Cross-Fertilization of Earley and Tomita

Klaas Sikkel

Computer Science Department, University of Twente,
PO Box 217, 7500 AE Enschede, The Netherlands
sikkel@cs.utwente.nl

Abstract

Earley's and Tomita's algorithm are superficially rather different. Their underlying structure, however, is remarkably similar. For a closer inspection of differences and similarities, we define variants of Earley and Tomita that produce (almost) indistinguishable results: a parse stack annotated with recognized items. Tomita's algorithm is more sensitive to the complexity of this parse stack. It will be better than Earley's algorithm, provided that the grammar is not too complicated, i.e., not densely ambiguous.

As an application of cross-breeding between the two species, we sketch a novel, elegant parallel parser. Combining a well-known parallelization strategy of Earley with the algorithmic approach of Tomita yields a Parallel Bottom-up Tomita parser.

1 Introduction

There is a variety of parsing algorithms and an even wider variety of parallel parsing algorithms. The work presented here is part of a research project that aims at bringing some order into the not yet very structured field of parallel parsing. We envisage the following general approach. If we abstract from the data structures used and other algorithmic details, we can perceive *parsing strategies*: which (sub)products are to be computed and — not unimportant for parallelization — what are the constraints on the ordering of the computation steps? More specifically, we can classify bottom-up parsing algorithms according to their *recognized item set* (RIS). A recognized item denotes the fact that some symbol, or part of a production, has been identified to produce a specific part of the sentence. Such a categorization seems rather natural for chart parsers (c.q. Earley), but we will show that it easily applies to LR parsers (c.q. Tomita) as well.

We will first show that — with minor modifications — Earley and Tomita have an identical RIS. Moreover, we introduce variants of Earley and Tomita that produce an almost identical data structure. This graph structured variant of Earley and annotated variant of Tomita allow a detailed comparison of both algorithms. We know quite a few people who shared our conviction that, basically, Earley and Tomita are rather similar, but we do not know of any paper in which the two algorithms are systematically compared. It is generally acknowledged that Tomita is more efficient on easy grammars; it has compiled a fair part of the work that needs to be done run-time by Earley. The worst-case behaviour, however, is less clear. It can be argued that Tomita need *not* be slower (up to a constant factor) than Earley, even for hard (i.e. densely ambiguous) grammars. But this is only of theoretical value.

Having uncovered the similarity between the two algorithms, we can apply extensions and variations of one species to the other species as well. As a particularly interesting case, we sketch how a well-known parallelization strategy of Earley [Graham80] can be combined with the algorithmic approach of Tomita. The resulting Parallel Bottom-up Tomita (PBT) algorithm is a (with hindsight) rather obvious, but nevertheless unknown parallelization of Tomita. The PBT algorithm is investigated further in [Lankhorst91].

The successful cross-breeding of Parallel Bottom-up Earley and Tomita into Parallel Bottom-up Tomita is a sign that our approach towards parallel parsing is a fruitful one.

2 Definitions and notation

We consider context-free grammars $G = (N, \Sigma, P, S)$, with nonterminal symbols N, terminal symbols Σ, a set P of productions of the form $A \rightarrow \alpha$ with $A \in N$, $\alpha \in (N \cup \Sigma)^*$ and a start symbol S. We write V for $N \cup \Sigma$. We assume G to be cycle-free, i.e., there is no sequence of productions $A \Rightarrow^+ A$. Within the context of natural languages this is not an unreasonable assumption.

Triples of the form $[i, A \rightarrow \alpha \bullet \beta, j]$, with $A \rightarrow \alpha\beta$ a production and i, j numbers that denote positions in the sentence, are called *items*. For a string of length n, the *set of items* Ξ is defined by

$$\Xi \stackrel{\text{def}}{=} \{[i, A \rightarrow \alpha \bullet \beta, j] \mid A \rightarrow \alpha\beta \in P,\ 0 \leq i \leq j \leq n\}.$$

If at some moment during parsing it is established that some prefix α of the right-hand side of a production $A \rightarrow \alpha\beta$ produces a substring $a_{i+1} \cdots a_j$ of the sentence, we will denote this fact by adding the item $[i, A \rightarrow \alpha \bullet \beta, j]$ to a set I of items recognized so far.

Parsers that deal with such items explicitly can be classified according to their *recognized item set* (*RIS*), that is, all items that are recognized during parsing. Thus $I = \emptyset$ at the start and $I =$ *RIS* at the end of an algorithm. Earley's algorithm is characterized by a recognized item set

$$RIS_E = \{[i, A \rightarrow \alpha \bullet \beta, j] \mid \alpha \Rightarrow^* a_{i+1} \cdots a_j \wedge \exists \gamma \in V^* : S \Rightarrow^* a_1 \cdots a_i A \gamma\}.$$

Subsequently we will present a particular variant of Tomita's algorithm that can be characterized by the same recognized item set RIS_E.

As an alternative notation for items we write

$$\Xi_j \stackrel{\text{def}}{=} \{[i, A \rightarrow \alpha \bullet \beta] \mid A \rightarrow \alpha\beta \in P,\ 0 \leq i \leq j\},$$

and we maintain separate item sets $I_j \subset \Xi_j$ for $0 \leq j \leq n$. If $[i, A \rightarrow \alpha \bullet \beta, j] \in I$ then $[i, A \rightarrow \alpha \bullet \beta] \in I_j$ and vice versa. As a third, equivalent notation we introduce (identical) sets of items

$$\Xi_{i,j} \stackrel{\text{def}}{=} \{[A \rightarrow \alpha \bullet \beta] \mid A \rightarrow \alpha\beta \in P\}$$

for $0 \leq i \leq j \leq n$ and maintain separate item sets $I_{i,j} \subset \Xi_{i,j}$. Consequently, $[i, A \rightarrow \alpha \bullet \beta] \in I_j$ iff $[A \rightarrow \alpha \bullet \beta] \in I_{i,j}$. We will switch notation according to what is most practical in a particular context.

As an example throughout this article, we will use the following simple grammar G:

(1) $S \rightarrow NP\ VP$
(2) $S \rightarrow S\ PP$
(3) $*det \rightarrow \varepsilon$
(4) $NP \rightarrow *det\ *noun$
(5) $NP \rightarrow NP\ PP$
(6) $PP \rightarrow *prep\ NP$
(7) $VP \rightarrow *verb\ NP$

Production (3) usually reads $NP \rightarrow *noun$. That is, a noun phrase can have the form **det *noun* or **noun*. In this case, a noun phrase always has the form **det *noun*, but the **det* may be omitted. Both grammars are equivalent, but G as defined above is slightly more awkward to parse. This makes it very suitable for the examples to follow. Note that **det* is also used as a nonterminal in $*det \rightarrow \varepsilon$, but this will not cause any confusion. When space is limited (as in pictures) we write **d*, **n*, **v*, **p* for **det*, **noun*, **verb* and **prep*, respectively.

3 Earley's algorithm

Earley's algorithm is one of the first efficient parsing algorithms that can handle any context-free grammar (including cyclic grammars) [Earley68,70]. In the past two decades some smaller improvements have been made and many approaches to parallelization of the algorithm have been suggested. See, e.g., [Graham80], [Chiang84], [Nijholt91]. In this paper we describe Earley's algorithm rather tersely.

In Figure 1, the item sets I_j are shown that should be computed by Earley's algorithm for the input sentence *John saw a lion*. It can be verified that $RIS_E = \bigcup_{0 \le j \le n} I_j$. Such a table can be completed in the following way. We start with an item $[0, S \rightarrow \bullet NP\ VP] \in I_0$. That is, we intend to recognize a sentence, but no part of it has been recognized so far. We add items in each of the following ways:

- If $[i, A \rightarrow \alpha \bullet B\beta] \in I_j$ and $B \rightarrow \gamma \in P$ then $[i, B \rightarrow \bullet \gamma]$ is added to I_j (*predict*);
- if $[i, B \rightarrow \alpha \bullet a\beta] \in I_{j-1}$ and $a = a_j$ then $[i, B \rightarrow \alpha a \bullet \beta]$ is added to I_j (*scan*);
- if $[k, B \rightarrow \gamma \bullet] \in I_j$ and $[i, A \rightarrow \alpha \bullet B\beta] \in I_k$ then $[i, A \rightarrow \alpha B \bullet \beta]$ is added to I_j (*complete*).

Exhaustive application of these steps yields the item sets as shown in Figure 1.

In the following formal description of Earley's algorithm we have incorporated the *predict* operation into *scan* and *complete*. Let D be the set of dotted rules $\{A \rightarrow \alpha \bullet \beta \mid A \rightarrow \alpha\beta \in P\}$. We define functions $predict : N \rightarrow D$ and $predictor : \Xi_j \times \{1, \cdots, n\} \rightarrow 2^{\Xi_j}$ as follows:

$$predict(A) \stackrel{\text{def}}{=} \{B \rightarrow \bullet \beta \mid B \rightarrow \beta \in P,\ \exists\, \gamma \in V^* : A \Rightarrow^* B\gamma\}.$$

$$predictor([i, A \rightarrow \alpha \bullet \beta], j) \stackrel{\text{def}}{=} \begin{cases} \{[i, A \rightarrow \alpha \bullet \beta]\} & \text{if } \beta = \varepsilon \text{ or } \beta = a\gamma \text{ for some } a \in \Sigma, \gamma \in V^* \\ \{[i, A \rightarrow \alpha \bullet \beta]\} \cup \{[j, C \rightarrow \bullet \delta] \mid C \rightarrow \bullet \delta \in predict(B)\} & \text{if } \beta = B\gamma \text{ for some } B \in N, \gamma \in V^* \end{cases}$$

We define operations *SCAN* and *COMPLETE* including the *predictor* function

SCAN (j): **for each** $[i, B \rightarrow \alpha \bullet a\beta] \in I_{j-1}$ **such that** $a = a_j$
do $I_j := I_j \cup predictor([i, B \rightarrow \alpha a \bullet \beta], j)$;

COMPLETE (j): **while** $\exists\, A, B \in N, \alpha, \beta, \gamma \in V^*, i, k\ (0 \le i \le k \le j)$
such that $[k, B \rightarrow \gamma \bullet] \in I_j$, $[i, A \rightarrow \alpha \bullet B\beta] \in I_k$, $[i, A \rightarrow \alpha B \bullet \beta] \notin I_j$
do $I_j := I_j \cup predictor([i, A \rightarrow \alpha B \bullet \beta], j)$;

yielding the following top-level description of Earley's algorithm:

$I_0 := \{[0, A \rightarrow \alpha \bullet] \mid A \rightarrow \bullet a \in predict(S)\}$;
COMPLETE (0);
for $j := 1$ **to** n **do**
begin
    $I_j := \varnothing$;
    SCAN (j);
    COMPLETE (j)
end;
if $[0, S \rightarrow \omega \bullet] \in I_n$ for some $S \rightarrow \omega \in P$ **then** *accept* **else** *reject*;

We will define a graph structure — using the items sets I_j — that shows the structure of the recognition process somewhat clearer. An example of such a graph is shown in Figure 2. The graph illustrates *how* items have been added to the item sets I_j. The graph contains two types of nodes. *Item set nodes* (circles) are labelled with subsets of some item set I_j. Such nodes are connected via *symbol nodes* (squares). These show how a new set of items has been derived from a given set of items by scanning a particular symbol. The root node, denoted u_0, is labelled with the set of items *predict*(S). As it happens to be the case that the sentence starts with a *NP*, u_0 is

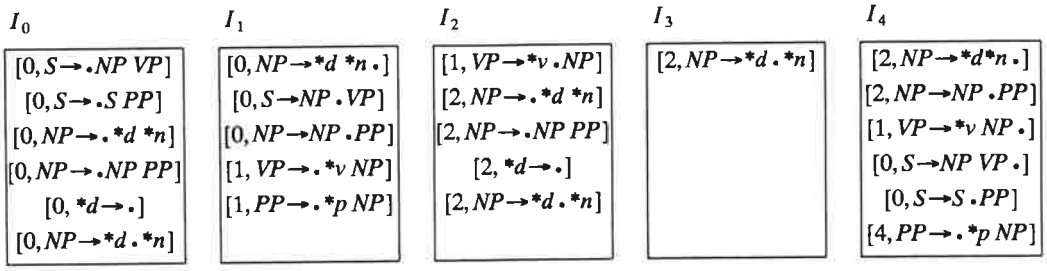

Figure 1: completed item sets I_j for the sentence *John saw a lion*

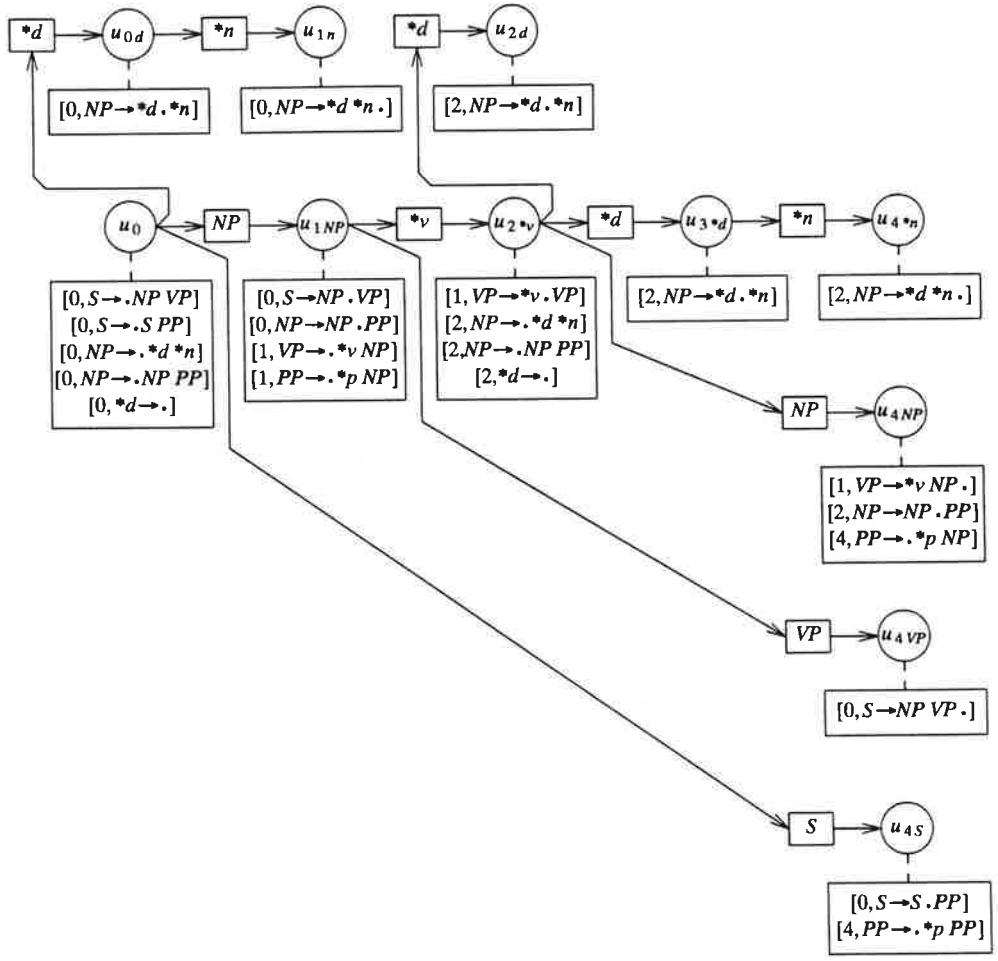

Figure 2: A completed Earley graph for the sentence *John saw a lion*

connected — via an *NP* symbol node — to an item set node labelled with

- the applicable items from the label of u_0 in which the dot has been carried over an *NP*,
- the *predictor* sets of those items.

Every symbol node in the graph, in similar fashion, represents a step in the recognition process, in which the items of its sucessor node have been derived from the items of its predecessor.

Formally, we introduce the *set of item set nodes* $U = \{u_0\} \cup \{u_{j,X} \mid 0 \le j \le n, X \in V\}$. We may write U_j for subsets of U with fixed j. Each $u \in U$ is labeled with a set of items $ITEMS(u)$, defined by

$$ITEMS(u_0) \stackrel{\text{def}}{=} \{[0, C \rightarrow \bullet\gamma] \mid C \rightarrow \bullet\gamma \in predict(S)\}$$

$$ITEMS(u_{j,X}) \stackrel{\text{def}}{=} \bigcup_{[i, A \rightarrow \alpha X \bullet \beta] \in I_j} predictor([i, A \rightarrow \alpha X \bullet \beta], j)$$

As a consequence, $I_j = \bigcup_{u \in U_j} ITEMS(u)$ for each j. I.e., taking the union of the vertically aligned $ITEMS(u)$ boxes in Figure 2 yields the item sets of Figure 1. Items of the form $[i, A \rightarrow \bullet\beta]$ may appear in more than one $ITEMS(u_{j,X})$ set for the same j, as they have been added to I_j more than once in different *predictor* sets. The *set of symbol nodes*, denoted Y, is defined by

$$Y \stackrel{\text{def}}{=} \{y_{j,X} \mid \exists\, u_{j,X} \in U \setminus \{u_0\} : ITEMS(u_{j,X}) \neq \emptyset\}\,.$$

That is, for each $u_{j,X}$ with a non-empty set of items, a node $y_{j,X}$ is introduced. Symbol nodes are labeled with their second index: $SYMBOL(y_{j,X}) = X$. Note that $y_{j,X} \in Y$ iff $X \Rightarrow^* a_{i+1} \cdots a_j$ for some i. This fact, and the particular i, will be indicated by the edges that are contained in the graph. If $y_{j,X} \in Y$ then there is some item $[k, A \rightarrow \alpha X \bullet \beta] \in ITEMS(u_{j,X})$. Also, there must be some v with $[k, A \rightarrow \alpha \bullet X\beta] \in ITEMS(v)$. For each such v, an edge $v \rightarrow y_{j,X}$ is added to set of edges E. Finally, $y_{j,X} \rightarrow u_{j,X}$ is also added to E.

Consequently, $X \Rightarrow^* a_{i+1} \cdots a_j$ iff there are $v \in U_i$, $u \in U_j$, $y \in Y$ such that $v \rightarrow y$, $y \rightarrow u \in E$ and $SYMBOL(y) = X$. Thus we have formalized the underlying intuition: one can move from $v \in U_i$ to $u \in U_j$ by scanning a symbol X that produces $a_{i+1} \cdots a_j$.

In [Sikkel90] it is shown in some more detail how the graph can be constructed directly, rather than from the sets I_j.

4 Tomita's algorithm

We will describe a variant of Tomita's algorithm that allows close comparison with Earley's algorithm. At the end of the section we will show how a *recognized item set* can be defined for an LR parser. The *RIS* of the variant of Tomita's algorithm constructed here is identical to RIS_E of Earley's algorithm.

Tomita's algorithm is a generalization of the well-known LR parsing technique. In fact Tomita was not the only, nor even the first person to consider "generalized LR parsing", but the algorithm is known under his name because his book is very easy to read and enjoys widespread circulation. A more theoretical but rather more complex approach can be found in [vdSteen88].

An LR parser scans a string from left to right while constructing a rightmost derivation of the parse tree. LR parsing is introduced in any textbook on parsing, e.g. [Aho77], [Harrison78]. Traditional LR parsing requires at every step that the next action be uniquely determined; in generalized LR parsing, table entries may contain multiple actions. If such an ambiguity arises, all actions are carried out on separate copies of the parse stack. For efficiency, the different parse stacks are merged into one graph-structured stack; if different parses share common sub-parts, the corresponding actions have to be carried out only once.

In [Tomita85] an SLR(1) parser is used. For a close comparison with Earley, it is better to use an LR(0) parser. That is, the set of possible actions depends only on the state of the parser, irrespective of the next input symbol. An LR(0) parse table for G is shown in Figure 3. This LR(0) table has the following interesting features:

- There is a column *dotted rules*. These sets of dotted rules play a vital role in the construction of the table (to be discussed shortly). With this extra column, we call it an *annotated* parse table.
- The *action* column gives the actions to be carried out in a particular state; "sh" means *shift* and "re p" mean *reduce* by production p. If the action is *shift*, the next state is determined

state	dotted rules	action	*d	*n	*v	*p	NP	PP	VP	S	$
0	$S' \rightarrow .S\ \$$ $S \rightarrow .NP\ VP$ $S \rightarrow .S\ PP$ $NP \rightarrow .*d\ *n$ $NP \rightarrow .NP\ PP$ $*d \rightarrow .$	re3 sh	3	–	–	–	2	–	–	1	–
1	$S' \rightarrow S.\$$ $S \rightarrow S.PP$ $PP \rightarrow .*p\ NP$	sh	–	–	–	6	–	5	–	–	4
2	$S \rightarrow NP.VP$ $NP \rightarrow NP.PP$ $VP \rightarrow .*v\ NP$ $PP \rightarrow .*p\ NP$	sh	–	–	7	6	–	9	8	–	–
3	$NP \rightarrow *d.*n$	sh	–	10	–	–	–	–	–	–	–
4	$S' \rightarrow S\ \$.$	acc	–	–	–	–	–	–	–	–	–
5	$S \rightarrow S\ PP.$	re2	–	–	–	–	–	–	–	–	–
6	$PP \rightarrow *p.NP$ $NP \rightarrow .*d\ *n$ $NP \rightarrow .NP\ PP$ $*d \rightarrow .$	re3 sh	3	–	–	–	11	–	–	–	–
7	$VP \rightarrow *v.NP$ $NP \rightarrow .*d\ *n$ $NP \rightarrow .NP\ PP$ $*d \rightarrow .$	re3 sh	3	–	–	–	12	–	–	–	–
8	$S \rightarrow NP\ VP.$	re1	–	–	–	–	–	–	–	–	–
9	$NP \rightarrow NP\ PP.$	re5	–	–	–	–	–	–	–	–	–
10	$NP \rightarrow *d\ *n.$	re4	–	–	–	–	–	–	–	–	–
11	$PP \rightarrow *p\ NP.$ $NP \rightarrow NP.PP$ $PP \rightarrow .*p\ NP$	re6 sh	–	–	–	6	–	9	–	–	–
12	$VP \rightarrow *v\ NP.$ $NP \rightarrow NP.PP$ $PP \rightarrow .*p\ NP$	re7 sh	–	–	–	6	–	9	–	–	–

Figure 3: an annotated LR(0) parse table

by the symbol that has been shifted. If there is no entry, the branch of the stack can be discarded.

- An extra production $S' \rightarrow S\ \$$ has been added; it is presumed that the input sentence is followed by an end-of-sentence-marker $\$$. The end-of-sentence marker must be shifted explicitly, in order to to determine that the sentence has really finished before it is accepted. Clearly, $S' \Rightarrow^* a_1 \cdots a_n\$$ iff $S \Rightarrow^* a_1 \cdots a_n$.

How is such a table constructed? The states represent sets of dotted rules of the form $A \rightarrow \alpha \bullet \beta$. (In LR literature, a dotted rule is called an *LR(0) item*, but we prefer to avoid the word "item" so as not to create confusion with items of the form $[i, A \rightarrow \alpha \bullet \beta] \in I_j$u. We start with state s_0, which contains $PREDICT(S')$. If a state contains a dotted rule $A \rightarrow \alpha \bullet$, the action "reduce $A \rightarrow \alpha$" is added to the possible actions. In s_0, production (3) $*det \rightarrow \varepsilon$ can be reduced. For every symbol that appears directly after the dot in a dotted rule, we need another state to go to. From state s_0, we need states to go to for the symbols S, NP and $*det$. States s_1, s_2 and s_3 are defined accordingly. If, for example, a noun phrase is parsed, the dot is carried over the NP symbol where applicable, hence $S \rightarrow NP \bullet VP$, $S \rightarrow NP \bullet PP \in s_2$. Additionally, for every $A \rightarrow \alpha \bullet X\beta$ we add $PREDICT(X)$ to the same state, e.g., $VP \rightarrow \bullet *verb\ NP$ and $PP \rightarrow \bullet *prep\ NP \in s_2$. This process is

continued until no new states can be added.

We will exemplify the maintenance of the graph structured stack by looking at a few instances during the parsing of the sentence *John saw a lion at the zoo.* After having processed *John saw a lion*, we have a stack as shown in Figure 4(a). The next action could be *reduce* $VP \rightarrow *verb\ NP$ or a *shift*. As a matter of policy, all possible reduce actions are carried out before the next *shift* is done. So the reduction $VP \rightarrow *verb\ NP$ is carried out, yielding 4(b). Figure 4(c) shows the stack after another reduction $S \rightarrow NP\ PP$. There are no more reductions and both branches do a *shift*. As it turns out, the next symbol **prep* yields state 6 from both states 1 and 12, so we can join the branches again, yielding 4(d).

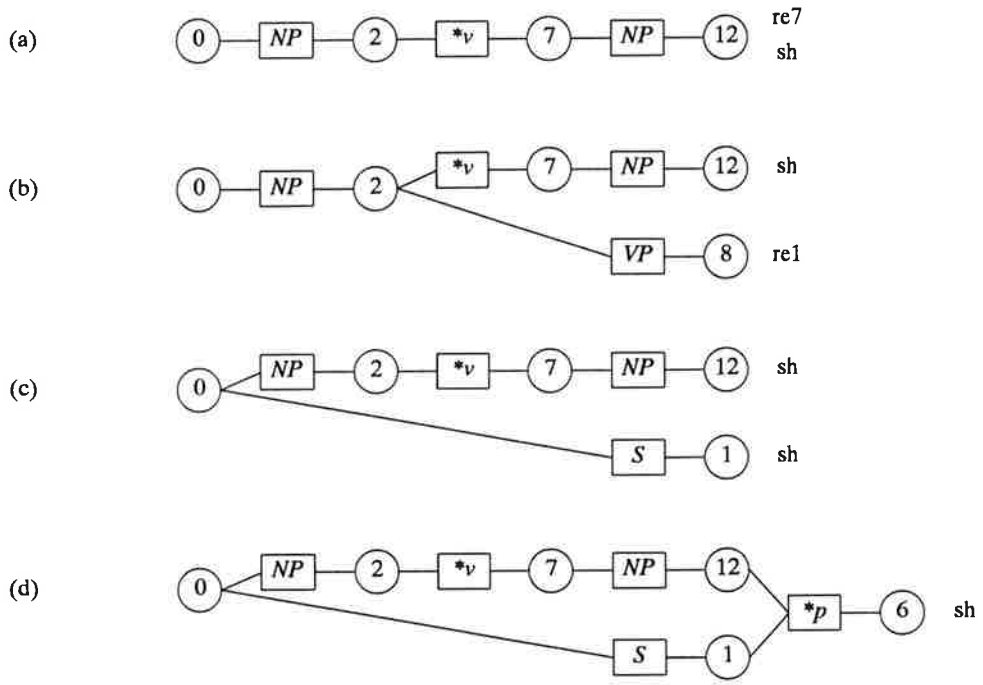

Figure 4: managing a graph structured stack

The stack of Tomita's parsing algorithm can be described as a bipartite directed graph $\Gamma = (U, Y; E)$. The set $U = U_0 \cup \cdots \cup U_n$ contains *state nodes*. A state node u is in U_j if the state has been reached after reading the first j input symbols. A label $STATE(u)$ gives the state number corresponding to that particular vertex. The set of dotted rules of that particular state is given by $rules(STATE(u))$. The set Y contains symbol nodes, as in the Earley graph. For a more detailed description of how the algorithm maintains the graph structured stack see [Tomita85].

We will make three small changes to Tomita's stack. Firstly, we reverse the direction of the arrows as compared to the original Earley stack. This should not upset the reader at all, as we have not shown any arrows so far. Secondly, when a reduction is carried out, there is no need to delete the part of the stack that is being reduced. We can simply leave it in the graph and start a new branch from the appropriate state node. This does not change the algorithm in any way, only the presentation of what a graph structured stack looks like is different.

Last, and most important, we will label the state nodes with a set of items, $ITEMS(u)$, containing items of the form $[i, A \rightarrow \alpha \bullet \beta]$ for each dotted rule $A \rightarrow \alpha \bullet \beta$ in $rules(STATE(u))$. The intention is clear: if $[i, A \rightarrow \alpha \bullet \beta] \in ITEMS(u)$ and $u \in U_j$, then $[i, A \rightarrow \alpha \bullet \beta, j] \in RIS$, and reversed. Items $[i, A \rightarrow \alpha \bullet \beta]$ are added to $ITEMS(u)$ in the following way. For each $u \in U_j$:

- if $y \rightarrow u \in E$ and $SYMBOL(y) = X$, then for each v such that $v \rightarrow y \rightarrow u$ and for every $[i, A \rightarrow \alpha \bullet X\beta] \in ITEMS(v)$ add an item $[i, A \rightarrow \alpha X \bullet \beta]$ to $ITEMS(u)$. It is easily verified that

this can only happen if indeed $A \rightarrow \alpha X \bullet \beta \in rules(STATE(u))$.

- For each $A \rightarrow \bullet \gamma \in STATE(u)$, add an item $[j, A \rightarrow \bullet \gamma]$ to $ITEMS(u)$.

As an example, the annotated stack of the sentence *John saw a lion* is shown in Figure 5.

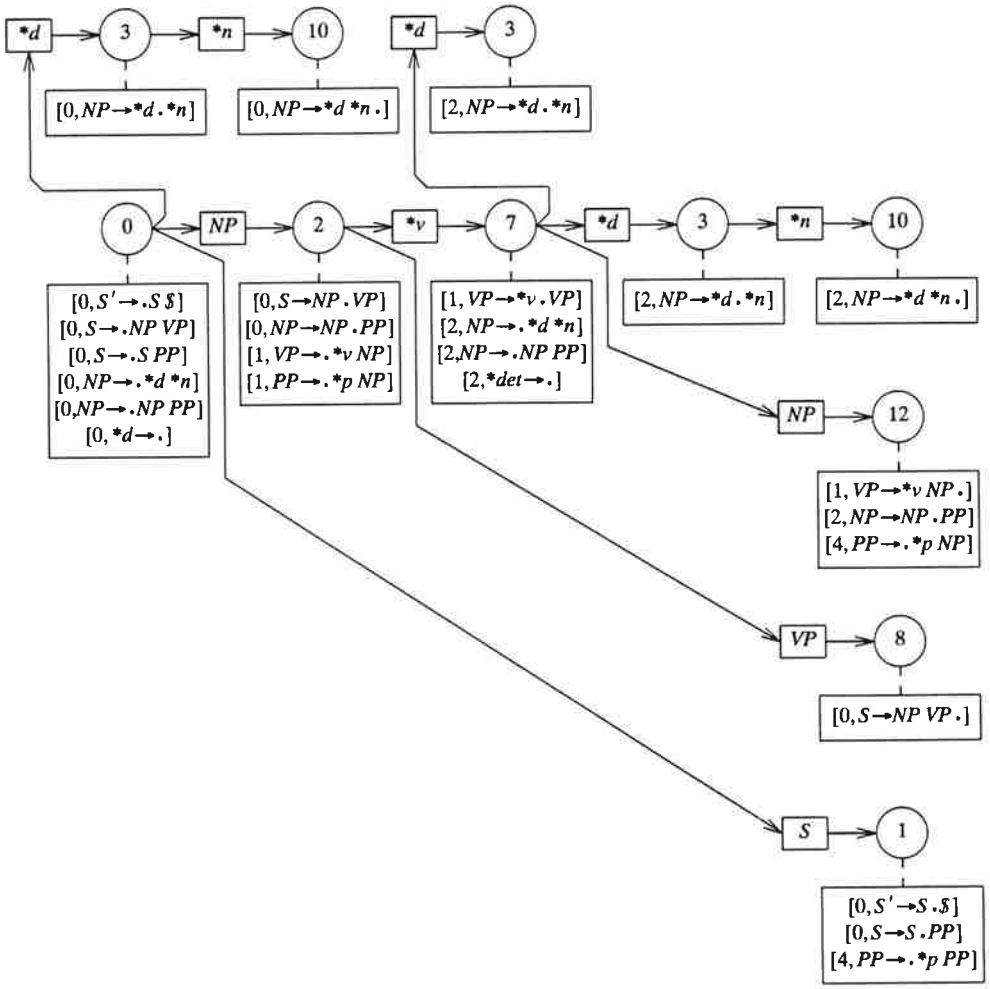

Figure 5: An annotated Tomita stack for *John saw a lion*

It is not a coincidence that Figure 5 and Figure 2 are very similar. The most important difference (not present in the example graphs) is the fact that one $u \in U_j$ in the Tomita graph may correspond to multiple $u_{j,X}$'s in the Earley graph. This will be discussed in more detail in the next section. It is has no consequences for the collection of items represented in the graph.

In order to relate the Earley item set I_j and the Tomita item sets $ITEMS(u)$ we have to sweep one more detail under the rug: We introduced a supplementary production $S' \rightarrow S\ \$$ for the LR parser. If we discard items $[i, S' \rightarrow \alpha \bullet \beta]$, or alternatively, if we run Earley's algorithm on the augmented grammar, we find the equality

$$RIS_E = \bigcup_{u \in U} ITEMS(u) .$$

5 A close comparison of Earley and Tomita

Having discussed Earley and Tomita in the previous sections, we are ready to precisely establish their relation.

The graphs of Graph Structured Earley (GSE) and Annotated LR(0) Tomita (AT) are *almost* identical (in a sense that will be made more precise shortly). Furthermore, both algorithms are naturally divided into steps $0 \cdots n$ such that Y_j and U_j are constructed in step j. The main difference between these two algorithms is the way in which each individual step is carried out. Before we look into the construction of the parse graph, we will review the differences between the GSE and AT graph and argue that they are not essential. We may safely ignore the differences caused by the fact that AT uses a grammar $G' = G \cup \{S' \rightarrow S\,\$\}$. A more substantial difference is that the U_j(GSE) nodes are classified according to the *symbol* that has been recognized, whereas the U_j(AT) nodes are classified according to the *state* of the parser. Each state can be associated with a recognized symbol, as each state contains dotted rules of the form $A \rightarrow \bullet\gamma$ and $A \rightarrow \alpha X \bullet \beta$ for some $X \in V$. Unfortunately, there can be several states with the same symbol X. Furthermore, as for every $u \in U_j$ in either algorithm there is exactly one $y \in Y_j$, we also find that for every $y \in Y_j$(GSE) there is a corresponding set of symbol nodes in Y_j(AT).

The next question, to be answered immediately, is: *how much larger* is the AT graph compared to the GSE graph? We will argue that the difference in size is small enough to be ignored. For that purpose, we define a refinement of GSE, called GSE+. We define symbol nodes in Y_j according to *occurrences* of symbols in right-hand sides of productions, and item nodes accordingly. Grammar G', with different occurrences of symbols numbered differently is

(0)	$S \rightarrow S_0\,\$$	(4)	$NP \rightarrow *det_1\ *noun_1$
(1)	$S \rightarrow NP_1\ VP_1$	(5)	$NP \rightarrow NP_2\ PP_2$
(2)	$S \rightarrow S_1\ PP_1$	(6)	$PP \rightarrow *prep_1\ NP_3$
(3)	$*det \rightarrow \varepsilon$	(7)	$VP \rightarrow *verb_1\ NP_4$

In Figure 2 a GSE graph for the sentence *John saw a lion* was shown. In order to change it into a GSE+ graph, we would have to make the following changes:

- The node $u_{1,NP}$ must be split up into u_{1,NP_1} and u_{1,NP_2}, similarly for $y_{1,NP}$.
- The node $u_{4,NP}$ must be split up into u_{4,NP_4} and u_{4,NP_2}, similarly for $y_{4,NP}$.

The administrative work in creating new nodes and arrows and the multiplicity of *predicted* items may increase slightly. In the worst case with a factor equal to the largest number of occurrences of one particular grammar symbol; it will be rather hard to notice a difference in performance.

We have associated each node in U_j(GSE) with a subset of U_j(AT); in the same way we can associate each node in U_j(AT) with a subset of U_j(GSE+). It is not hard to prove that for arbitrary CFG's the number of states in LR(0) Tomita is bounded by the number of different occurrences of symbols in right-hand sides of productions. In particular, for any grammar, any input and any j the inequality

$$|U_j(\text{GSE})| \leq |U_j(\text{AT})| \leq |U_j(\text{GSE+})|$$

holds. A similar inequality holds for Y_j. As we have argued that the difference in size between a GSE and GSE+ graph is irrelevant, the different in size between either graph and an AT graph is irrelevant too. Thus we have trivialized the differences between GSE and AT graphs, and we may consider them almost identical indeed.

Let us now reconsider in some detail the operation of both algorithms. GSE creates the $ITEMS(u_{j,X})$ sets (and, as a by-product, nodes $y_{j,X}$ and $u_{j,X}$ by scanning the appropriate $ITEMS(u_{i,X'})$ sets for $0 \leq i \leq j$. For a *scan* operation, a scan over $I_{j-1} = \bigcup_{u \in U_{j-1}} ITEMS(u)$ yields all items $[i, A \rightarrow \alpha \bullet a\beta]$ with $a = a_j$ that cause items $[i, A \rightarrow \alpha a \bullet \beta]$ to be added to u_{j,a_j}. The *predict* has been incorporated into the *scan* and *complete* operations, but the work still has to be carried out. It is easy to verify that *predict*, like *scan*, takes $O(j)$ time for each value of j, i.e. $O(n^2)$ time for the algorithm. *Complete* involves some more work: a scan over I_j reveals the various items $[k, B \rightarrow \gamma \bullet]$ that take part in a complete step. For each k involved, I_k must be scanned for items $[i, A \rightarrow \alpha \bullet B\beta]$. It is possible that I_j contains items $[k, A \rightarrow \alpha \bullet \beta]$ for multiple

values of k. In the worst case, their number is proportional to j, hence *complete* is bounded by $O(n^2)$, and the algorithm by $O(n^3)$ time. If the size of the item sets does not (significantly) expand for larger j, one *complete* step takes only $O(n)$ time and the algorithm finishes in $O(n^2)$ time. Some clever indexing can help to reduce the number of items that is really scanned during a *scan* or *complete*; but the essence of Earley's approach is scanning completed item sets in order to compute a new item set.

AT, while producing (almost) the same graph, operates in quite a different fashion. *Shift*, like *scan*, is straightforward. For each state node $u_{j-1,s} \in U_{j-1}$ that allows a shift, nodes y_{j,a_j} and $u_{j,s'}$ are introduced. As a by-product, $ITEMS(u_{j,s'})$ is created. *Predict* has no counterpart in the AT algorithm, because the work has been carried out compile-time in the construction of the parse table! *Reduce*, as counterpart of *complete*, is the more complex operation. Let's call the node to be reduced the *reductand* and the (one or more) nodes, at which the stack should be continued with the left-hand side symbol, the *ancestors*. If the number of edges is not significantly larger than the number of nodes, this can be done in $O(n)$ time. Because the size of U_j, is bounded by a constant, $O(n)$ suffices for all reductions in U_j. If there is no ambiguity, a *reduce* step takes only constant time, and AT is of order $O(n)$.

A worst case analysis is somewhat more complex. For a reductand in U_j, the number of ancestors is at most $O(n)$. The way in which the ancestors are searched for by the Tomita algorithm is: enumerating all paths from the reductand back to its ancestors. With dense ambiguity, this may take $O(n^\rho)$ steps when the production has a right-hand side with size ρ. As [Kipps89] has pointed out, there are search algorithms with lower worst case complexity bounds (but much more overhead in average cases). It is possible to find the ancestor set in $O(n^2)$ time. Thus, a variant of Tomita's algorithm can be constructed that is guaranteed to *recognize* a sentence in $O(n^3)$. When it comes to *parsing*, i.e., a shared forest is also required, the argument no longer holds, because a node in the shared forest may have $O(n^\rho)$ sub-nodes, corresponding to the enumeration of all paths from an ancestor to the reductand [Johnson90].

In GSE, the task of finding the ancestors of some $u \in U_j$ in a densely ambiguous sentence poses no particular problem. It is done in an $O(n^2)$ scan over the items. On the other hand, Earley's algorithm recognizes, rather than parses. Johnson's argument applies to Earley's algorithm as well: if a shared forest is constructed with $O(n^\rho)$ sub-nodes at some nodes, this takes $O(n^{\rho+1})$ time. A parse can be constructed in $O(n^3)$ time, however, if a more packed representation of a parse forest is used, as in [Leermakers91]. Thus a cubic-time Earley parser can be constructed. On the other hand, combined with Kipps' $O(n^2)$ *reduce* operation, a Tomita-like parser can be constructed that is guaranteed to operate in cubic time.

There is no theoretical evidence that one algorithm is significantly better than the other. Much depends on the grammar. Tomita will win on "easy" grammars, while Earley will work better on "difficult" grammars. In [Tomita86] it is claimed that natural language grammars are "easy", that is, *almost* LR and *almost* ε-free. For the examples in his book, Tomita's algorithm is 5–10 times faster than Earley's algorithm and 2-3 times faster than the improved GHR variant of Earley's algorithm (not discussed here) [Graham80]. However, we do not know of any empirical study that has systematically tested Tomita's claim against a variety of natural language grammars.

Drawing the discussion to an end, it is clear that GSE is based on managing the items, whereas AT is based on managing the graph. Earley's algorithm is a simplification of GSE in which the graph is dropped altogether; LR(0) Tomita is a simplification of AT in which the items are left out completely. The strong points of Tomita — if the graph does not branch too much — are:

- The states correspond to precompiled *sets* of dotted rules. adding one state to the graph structured stack means adding several items to an item set in one stroke. The *predict* function has been eliminated because it is incorporated in the parse table.

- Retracing a path of fixed, short length in the graph is less work than scanning the possibly useful items in an item set.

The weak point of Tomita is:

- The performance is rather dependent on the simplicity of the parse graph. Dense ambiguity combined with large productions leads to extremely poor performance.

6 Parallel Bottom-up Tomtita

Having established the similarity of Earley and Tomita in sections 3–5, we can apply these results in the construction of a novel Parallel Bottom-up Tomita (PBT) algorithm, being the Tomita counterpart of a well-known parallelization of Earley. The PBT algorithm uses an array of processors $P_0 \cdots P_n$; each processor P_i parses those symbols X that produce a string $a_{i+1} \cdots a_j$.

A parallel bottom-up approach to Earley is possible if we enlarge the recognized item set to

$$RIS_{bu} = \{[i, A \rightarrow \alpha \bullet \beta, j] \mid \alpha \Rightarrow^* a_{i+1} \cdots a_j\}.$$

The restriction $S \Rightarrow^* a_1 \cdots a_i A\gamma$ for some $\gamma \in V^*$ is dropped. Now we can compute items $[i, A \rightarrow \alpha \bullet \beta, j]$ for some specific value of $j - i$ in parallel. If all items with $j - i < k$ have been computed before, we have enough information to compute the items with $j - i = k$. We compute item sets $I_{i,j}$, containing items of the form $[A \rightarrow \alpha \bullet \beta]$ such that $\alpha \Rightarrow^* a_{i+1} \cdots a_j$. The lay-out of Figure 6 suggests a parallel implementation: n processors P_i, each one computing table entries $I_{i,i} \cdots I_{i,n}$.

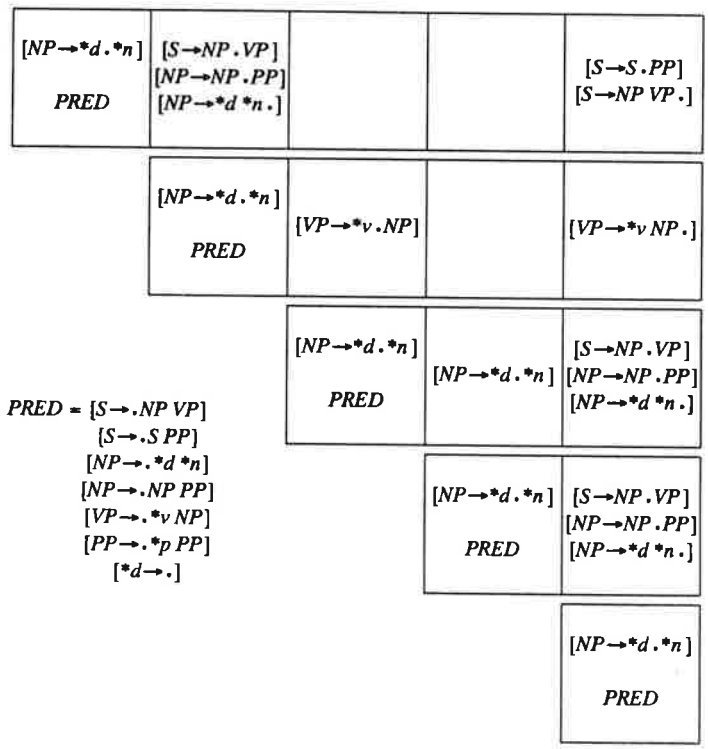

Figure 6: Tabular representation of RIS_{bu} for *John saw a lion*

The main difference to the original algorithm is in the computation of the item sets $I_{j,j}$. As it is not clear beforehand which items ought to be added by the *predictor*, we simply add $C \rightarrow \bullet\gamma$ for *every* production $C \rightarrow \gamma \in P$.

We have allocated a processor to each row in the upper triangular matrix of item sets. An equivalent parallel Earley algorithm is obtained when processors are allocated to columns of the matrix, as in [Nijholt90]. The division into rows is the more natural one for the Tomita case because each row can be computed in left-to-right fashion. For other parallel approaches see [Sijstermans86], who uses a rectangular grid of processors, one for each $I_{i,j}$, or [Yonezawa89] where processors are allocated to productions rather than positions in the sentence.

We will now develop a parallel LR-like parsing algorithm, such that processor P_i creates a *partial parse stack*, covering all partial parses $A \Rightarrow^* a_{i+1} \cdots a_j$ for $i \leq j \leq n$. While building such a parse stack, processor P_i can use other partial parses that have been delivered by processors $P_{i+1} \cdots P_n$. Eventually, all complete parses are delivered by P_0. Consider, for example, the prepositional phrase *in the zoo*. We will denote the corresponding grammar symbols as $\langle 4, *prep, 5\rangle$, $\langle 5, *det, 6\rangle$, $\langle 6, *noun, 7\rangle$. Place markers are necessary to identify the part of the sentence that corresponds with a recognized grammar symbol. Processor P_4 will be offered the marked terminal $\langle 4, *prep, 5\rangle$, and it constructs a parse stack

$$0 — \langle 4, *prep, 5\rangle — 6 .$$

In conventional LR parsing, state 6 would have contained *predict*(*NP*), because the parser should continue parsing the constituents of an *NP*. This is different in the parallel case. It is not the task of P_4 to actually parse the noun phrase, it simply waits till some other processor delivers a noun phrase $\langle 5, NP, j\rangle$. P_5 will pass a noun phrase $\langle 5, NP, 7\rangle$ in due course, and P_4 can shift an entire noun phrase, yielding

$$0 — \langle 4, *prep, 5\rangle — 6 — \langle 5, NP, 7\rangle — 11 .$$

A subsequent reduction $PP \rightarrow *prep\ NP$ constructs a marked symbol $\langle 4, PP, 7\rangle$ that can be passed to P_3 and further.

The processors are connected in a pipeline. P_i receives marked symbols from P_{i+1} and sees if they fit to its own stack. Meanwhile, symbols that could also be useful to processors with a lower rank number are passed on to P_{i-1}, supplemented with the the marked symbols renized by P_i.cognized by P_i. All processors use the same parse table. The construction of this table differs in two respects from the construction of Tomita's table:

- State 0 contains dotted rules $A \rightarrow \bullet \alpha$ for every $A \rightarrow \alpha \in P \cup \{S' \rightarrow S\ \$\}$,
- When a new state s' is constructed that is reachable via an entry $\langle s, X\rangle$ in the GOTO table, do *not* add *predict*(X) to the dotted rules of s'.

The parallel annotated LR(0) Tomita parse table for our example grammar is shown in Figure 7. Furthermore, there are the following differences in the construction of the partial parse stacks:

- On a *reduce* it is not allowed to prune the branch of the stack that is reduced. It may be still be needed at some moment in future, as synchronize on *shift* is no longer possible.
- The *shift* operation is not restricted to terminals, but may also be applied to appropriate nonterminal symbols that are passed down the pipeline.
- Two place markers are tagged onto each symbol. Without these place markers it would not be possible to decide at which position in the stack a new symbol is to be added.

The completed partial parse stack for P_1, parsing $\cdots$ *saw a lion in the zoo* is shown in Figure 8. In some cases, when a reduction is carried out, the GOTO table shows no applicable state. For example,

$$0 — \langle 1, *verb, 2\rangle — 7 — \langle 2, NP, 4\rangle — 12$$

can be reduced to

$$0 — \langle 1, VP, 4\rangle — ? \quad .$$

Absence of a next state in the GOTO table means that neither another symbol can be added after a *VP*, nor a further reduction is possible. This is a completed branch of the partial parse stack that serves no function, other than indicating that $\langle 1, VP, 4\rangle$ has been parsed. In Figure 9 we write C

state	*dotted rules*	*action*	**d*	**n*	**v*	**p*	*NP*	*PP*	*VP*	*S*	*$*
0	$S' \rightarrow .\, S\,\$$ $S \rightarrow .\, NP\ VP$ $S \rightarrow .\, S\ PP$ $NP \rightarrow .\, *d\ *n$ $NP \rightarrow .\, NP\ PP$ $VP \rightarrow .\, *v\ NP$ $PP \rightarrow .\, *p\ NP$ $*d \rightarrow .$	re3 sh	3	–	7	6	2	–	–	1	–
1	$S' \rightarrow S .\, \$$ $S \rightarrow S .\, PP$	sh	–	–	–	–	–	5	–	–	4
2	$S \rightarrow NP .\, VP$ $NP \rightarrow NP .\, PP$	sh	–	–	–	–	–	9	8	–	–
3	$NP \rightarrow *d .\, *n$	sh	–	10	–	–	–	–	–	–	–
4	$S' \rightarrow S\,\$.$	acc	–	–	–	–	–	–	–	–	–
5	$S \rightarrow S\ PP .$	re2	–	–	–	–	–	–	–	–	–
6	$PP \rightarrow *p .\, NP$	sh	–	–	–	–	11	–	–	–	–
7	$VP \rightarrow *v .\, NP$	sh	–	–	–	–	12	–	–	–	–
8	$S \rightarrow NP\ VP .$	re1	–	–	–	–	–	–	–	–	–
9	$NP \rightarrow NP\ PP .$	re5	–	–	–	–	–	–	–	–	–
10	$NP \rightarrow *d\ *n .$	re4	–	–	–	–	–	–	–	–	–
11	$PP \rightarrow *p\ NP .$	re6	–	–	–	–	–	–	–	–	–
12	$VP \rightarrow *v\ NP .$	re7	–	–	–	–	–	–	–	–	–

Figure 7: parallel annotated LR(0) parse table

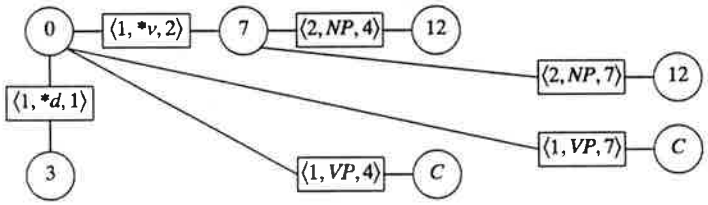

Figure 8: completed partial parse stack of P_1

for the state of such a completed branch. It is left to the reader to verify that an annotated version of PBT recognizes exactly the item set RIS_{bu}, when bottom-up Earley is run with the augmented grammar G'.

If all recognized triples are passed on to all other processors, there is an obvious communication bottleneck. In [Lankhorst91] it is analysed how most of the *junk communication*, — i.e., symbols not used by any subsequent processor — can be discarded with a simple filtering scheme. For example, if P_i receives a marked symbol $\langle i+1, X, j\rangle$ and $X \notin FOLLOW(a_{i+1})$ [Aho77], the marked symbol need not be sent on to P_{i-1}.

We have shown how a sentence can be recognized by the PBT algorithm. A parse forest can be constructed in a way similar to [Tomita86]. Each processor constructs a list of recognized symbols, with pointers to (c.q. labels of) its children. In case of ambiguity, a symbol has a list of groups of children. In order to allow the necessary administration, the label of a symbol X in the parse list of processor P_i is also passed with the marked symbol. We change the notation of marked symbols to $\langle i.p, X, j\rangle$, where p is the (i-)label of $\langle i, X, j\rangle$. The result for our sample grammar and input is shown in Figure 9. Note that the parse list in Figure 9 contains quite a few

symbol	*children*
⟨7.*a*, **det*, 7⟩	
⟨6.*a*, **det*, 6⟩	
⟨6.*b*, **noun*, 7⟩	
⟨6.*c*, *NP*, 7⟩	(6.*a*, 6.*b*)
⟨5.*a*, **det*, 5⟩	
⟨5.*b*, **det*, 6⟩	
⟨5.*c*, *NP*, 7⟩	(5.*b*, 6.*b*)
⟨4.*a*, **det*, 4⟩	
⟨4.*b*, **prep*, 5⟩	
⟨4.*c*, *PP*, 7⟩	(4.*b*, 5.*c*)
⟨3.*a*, **det*, 3⟩	
⟨3.*b*, **noun*, 4⟩	
⟨3.*c*, *NP*, 4⟩	(3.*a*, 3.*b*)
⟨3.*d*, *NP*, 7⟩	(3.*c*, 4.*c*)
⟨2.*a*, **det*, 2⟩	
⟨2.*b*, **det*, 3⟩	
⟨2.*c*, *NP*, 4⟩	(2.*b*, 3.*b*)
⟨2.*d*, *NP*, 7⟩	(2.*c*, 4.*c*)
⟨1.*a*, **det*, 1⟩	
⟨1.*b*, **verb*, 2⟩	
⟨1.*c*, *VP*, 4⟩	(1.*b*, 2.*c*)
⟨1.*d*, *VP*, 7⟩	(1.*b*, 2.*d*)
⟨0.*a*, **det*, 0⟩	
⟨0.*b*, **noun*, 1⟩	
⟨0.*c*, *NP*, 1⟩	(0.*a*, 0.*b*)
⟨0.*d*, *S*, 4⟩	(0.*c*, 1.*c*)
⟨0.*e*, *S*, 7⟩	(0.*c*, 1.*d*) (0.*d*, 4.*c*)

Figure 9: A complete parse list

useless symbols (not reachable from ⟨0.*e*, *S*, 7⟩). This is due to the rather clumsy structure of G. If we had used a grammar with production (3) $NP \rightarrow$ **noun*, rather than (3) **det* $\rightarrow \varepsilon$, all determiners of zero width would be absent in the table.

In [Lankhorst91], the PBT algorithm is analyzed in more detail and proven to be correct. Moreover, unlike Tomita, PBT can handle cyclic grammars as well. Analysis of a simulated parallel implementation with respect to a comparable implementation of Tomita's algorithm shows that using $O(n)$ processors in parallel reduces the time complexity with, roughly, $O(n^{0.5})$.

7 Conclusions

We have shown the structural similarity between Earley's algorithm and Tomita's algorithm using the Graph Structured Earley (GSE) and Annotated LR(0) Tomita (AT) variants. Earley and LR(0) Tomita have identical recognized item sets. Furthermore, GSE and AT construct annotated parse graphs that are almost identical This graph has been used for a detailed comparison of the performance of both algorithms. Noteworthy points are

- In "average" cases (no dense ambiguity), Tomita's algorithm should perform better because it has compiled some of the work that needs to be done in many instances in Earley's algorithm. The *predictor* function is not needed in Tomita's algorithm because it is incorporated in the parse table.
- The widely held belief (based on [Tomita86]) that the worst-case complexity of Tomita's algorithm is inferior when confronted with densely ambiguous grammars needs reconsideration:

- With some effort, a variant of Tomita's algorithm can be constructed that recognizes in $O(n^3)$ time, the same complexity as Earley's algorithm [Kipps90].
- When we regard Tomita as a parsing algorithm (which it actually is), rather than a recognition algorithm, a more than cubic time corresponds to a more than cubic number of sub-nodes in the shared parse forest. Earley's algorithm does not have this problem simply because it does not create a parse forest. Either parsing algorithm can be guaranteed to finish in cubic time, however, if a more packed representation of a shared forest is used.

Cubic Tomita, however, is only interesting as a theoretical concept. In practice it will generally be slower than ordinary Tomita, due to the much larger overhead in the *reduce* step.

As a fertile example of cross-breeding we have discovered a Parallel Bottom-up Tomita parser (PBT). In PBT, a sentence $a_1 \cdots a_n$ is parsed by $n+1$ processors. Each processor P_j makes a partial graph structured parse stack covering the suffix $a_{j+1} \cdots a_n$ of the sentence. Nonterminals that have been parsed by P_{j+k} can be used as atomic symbols by P_j. The PBT algorithm is analysed in depth in [Lankhorst91].

Acknowledgements

I am grateful to Anton Nijholt for constructive comments on earlier versions of this paper, to René Leermakers for some comments on the complexity of Tomita's algorithm, and to two anonymous referees for suggesting some improvements in presentation.

References

[Aho77] A.V. Aho, J.D. Ullman, *Principles of Compiler Design*, Addison-Wesley, Reading, Mass. (1977).

[Chiang84] Y.T. Chiang, K.S. Fu: Parallel Parsing Algorithms and VLSI implementations for Syntactic Pattern Recognition, *Transactions on Pattern Analysis and Machine Intelligence* **PAMI-6**, 3 (1984) 302–314.

[Earley68] J. Earley: *An Efficient Context-Free Parsing Algorithm*, Ph.D. Thesis, Computer Science Dept., Carnegie-Mellon University, Pittsburgh, PA (1986).

[Earley70] J. Earley: An Efficient Context-Free Parsing Algorithm, *Communications of the ACM* **13** (1970) 94–102.

[Graham80] S.L. Graham, M.A. Harrison, W.L. Ruzzo: An Improved Context-Free Recognizer, *Transactions on Programming Languages and Systems* **2** (1980) 415–462.

[Harrison78] M.A. Harrison: *Introduction to Formal Language Theory*, Addison-Wesley, 1978.

[Johnson89] M. Johnson: The Computational Complexity of Tomita's Algorithm, *Proc. Int. Workshop on Parsing Technologies*, Pittsburgh (1989) 203–208.

[Kipps89] J.R. Kipps: Analysis of Tomita's Algorithm for General Context-Free Parsing, *Proc. Int. Workshop on Parsing Technologies*, Pittsburgh (1989) 193–202.

[Lankhorst91] M.M. Lankhorst, K. Sikkel: *PBT: A Parallel Bottom-up Tomita Parser*, Memoranda Informatica 91-69, Department of Computer Science, University of Twente (1991).

[Leermakers91] R. Leermakers: Non-deterministic Recursive Ascent Parsing, *Proc. 5th European Chapter of the ACL*, Berlin (1991) 63–68. A functional LR-parser, to appear in *Information Processing Letters* (1991).

[Nijholt90] A. Nijholt: The CYK Approach to Serial and Parallel Parsing, *Proc. Seoul Int. Conf. on Natural Language Processing*, Seoul National University (1990) 144–155.

[Nijholt91] A. Nijholt: The Parallel Approach to Context-Free Language Parsing, in: U. Hahn, G. Adriaens (Eds.), *Parallel Models of Natural Language Computation*, Ablex Publishing Co., Norwood, N.J. (1991).

[Sijstermans86] F.W. Sijstermans: *Parallel parsing of context-free Languages*, Doc. no. 202, ESPRIT project 415 sub A, Philips Research Laboratories, Eindhoven, the Netherlands (1986).

[Sikkel90] K. Sikkel: *Cross-Fertilization of Earley and Tomita*, Memoranda Informatica 90-69, Department of Computer Science, University of Twente (1990).

[vdSteen88] G.J. van der Steen: A program generator for recognition, parsing and transduction with syntactic patterns, *CWI Tract* **55**, Centre for Mathematics and Computer Science, Amsterdam (1988).

[Tomita85] M. Tomita: *Efficient Parsing for Natural Language*, Kluwer Academic Publishers, Boston, Mass. (1985).

[Yonezawa89] A. Yonezawa, I. Ohsawa: Object-Oriented Parallel Parsing for Context-Free Grammars, in: A. Yonezawa (ed.) *ABCL: an Object- Oriented Concurrent System*, The MIT Press (1989).