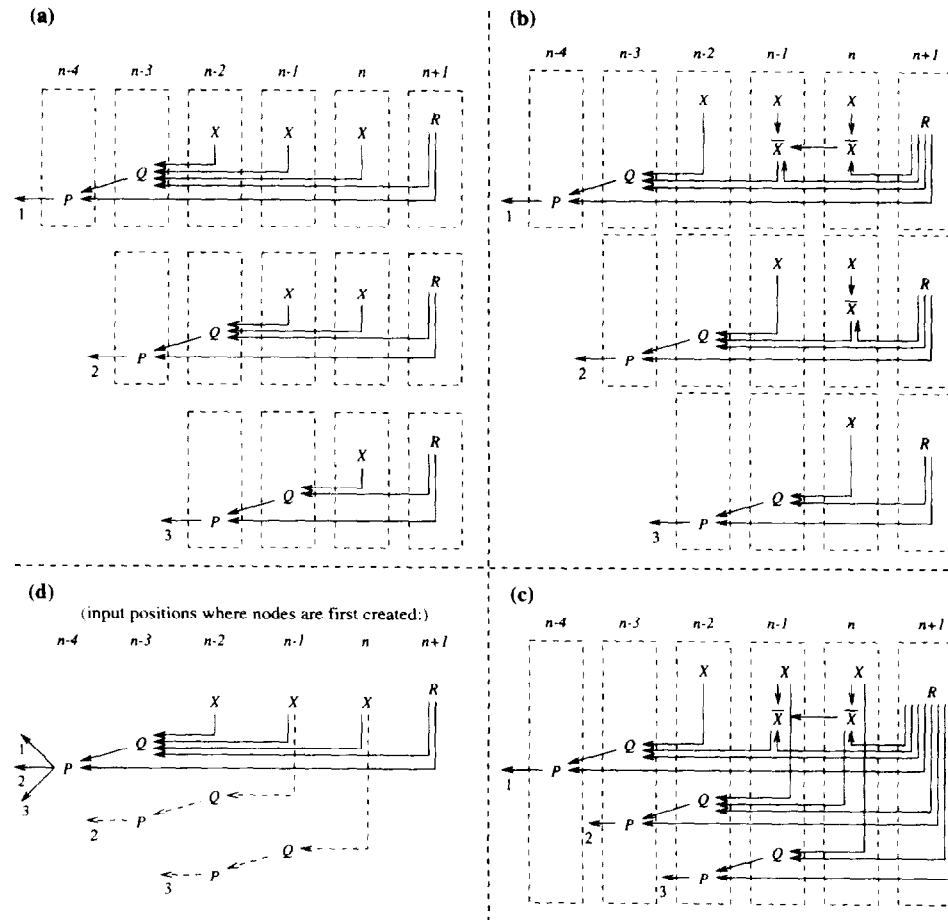


FIG. 7. Two different ways of achieving a linear time complexity.

- Pictorial representation (parse forest) for items in U , distinguished according to three sequences of configurations simulated by Algorithm **Dyna-Suf**. The total number of items (Q, j, X, i) grows quadratically with n .
- The items corresponding to those in (a) in the case of $\pi(\mathcal{A})$. Certain items are shared between the three simulated sequences of configurations.
- The items from (b) represented together. The number of items grows linearly with n (cf. Figure 6).
- Forest-structured stack according to Bates and Lavie [1994]. The dashed arrows represent parts of paths that are merged into other paths, without regard to input positions.

down the paths (from the two arrows in our picture labeled 1 and 2) until the paths differ with regard to the labels at the nodes.

Let us now investigate the advantages of our approach over the one in Bates and Lavie [1994].

A superficial advantage of our approach is that we allow suffix parsing to be based on any parsing technique, be it top-down, left-corner or LR parsing, whereas Bates and Lavie [1994] only treats LR parsing. This is not a significant difference however, since the method to obtain a linear time complexity from Bates and Lavie [1994] can be generalized to other parsing techniques besides

LR parsing. In discussing the example above, we tacitly assumed that this kind of generalization had already taken place.

A more substantial advantage is the conceptual simplicity of our approach. This is achieved by separating the PDA transformation τ from the tabular algorithm (Algorithm **Dyna-Suf**). Taking into account that the tabular algorithm is based on the well-established algorithm by Earley [1970], the remaining proof obligations involve no more than a simple PDA transformation. These proof obligations have been fulfilled by straightforward deductions in Section 4.

Our modular approach results in increased flexibility. In particular, we may safely omit the PDA transformation or take different variants for the tabular algorithm.

An example of when it may be preferable to omit the PDA transformation τ occurs when the original PDA is not deterministic, as explained in Section 9. We cannot use the algorithm in Bates and Lavie [1994] in this case, however: when the PDA is nondeterministic, each node in the forest-structured stack may have multiple incoming arrows which are reachable from some node which represents the top of some stack. When such a node is merged with an older node, then some of these incoming arrows become “dangling references”, causing loss of completeness of the recognition algorithm.

Different variants for our tabular Algorithm **Dyna-Suf** are, for example, synchronous and asynchronous ones, with or without left-to-right order of processing (see Section 7). Since in Bates and Lavie [1994] the process of merging paths is deeply integrated into the tabular algorithm, such variants cannot be defined without also adapting this process. The approach in Bates and Lavie [1994] is therefore less flexible.

The most important property that distinguishes the algorithm in Bates and Lavie [1994] from our own is that the former is intrinsically a *recognition* algorithm, as opposed to a *parsing* algorithm. The reason that it cannot be augmented to be a parsing algorithm is because the unique identification of items in the parsing table with input positions is essential for the construction of parse forests as explained in Section 6.

To illustrate the problem, consider in the running example the two arrows from the node labeled X at $n - 1$ to the two nodes labeled Q at $n - 3$ and $n - 2$. These two arrows may be identified with two subparses, one parsing input from position $n - 3$ up to $n - 1$, the other from $n - 2$ to $n - 1$. After these two arrows have been merged, the correct identification of the two subparses is lost, and defective parses result when these are combined with other subparses of preceding or following input.

For the same reason, the algorithm cannot be augmented to handle attribute grammars nor related kinds of formalism. However, none of these difficulties occur for our approach, since the PDA transformation can be adapted to take the construction of parse forests or the evaluation of attributes into account, as we have shown in Section 6.

11. Conclusions

The purpose of this paper is to lay a theoretically attractive foundation for efficient noncorrecting error recovery.

The core of our discussion has been the observation that recognition or rejection of a string as a suffix of a sentence can be done in linear time for deterministic languages (Section 4, and Section 3 for the special case of LL(1) languages). In case the string is rejected as suffix, the algorithm indicates the longest prefix of the string which is a substring of some sentence, provided we may assume the correct-prefix property. By repeating this at each input position where the algorithm has stopped during the previous iteration, an incorrect input is partitioned into pieces each of which is a substring of some sentence (Section 7). This results in a linear-time recognition algorithm for incorrect input, which can be extended to be a parsing algorithm, producing parse trees for each correct substring separately (Section 6). In Section 7, we also discussed subsequence recognition, which however does not have a linear time complexity in the general case.

In order to argue the practical value of these ideas, we have shown that the tabular techniques can also be cast into a functional form, which avoids some interpretative processing and allows compilation of the grammar into parsing procedures. We then rely on memoization for maintaining a linear time complexity (Section 8).

ACKNOWLEDGMENTS. Holger Streit and Martin Tenhaven helped us to find a flaw in an earlier version of the paper. An anonymous referee provided us with valuable comments that we have used to improve the readability of the text. We thank Kees Koster for proofreading.

REFERENCES

- AHO, A. V., HOPCROFT, J. E., AND ULLMAN, J. D. 1968. Time and tape complexity of pushdown automaton languages. *Inf. Cont.* 13, 186–206.
- BATES, J., AND LAVIE, A. 1994. Recognizing substrings of LR(k) languages in linear time. *ACM Trans. Prog. Lang. Syst.* 16, 3 (May), 1051–1077.
- BERTSCH, E. 1994. An asymptotically optimal algorithm for noncorrecting LL(1) error recovery. Bericht Nr. 176, Fakultät für Mathematik, Ruhr-Universität Bochum.
- BILLOT, S., AND LANG, B. 1989. The structure of shared forests in ambiguous parsing. In *Proceedings of the 27th Annual Meeting of the Association for Computational Linguistics*, (Vancouver, British Columbia, Canada, June), pp. 143–151.
- BOUCKAERT, M., PIROTTE, A., AND SNELLING, M. 1975. Efficient parsing algorithms for general context-free parsers. *Inf. Sci.* 8, 1–26.
- CORMACK, G. V. 1989. An LR substring parser for noncorrecting syntax error recovery. *SIGPLAN Notices* 24, 7, 161–169.
- EARLEY, J. 1970. An efficient context-free parsing algorithm. *Commun. ACM* 13, 2 (Feb.), 94–102.
- FISCHER, C. N., AND MAUNEY, J. 1992. A simple, fast, and effective LL(1) error repair algorithm. *Acta Inf.* 29, 109–120.
- GRAHAM, S. L., HARRISON, M. A., AND RUZZO, W. L. 1980. An improved context-free recognizer. *ACM Trans. Prog. Lang. Syst.* 2, 3 (July), 415–462.
- GRUN, D., AND JACOBS, C. J. H. 1990. *Parsing Techniques, A Practical Guide*. Ellis Horwood, Chichester, England.
- LANG, B. 1974. Deterministic techniques for efficient non-deterministic parsers. In *Automata, Languages and Programming, 2nd Colloquium*. Lecture Notes in Computer Science, vol. 14. Springer-Verlag, New York, pp. 255–269.
- LANG, B. 1988. Parsing incomplete sentences. In *Proceedings of the 12th International Conference on Computational Linguistics*, vol. 1. (Budapest, Hungary, Aug.), pp. 365–371.
- LANG, B. 1991. Towards a uniform formal framework for parsing. In *Current Issues in Parsing Technology*, chapter 11, M. Tomita, ed. Kluwer Academic Publishers, pp. 153–171.

- LAVIE, A., AND TOMITA, M. 1993. GLR*—An efficient noise-skipping parsing algorithm for context free grammars. In *Proceedings of the 3rd International Workshop on Parsing Technologies*. Tilburg (The Netherlands) and Durbuy (Belgium). pp. 123–134.
- LEERMAKERS, R. 1992. Recursive ascent parsing: From Earley to Marcus. *Theoret. Comput. Sci.* 104, 299–312.
- LEISS, H. 1990. On Kilbury's modification of Earley's algorithm. *ACM Trans. Prog. Lang. Syst.* 12, 4 (Oct.). 610–640.
- NEDERHOF, M. J. 1994. Linguistic parsing and program transformations. Ph.D dissertation. Univ. Nijmegen.
- REKERS, J., AND KOORN, W. 1991. Substring parsing for arbitrary context-free grammars. *SIGPLAN Notices* 26, 5, 59–66.
- RICHTER, H. 1985. Noncorrecting syntax error recovery. *ACM Trans. Prog. Lang. Syst.* 7, 3 (July) 478–489.
- SAITO, H. 1990. Bi-directional LR parsing from an anchor word for speech recognition. In *Papers presented to the 13th International Conference on Computational Linguistics*, vol. 3. pp. 237–242.
- SIPPU, S., AND SOISALON-SOININEN, E. 1990. Parsing Theory, Vol. II: $LR(k)$ and $LL(k)$ Parsing. In *EATCS Monographs on Theoretical Computer Science*, vol. 20. Springer-Verlag, New York.
- TOMITA, M. 1986. *Efficient Parsing for Natural Language*. Kluwer Academic Publishers, Dordrecht, The Netherlands.
- VALIANT, L. G. 1975. General context-free recognition in less than cubic time. *J. Comput. Syst. Sci.* 10, 308–315.
- VAN DEUDEKOM, B., AND KOOIMAN, P. 1993. Top-down non-correcting error recovery in LLgen. Report IR-338, Vrije Universiteit Amsterdam, Amsterdam, The Netherlands.

RECEIVED NOVEMBER 1994; REVISED SEPTEMBER 1995; ACCEPTED DECEMBER 1995