Universität Stuttgart
Fakultät Informatik

A New Parsing Strategy for Context-Free Grammars

Thomas Schöbel

Computer Science Report
No. 7/91 May 1991

1st printing May 1991

Dekanat der Fakultät Informatik
Universität Stuttgart
Breitwiesenstr. 20-22, D-7000 Stuttgart 80, Germany

A New Parsing Strategy for Context-Free Grammars

Extended Abstract from a currently prepared paper

Thomas Schöbel

2 May 1991

Abstract

We present a family of new top-down parsing algorithms for arbitrary context-free grammars based on the notion of a "tree of possible left-derivations". When coded in a special way, this tree can be compressed to a graph. While the time complexity on uniform RAMs is $O(n^3)$ in worst case, we need $O(n^2)$ space. We show that the asymptotic time complexities are equal to Earley's algorithms, and give some arguments to show our algorithm will be better in almost all cases.

1 Introduction

Since Cocke, Younger and Kasami (CYK) presented the first polynomial algorithms for recognition of context-free languages, a lot of investigation has been done in the field of recognition and parsing. Amoung the various parsers, Earley's famous algorithm [Ea70] is the most commonly used. Graham, Harrison and Ruzzo [GHR80] have shown that the CYK and Earley methods are "almost" the same, based on an upper triangle matrix, principally containing triples as elements.

Another approach is Tomita's method [To86], which uses a graph for compressing multiple stacks to a smaller structure. In contrast to our proposed algorithm, he takes the various bottom-up algorithms (SLR, $LR(0)$, $LR(k)$, $LALR$ ...) as base for compressing the exponentially growing computation tree.

Our method is founded on a theoretical basis, from which we may get multiple algorithms. So we think the primary contribution of this paper to the field will be another view of parsing. At current state, however, we feel also that there are gains when compared to Earley's algorithm and many improvements which could be found in the literature (e. g. [BPS75],[GHR80],[KA89],[Lei90]). We are currently working on a detailed discussion of the related topics.

2 Basic Definitions, TPL and CTPL

We will use a similar notation as in [HU79]. Kleene-Closure will be denoted by an asterisk *. A context-free grammar G (or simply Grammar) is defined as $G = (\Sigma, N, P, S)$, where Σ is the terminal alphabet, N the nonterminal alphabet, $\Sigma \cap N = \emptyset$, $V := \Sigma \cup N$ the alphabet, $P \subseteq N \times V^*$ the set of productions, and $S \in N$ the start symbol. Terminals will be denoted as a, b, c, ..., nonterminals as A, B, C, ..., strings from V^* as α, β, γ, ..., the empty string is ϵ, and productions will be denoted as $X \to \gamma$.

We introduce the notion of a left-derivation by terms of relations: Let G be a Grammar, then the relation $\Rightarrow_L$ on V^* is defined by $\alpha \Rightarrow_L \beta$ if and only if $\alpha = \alpha_1 X \alpha_2$, $\beta = \alpha_1 \gamma \alpha_2$, $X \to \gamma \in P, \alpha_1 \in \Sigma^*$, $\alpha_2 \in V^*$. A left–derivation D_L is a (finite or infinite) sequence of strings $D_L(\alpha) := (\alpha_0, \alpha_1, \alpha_2, \ldots)$ where $\alpha_i \Rightarrow_L \alpha_{i+1}$, $\alpha_0 = \alpha$. This definition directly leads to the notion of a "Tree of Possible Left-Derivations": We just write the left-derivation relation as a tree.

Definition 2.1 *Let G be a Grammar.*
The Tree of Possible Left-Derivations (TPL) of a string $\alpha \in V^*$ is a (possibly infinite) tree $\dot{\mathrm{G}} = (\dot{\mathrm{V}}, \dot{\mathrm{E}})$ with root $\dot{\mathrm{r}}(\dot{\mathrm{G}}) := \dot{\mathrm{s}} \in \dot{\mathrm{V}}$, where $\dot{\mathrm{V}}$ is the set of nodes, labeled with strings $\mu : \dot{\mathrm{V}} \mapsto V^*$, $\mu(\dot{\mathrm{s}}) = S$, $\dot{\mathrm{E}} \subseteq \dot{\mathrm{V}} \times \dot{\mathrm{V}}$ is the set of edges, and

$$\forall_{\dot{\mathrm{v}} \in \dot{\mathrm{V}}} \forall_{\alpha \in V^*, \mu(\dot{\mathrm{v}}) \Rightarrow_L \alpha} : \exists_{\dot{\mathrm{w}} \in \dot{\mathrm{V}}, (\dot{\mathrm{v}}, \dot{\mathrm{w}}) \in \dot{\mathrm{E}}} : \mu(\dot{\mathrm{w}}) = \alpha$$

$$\forall_{(\dot{\mathrm{v}}, \dot{\mathrm{w}}_1), (\dot{\mathrm{v}}, \dot{\mathrm{w}}_2) \in \dot{\mathrm{E}}} : \dot{\mathrm{w}}_1 \neq \dot{\mathrm{w}}_2 \implies \mu(\dot{\mathrm{w}}_1) \neq \mu(\dot{\mathrm{w}}_2)$$

Note that we always write objects belonging to a tree or directed graph with a dot on it, for distinction of $\dot{\mathrm{V}}$ (the set of vertices) from V (the alphabet). Also, we regard a tree as a specialized directed graph being acyclic, fully connected, and every node having at most one predecessor. The root of a tree has no predecessors.

Intuitively, the TPL shows all possible left-derivations, starting from the root $\dot{\mathrm{s}}$ labeled with S. Siblings always mean different alternatives of derivations. In general, the TPL will be infinite in depth and growing exponentially in breadth.

It is clear that the concept of the TPL will not directly serve for efficient recognition or parsing, because of its excessive consuming of resources. Therefore, we introduce a more efficient coding of the TPL, called CTPL (Compressed Tree of Possible Left-Derivations). Every label α of the TPL is partitioned as follows:

$$\alpha = xX\delta$$

where $x \in \Sigma^*$ is called the (terminal) prefix, $X \in N \cup \{\epsilon\}$ the (nonterminal) infix and $\delta \in V^*$ the suffix. Additionally, when $X = \epsilon \implies \delta = \epsilon$, this partitioning is unique.

Let us look at a derivation step in the TPL starting at $\dot{\mathrm{v}} \in \dot{\mathrm{V}}$ with label $\mu(\dot{\mathrm{v}}) = \alpha = xX\delta$: Let $X \to \gamma$ be a production, then $xX\delta \Rightarrow_L x\gamma\delta$ where $x\gamma\delta$ labels a son of $\dot{\mathrm{v}}$. Derivation is taking place only at the X; the other components (x and δ) remain unchanged in a derivation step (a property of context-freeness).

The basic idea of the CTPL is to register only the modifications along a path in the TPL. This is accomplished by repeatedly partitioning of γ into components $x'X'\delta'$ by use of three different labeling functions. The same is also done with the direct derivation of X' and so on. As far as $X \neq \epsilon$, this will result in a sequence of derivation steps, where only the labeling differences from the predecessor need to be registered.

For the CTPL, we need a prefix labeling function π assigned to the edges of the tree to register the local differences of the terminal prefix x. The infix labeling function ι contains the nonterminal X, the suffix labeling function σ contains the rest δ. Both are assigned to the nodes of the tree.

As we shall see, every node $\dot{\mathrm{v}}$ in the TPL has a corresponding node $\dot{\mathrm{v}}'$ in the CTPL. To get the same full prefix of $\dot{\mathrm{v}}$, we have to concatenate all prefixes in the path in the CTPL from the root $\dot{\mathrm{s}}'$ to $\dot{\mathrm{v}}'$. Similarly, the full suffix can be achieved by concatenation of all suffixes along the path upwards to the root. So we find that the CTPL will contain the same information as the TPL.

However, there are difficulties: When deriving a sentential form, a γ may be empty or may consist only of terminals. In other words, the existence of a nonterminal X and therefore of a non-empty infix is not guaranteed, and further derivations would be impossible. In this case, we have to do some transformations on the tree, which will produce an equivalent subtree for the "dead end". Normally, the suffixes δ of other γ's contain nonterminals, which have to be expanded. So it is clear that our tree transformation has to activate these nonterminals.

Definition 2.2 *Let G be a grammar.*
A <u>Compressed Tree of Possible Left-Derivations</u> (<u>CTPL</u>) is a tree $\dot{\mathrm{G}} = (\dot{\mathrm{V}}, \dot{\mathrm{E}})$ with root $\dot{\mathrm{r}}(\dot{\mathrm{G}}) := \dot{\mathrm{s}}$ and labeling functions

$$\begin{array}{rll} \pi : \dot{\mathrm{E}} \mapsto \Sigma^* & & \textit{(prefix labeling)} \\ \iota : \dot{\mathrm{V}} \mapsto N \cup \{\epsilon\} & & \textit{(infix labeling)} \\ \sigma : \dot{\mathrm{V}} \mapsto V^* & & \textit{(suffix labeling)} \end{array}$$

satisfying the following conditions:

(1) $\iota(\dot{\mathrm{s}}) = S,\ \sigma(\dot{\mathrm{s}}) = \epsilon$

(2) $\forall_{\dot{\mathrm{v}} \in \dot{\mathrm{V}}}\ \forall_{\iota(\dot{\mathrm{v}}) \to \gamma \in P}\ \exists_{(\dot{\mathrm{v}}, \dot{\mathrm{w}}) \in \dot{\mathrm{E}}} \ : \ \gamma = \pi((\dot{\mathrm{v}}, \dot{\mathrm{w}}))\iota(\dot{\mathrm{w}})\sigma(\dot{\mathrm{w}})$

(3) $\iota(\dot{\mathrm{v}}) = \epsilon \implies \sigma(\dot{\mathrm{v}}) = \epsilon$

(4) $(\dot{\mathrm{u}}, \dot{\mathrm{v}})\,,\,(\dot{\mathrm{v}}, \dot{\mathrm{w}}) \in \dot{\mathrm{E}} \wedge \iota(\dot{\mathrm{w}})\sigma(\dot{\mathrm{w}}) = \epsilon$ *if and only if*
$\exists_{(\dot{\mathrm{u}}, \dot{\mathrm{b}}) \in \dot{\mathrm{E}}} \ : \ \pi((\dot{\mathrm{u}}, \dot{\mathrm{b}})) = \pi((\dot{\mathrm{u}}, \dot{\mathrm{v}}))\pi((\dot{\mathrm{v}}, \dot{\mathrm{w}}))z,\ \iota(\dot{\mathrm{b}}) = X',\ \sigma(\dot{\mathrm{b}}) = \delta',$
where $\sigma(\dot{\mathrm{v}}) = zX'\delta',\ z \in \Sigma^*,\ X' \in N \cup \{\epsilon\},\ \delta' \in V^*$

(5) $\dot{\mathrm{G}}$ *contains nothing else.*

Conditions (1) and (2) give an inductive definition of directly derivable nodes in the CTPL. Condition (3) ensures that existing nonterminals will always appear in ι rather than in δ. Condition (4) is the tree transformation, when some $\iota(\dot{\mathrm{w}})$ is ϵ. In this case, we generate a brother $\dot{\mathrm{b}}$ to the father $\dot{\mathrm{v}}$, which receives the suffix $\sigma(\dot{\mathrm{v}})$ for partitioning into three components. As we will see, $\dot{\mathrm{w}}$ and $\dot{\mathrm{b}}$ have the same meaning for denoting a derivation. Furthermore, note that $\dot{\mathrm{b}}$ may have an empty $\iota(\dot{\mathrm{b}})$ also; then condition (4) generates another "uncle" denoting the same derivation.

To show the equivalence of TPL and CTPL, we introduce the notions of total prefixes and total suffixes.

Definition 2.3 *Let $\dot{\mathrm{G}}$ be a CTPL with root $\dot{\mathrm{r}}(\dot{\mathrm{G}}) := \dot{\mathrm{s}}$ over a grammar G.*
Let $\hat{a}$ be a path $(\dot{\mathrm{s}}, \dot{\mathrm{v}}_1, \dot{\mathrm{v}}_2, \ldots, \dot{\mathrm{v}}_k, \dot{\mathrm{v}})$ from the root $\dot{\mathrm{s}}$ to a node $\dot{\mathrm{v}}$.
We define the following functions:

1. *The total prefix $\hat{\pi}(\hat{a})$ of path $\hat{a}$ is*

$$\hat{\pi}(\hat{a}) := \pi((\dot{\mathrm{s}}, \dot{\mathrm{v}}_1))\, \pi((\dot{\mathrm{v}}_1, \dot{\mathrm{v}}_2))\, \ldots \pi((\dot{\mathrm{v}}_{k-1}, \dot{\mathrm{v}}_k))\, \pi((\dot{\mathrm{v}}_k, \dot{\mathrm{v}}))$$

2. *The total suffix $\hat{\sigma}(\hat{a})$ of path $\hat{a}$ is*

$$\hat{\sigma}(\hat{a}) := \sigma(\dot{\mathrm{v}})\, \sigma(\dot{\mathrm{v}}_k)\, \ldots \sigma(\dot{\mathrm{v}}_1)\, \sigma(\dot{\mathrm{s}})$$

3. *The sentential form $\hat{\phi}(\hat{a})$ of path $\hat{a}$ is*

$$\hat{\phi}(\hat{a}) := \hat{\pi}(\hat{a})\, \iota(\dot{\mathrm{v}})\, \hat{\sigma}(\hat{a})$$

When dealing with trees (rather than later with less restricted graphs), every node $\dot{\mathrm{v}}$ has a unique path $\hat{a}$ from the root $\dot{\mathrm{s}}$ to $\dot{\mathrm{v}}$. Therefore, we can write $\hat{\phi}(\dot{\mathrm{v}})$ as a short form instead of explicitly giving the path $\hat{a}$ as argument for $\hat{\phi}(\hat{a})$.

Lemma 2.4 *Let $\dot{\mathrm{G}}$ be a CTPL with root $\dot{\mathrm{r}}(\dot{\mathrm{G}}) := \dot{\mathrm{s}}$. Let $\dot{\mathrm{u}}, \dot{\mathrm{v}}, \dot{\mathrm{w}}, \dot{\mathrm{b}} \in \dot{\mathrm{V}}$ satisfying condition (4) of definition 2.2. Then*

$$\hat{\phi}(\dot{\mathrm{w}}) = \hat{\phi}(\dot{\mathrm{b}})$$

Proofs are omitted here.

Lemma 2.5 *Let $\dot{\mathrm{G}}$ be a CTPL. The relation $\cong$ on $\dot{\mathrm{V}}$, defined by*

$$\dot{\mathrm{v}} \cong \dot{\mathrm{w}} \ \textit{ if and only if } \ \hat{\phi}(\dot{\mathrm{v}}) = \hat{\phi}(\dot{\mathrm{w}})$$

is an equivalence relation.

The proof is trivial, for $\cong$ is directly defined by the equivalence relation $=$.
Now we are ready to show the equivalence of TPL and CTPL:

Theorem 2.6 *Let G be a grammar, $\dot{\mathrm{G}} = (\dot{\mathrm{V}}, \dot{\mathrm{E}})$ the CTPL and $\dot{\mathrm{G}}' = (\dot{\mathrm{V}}', \dot{\mathrm{E}}')$ the TPL of G. Then*

$$\begin{array}{ll} (1) & \forall_{\dot{\mathrm{v}} \in \dot{\mathrm{V}}} : \exists_{\dot{\mathrm{v}}' \in \dot{\mathrm{V}}'} : \hat{\phi}(\dot{\mathrm{v}}) = \mu(\dot{\mathrm{v}}') \\ (2) & \forall_{\dot{\mathrm{v}}' \in \dot{\mathrm{V}}'} : \exists_{\dot{\mathrm{v}} \in \dot{\mathrm{V}}} : \hat{\phi}(\dot{\mathrm{v}}) = \mu(\dot{\mathrm{v}}') \\ (3) & \forall_{(\dot{\mathrm{v}}', \dot{\mathrm{w}}') \in \dot{\mathrm{E}}'} : \exists_{(\dot{\mathrm{v}}, \dot{\mathrm{w}}) \in \dot{\mathrm{E}}} : \hat{\phi}(\dot{\mathrm{v}}) = \mu(\dot{\mathrm{v}}'),\ \hat{\phi}(\dot{\mathrm{w}}) = \mu(\dot{\mathrm{w}}') \end{array}$$

Informally, we can see that TPL and CTPL are equivalent in some sense. This means that the CTPL has more nodes becauses of its internal equivalence relation (lemma 2.5), so only the equivalence classes of the CTPL are equivalent to the TPL.

3 The CGPL

Until now we were dealing with structures which, in the worst case, may grow exponentially in breadth. Furthermore, there may be infinite recursions in the CTPL when the grammar is left-recursive. Lastly, both structures infinitely generate all sentences of the language $L(G)$.

The last problem can be handled easily by a parsing algorithm: We just have to prune all branches whose total prefixes do not match the given input string $x \in L(G)$. The former problems can be solved when turning the CTPL to a true directed graph called CGPL, allowing to melt nodes with common subtrees.

Some difficulties arises from dealing with labeled infinite trees. Hence, we need a definition of equivalence between infinite trees. Additionally, we want to compress trees to graphs, and therefore we need all the definitions also for graphs, regarding trees as a special case. For convenience, we will call the source (a single particular node) of a directed graph $\dot{\mathrm{G}}$ also the root by $\dot{\mathrm{r}}(\dot{\mathrm{G}})$.[1]

[1]Note that the root of a tree has no predecessors; in contrast, the root of a graph can have one. In our algorithm, the root of the main graph will never have predecessors, in contrast to subgraphs, where cycles may cause this.

Definition 3.1 *Let* $\dot{\mathrm{G}} = (\dot{\mathrm{V}}, \dot{\mathrm{E}})$ *be an arbitrary graph with root* $\dot{\mathrm{r}}(\dot{\mathrm{G}}) := \dot{\mathrm{s}}$.
The set of successors $\dot{\mathrm{R}}(\dot{\mathrm{v}})$ *of a node* $\dot{\mathrm{v}} \in \dot{\mathrm{V}}$ *is defined as*

$$\dot{\mathrm{R}}(\dot{\mathrm{v}}) := \dot{\mathrm{R}}^1(\dot{\mathrm{v}}) := \{\dot{\mathrm{w}} | (\dot{\mathrm{v}}, \dot{\mathrm{w}}) \in \dot{\mathrm{E}}\}$$

The extension of $\dot{\mathrm{R}}$ *to sets of nodes* $\dot{\mathrm{A}}$ *is defined by*

$$\dot{\mathrm{R}}(\dot{\mathrm{A}}) := \dot{\mathrm{R}}^1(\dot{\mathrm{A}}) := \bigcup_{\dot{\mathrm{v}} \in \dot{\mathrm{A}}} \dot{\mathrm{R}}(\dot{\mathrm{v}})$$

The set of i-reachable nodes $\dot{\mathrm{R}}^i(\dot{\mathrm{v}})$ *resp.* $\dot{\mathrm{R}}^i(\dot{\mathrm{A}})$ *is defined by*

$$\dot{\mathrm{R}}^i(\dot{\mathrm{v}}) := \dot{\mathrm{R}}(\dot{\mathrm{R}}^{i-1}(\dot{\mathrm{v}}))\,,\ \dot{\mathrm{R}}^i(\dot{\mathrm{A}}) := \bigcup_{\dot{\mathrm{v}} \in \dot{\mathrm{A}}} \dot{\mathrm{R}}^i(\dot{\mathrm{v}})$$

The set of sucessors is the same as the set of 1-reachable nodes. Note also that if the graph has cycles, a node $\dot{\mathrm{w}}$ may infinitely often occur in the sequence $\{\dot{\mathrm{R}}^i(\dot{\mathrm{v}})\}$. Membership $\dot{\mathrm{w}} \in \dot{\mathrm{R}}^i(\dot{\mathrm{v}})$ means that there exists a path of exactly length i from $\dot{\mathrm{v}}$ to $\dot{\mathrm{w}}$. We close the definition for $i = 0$ by defining $\dot{\mathrm{R}}^0(\dot{\mathrm{v}}) := \{\dot{\mathrm{v}}\}$, and also for negative i denoting predecessors by $\dot{\mathrm{R}}^{-1}(\dot{\mathrm{w}}) := \{\dot{\mathrm{v}} | (\dot{\mathrm{v}}, \dot{\mathrm{w}}) \in \dot{\mathrm{E}}\}$ and analogously for $i \in \{-2, -3, \ldots\}$.

Definition 3.2 *Let* $\dot{\mathrm{G}} = (\dot{\mathrm{V}}, \dot{\mathrm{E}})$ *be a (possibly infinite) graph (tree).*
A subgraph (subtree) $\dot{\mathrm{S}}(\dot{\mathrm{v}})$ *of a node* $\dot{\mathrm{v}} \in \dot{\mathrm{V}}$ *is a graph (tree)* $\dot{\mathrm{S}}(\dot{\mathrm{v}}) := (\dot{\mathrm{V}}', \dot{\mathrm{E}}')$, *where* $\dot{\mathrm{V}}' \subseteq \dot{\mathrm{V}}$, $\dot{\mathrm{E}}' \subseteq \dot{\mathrm{E}}$, *and both subsets are defined by*

$$\dot{\mathrm{V}}' := \bigcup_{i \in \mathrm{N}} \dot{\mathrm{R}}^i(\dot{\mathrm{v}}), \qquad \dot{\mathrm{E}}' := \{(\dot{\mathrm{u}}, \dot{\mathrm{w}}) | \dot{\mathrm{u}} \in \dot{\mathrm{V}}'\}, \qquad \dot{\mathrm{r}}((\dot{\mathrm{V}}', \dot{\mathrm{E}}')) := \dot{\mathrm{v}}$$

Intuitively, this definition for a subgraph encompasses all reachable nodes. Note that in $\dot{\mathrm{S}}(\dot{\mathrm{v}})$, $\dot{\mathrm{v}}$ is the new root of the subgraph. When the subgraph has no cycles, the root has (like a tree) no predecessors. In cases where $\dot{\mathrm{v}}$ is also member of some $\dot{\mathrm{R}}^i(\dot{\mathrm{v}})$, the root also has predecessors.
Now we define the equivalence of arbitrary labeled graphs:

Definition 3.3 *Let* $\dot{\mathrm{G}} = (\dot{\mathrm{V}}, \dot{\mathrm{E}})$ *and* $\dot{\mathrm{G}}' = (\dot{\mathrm{V}}', \dot{\mathrm{E}}')$ *be (possibly infinite) graphs with arbitrary labeling functions* $\mu_1, \ldots, \mu_k$ *resp.* $\mu_1', \ldots, \mu_k'$ *mapping* $\dot{\mathrm{V}}$ *resp.* $\dot{\mathrm{V}}'$ *to an arbitrary free monoid* Γ^*, *and* $\nu_1, \ldots, \nu_l$ *resp.* $\nu_1', \ldots, \nu_l'$ *arbitrary labeling functions from* $\dot{\mathrm{E}}$ *resp.* $\dot{\mathrm{E}}'$ *to* Γ^*.

$$\dot{\mathrm{G}} \equiv \dot{\mathrm{G}}' \quad \textit{if and only if} \quad \forall_{h \in \mathrm{N}} : \dot{\mathrm{G}} \equiv_h \dot{\mathrm{G}}'$$

where $\equiv_h$ *is the finite-case equivalence defined by*

$$\begin{array}{ll}
(1) & \dot{\mathrm{G}} \equiv_0 \dot{\mathrm{G}}' \quad \textit{if and only if} \quad \mu_i(\dot{\mathrm{r}}(\dot{\mathrm{G}})) = \mu_i'(\dot{\mathrm{r}}(\dot{\mathrm{G}}')) \\
(2) & \dot{\mathrm{G}} \equiv_h \dot{\mathrm{G}}' \ (h > 0) \quad \textit{if and only if} \quad \dot{\mathrm{G}} \equiv_0 \dot{\mathrm{G}}', \\
& \forall_{\dot{\mathrm{e}} = (\dot{\mathrm{r}}(\dot{\mathrm{G}}), \dot{\mathrm{v}}) \in \dot{\mathrm{E}}} \exists_{\dot{\mathrm{e}}' = (\dot{\mathrm{r}}(\dot{\mathrm{G}}'), \dot{\mathrm{v}}') \in \dot{\mathrm{E}}'} : \nu_j(\dot{\mathrm{e}}) = \nu_j'(\dot{\mathrm{e}}')\,,\ \dot{\mathrm{S}}(\dot{\mathrm{v}}) \equiv_{h-1} \dot{\mathrm{S}}(\dot{\mathrm{v}}') \\
& \forall_{\dot{\mathrm{e}}' = (\dot{\mathrm{r}}(\dot{\mathrm{G}}'), \dot{\mathrm{v}}') \in \dot{\mathrm{E}}'} \exists_{\dot{\mathrm{e}} = (\dot{\mathrm{r}}(\dot{\mathrm{G}}), \dot{\mathrm{v}}) \in \dot{\mathrm{E}}} : \nu_j'(\dot{\mathrm{e}}') = \nu_j(\dot{\mathrm{e}})\,,\ \dot{\mathrm{S}}(\dot{\mathrm{v}}') \equiv_{h-1} \dot{\mathrm{S}}(\dot{\mathrm{v}})
\end{array}$$

Note that this is no structural equivalence; all we demand is that all edges of $\dot{\mathrm{G}}$ have an equivalent edge in $\dot{\mathrm{G}}'$ leading to a node with the same labels and an equivalent subtree, and vice versa. As we shall see later, there are cases in which the nodes have no 1:1 correspondence but the graphs are equivalent.

Definition 3.4 *Let* $\dot{\mathrm{G}}$ *be a graph labeled as in definition 3.3. We extend the definition of all labeling functions* μ_i *and* ν_j *from nodes resp. edges to paths as follows, yielding sequences of strings: Let* $\hat{a} = (\dot{\mathrm{v}}_1, \ldots, \dot{\mathrm{v}}_k)$ *be an arbitrary path in* $\dot{\mathrm{G}}$. *Then*

$$\mu_i(\hat{a}) := (\mu_i(\dot{\mathrm{v}}_1), \ldots, \mu_i(\dot{\mathrm{v}}_k)), \qquad \nu_j(\hat{a}) := (\nu_j((\dot{\mathrm{v}}_1, \dot{\mathrm{v}}_2)), \ldots, \nu_j((\dot{\mathrm{v}}_{k-1}, \dot{\mathrm{v}}_k)))$$

Now we present a simple lemma which justifies the intuitive notion of the equivalence $\equiv$ by the observation, that every labeled path in $\dot{\mathrm{G}}$ has a corresponding labeled path in $\dot{\mathrm{G}}'$ and vice versa (though the underlying paths may be different):

Lemma 3.5 *Let* $\dot{\mathrm{G}}$ *and* $\dot{\mathrm{G}}'$ *be labeled graphs with* $\dot{\mathrm{G}} \equiv \dot{\mathrm{G}}'$. *For every* $\dot{\mathrm{v}} \in \dot{\mathrm{V}}$ *exists a* $\dot{\mathrm{v}}' \in \dot{\mathrm{V}}'$ *such that*

$$\mu_i((\dot{\mathrm{r}}(\dot{\mathrm{G}}), \ldots, \dot{\mathrm{v}})) = \mu_i'((\dot{\mathrm{r}}(\dot{\mathrm{G}}'), \ldots, \dot{\mathrm{v}}')), \qquad \nu_j((\dot{\mathrm{r}}(\dot{\mathrm{G}}), \ldots, \dot{\mathrm{v}})) = \nu_j'((\dot{\mathrm{r}}(\dot{\mathrm{G}}'), \ldots, \dot{\mathrm{v}}'))$$

Now we get the main lemma of this paper, showing equivalences on the CTPL:

Lemma 3.6 *Let G be a grammar and $\dot{\mathrm{G}} = (\dot{\mathrm{V}}, \dot{\mathrm{E}})$ be the corresponding CTPL.*

$$\forall_{\dot{\mathrm{v}},\dot{\mathrm{w}}\in\dot{\mathrm{V}}\,,\,\iota(\dot{\mathrm{v}})=\iota(\dot{\mathrm{w}})\,,\,\sigma(\dot{\mathrm{v}})=\sigma(\dot{\mathrm{w}})}\ :\ \dot{\mathrm{S}}(\dot{\mathrm{v}}) \equiv \dot{\mathrm{S}}(\dot{\mathrm{w}})$$

Now we have the tools for compressing the CTPL to a graph. However, when applying this lemma to a CTPL, we could compress it to a finite graph reflecting all possible left-derivations as a static structure, which is not what we want. When concerned with parsing, we prune all branches of the CTPL not matching the input, so we cannot apply this lemma to parts of the pruned CTPL, because different subtrees beginning with the same labeling may derive to different parts of the input and therefore may be pruned in different ways.

First, we have to state a model how pruning of the CTPL is done. We will only consider the case of online parsing from left to right in this paper. In this model, we have sequential access to the input symbols a_i for i from 1 to n. After accessing each symbol, we update our CGPL in such a way that it reflects the known parts of the CTPL.

The process of updating the CGPL can be described as a sequence of graphs $\dot{\mathrm{G}}_i = (\dot{\mathrm{V}}_i, \dot{\mathrm{E}}_i)$ where $\forall_{\dot{\mathrm{v}}\in\dot{\mathrm{V}}_i}\ :\ \hat{\pi}(\dot{\mathrm{v}}) = a_0 \dots a_j$ where $j \leq i$. Those nodes where $j = i$ are called cursor nodes, denoted by $\dot{\mathrm{C}}_i$.

This means to keep only those parts of the CTPL in the graph where each node has a total prefix matching a substring $a_0 \dots a_i$ of the current input. In other words, the cursors at step i are all the nodes having exactly total prefix $a_0 \dots a_i$. What could be currently derived from them are unpruned subtrees of the CTPL, because we needed additional input symbols to prune these subtrees also. So it is sufficient not to store these subtrees as a graph by leaving them simply out, for they can be completed when reading the next input symbol, so only the matching parts of these subtrees will be added.

At first glance, this situation seems strange: For the known parts of our graph, lemma 3.6 cannot be applied in general, in contrast to the unknown parts. However, just this is the key for compressing the pruned CTPL to a graph with polynomial space complexity: All cursors having the same ι and σ label are equivalent by lemma 3.6, so they can be melted into one single node. In subsequent input steps, the pruning of the different original subtrees would be done in parallel; this means that after pruning has been consistently done, the different subtrees remain equivalent. So we are entitled to melt them.

Now we are ready to state the main theorem of this paper, showing that all identically labeled cursor nodes produced in one input step can be melted.

Theorem 3.7 *Let G be a grammar, $\dot{\mathrm{G}}^T$ the corresponding CTPL, and let $x = a_1 \dots a_n$ be an input word. Let $\{\dot{\mathrm{G}}_i\}$ be an arbitrary sequence of graphs for $i = 0, \dots, n$ with the property $\dot{\mathrm{G}}_i \equiv (\dot{\mathrm{V}}_i^T, \dot{\mathrm{E}}_i^T)$, where $(\dot{\mathrm{V}}_i^T, \dot{\mathrm{E}}_i^T)$ is the subtree of $\dot{\mathrm{G}}^T$ containing exactly all nodes with $\forall_{\dot{\mathrm{v}}\in\dot{\mathrm{V}}_i^T}\ :\ \hat{\pi}(\dot{\mathrm{v}}) = a_1 \dots a_j$, $0 \leq j \leq i$ [2]. Then for every $\dot{\mathrm{G}}_i$ exists an equivalent graph $\dot{\mathrm{G}}'_i \equiv \dot{\mathrm{G}}_i$ such that*

$$\forall_{\dot{\mathrm{v}},\dot{\mathrm{w}}\in\dot{\mathrm{C}}_i(\dot{\mathrm{G}}'_i),\,\dot{\mathrm{v}}\neq\dot{\mathrm{w}}}\ :\ \iota(\dot{\mathrm{v}}) \neq \iota(\dot{\mathrm{w}}) \vee \sigma(\dot{\mathrm{v}}) \neq \sigma(\dot{\mathrm{w}})$$

In other words, the graph $\dot{\mathrm{G}}'_i$ is minimal with respect to the number of cursor nodes. From this theorem, we can directly construct a parser for $L(G)$. In step 0, we start with $\dot{\mathrm{G}}'_0$ containing only the root with infix label S. In step i we have a graph $\dot{\mathrm{G}}'_i$, and now step $i+1$ begins with reading input symbol a_{i+1}. We produce $\dot{\mathrm{G}}_{i+1}$ from $\dot{\mathrm{G}}'_i$ by appending the additional cursor nodes $\dot{\mathrm{C}}_{i+1}$ using definiton 2.2 without rule (3), and lastly produce $\dot{\mathrm{G}}'_{i+1}$ by melting together all cursor nodes having the same labels using lemma 3.6. We can see at once, that in every input step i the graph $\dot{\mathrm{G}}'_i$ has at most k cursor nodes, where k is a constant depending on the grammar G. The difference between $\dot{\mathrm{G}}'_i$ and $\dot{\mathrm{G}}'_{i+1}$ lies only in the added cursor nodes, so we can use a single growing graph for our algorithm, and the total number of nodes can be bounded by $O(n)$. Since $\dot{\mathrm{E}} \subseteq \dot{\mathrm{V}} \times \dot{\mathrm{V}}$, each node is connected with at most $O(n)$ other nodes, and therefore the total space complexity is only $O(n^2)$. By construction, this growing graph is equivalent to (the matching parts of) the CTPL and therefore contains the same information; at least it contains the information of all successful parses when the algorithm terminates after reading the last input symbol.

4 A Recognizer Algorithm

We present a recognizer based on theorem 3.7 which is as simple as possible. The simplifications are:

- The prefix labels contain redundant information, for the prefix always matches the input symbols already processed. So we leave the prefix labels out.

[2] For $j = 0$, we define $a_1 \dots a_j := \epsilon$ as straightforward extension

- The labeling functions ι and σ are concatenated into one labeling function ς with $\varsigma(\dot{v}) := \iota(\dot{v})\sigma(\dot{v})$. We relax rule (3) of definition 2.2. When $\varsigma(\dot{v})$ begins with a nonterminal, then this denotes the infix; in all other cases the infix is empty.
- We introduce a new terminal symbol $\dashv$ as end-of-file marker and therefore augment the grammar by a new production $S' \to S \dashv$ without changing the generated language, using S' as new start symbol. This is to prevent the root from appearing in a cycle.

The set of all cursor nodes at step i is denoted by $\dot{C}_i$. We indicate the structure of the algorithm by indentation.

Algorithm 4.1 Let $G = (\Sigma, N, P, S)$ be a grammar, $x = a_1 ... a_n \dashv$ an input word.
Let $G' := (\Sigma \cup \{\dashv\}, N \cup \{S'\}, P', S')$ a modified grammar with new start symbol S' and $P' = P \cup \{S' \to S \dashv\}$.
The CGPL $\dot{G}$ is initialized with root $\dot{s}$ labeled $\varsigma(\dot{s}) := S'$ and an edge $(\dot{s}, \dot{t})$ with $\varsigma(\dot{t}) := S \dashv$. Initialize $\dot{C}_0 := \{\dot{t}\}$.
Program MAIN

FOR $i := 1, \ldots, n+1$ DO:

1) $\forall$ cursors $\dot{w} \in \dot{C}_{i-1}$ which have not yet been processed:
   a) IF $\varsigma(\dot{w}) = X\delta$, where $X \in N$:
      remove $\dot{w}$ from $\dot{C}_{i-1}$
      $\forall$ productions $X \to \gamma \in P$:
         IF $\gamma = \epsilon$:
            MAKEBROTHER ($\dot{w}$)
         ELSE
            GENERATE ($\dot{w}$, γ)
   b) IF $\varsigma(\dot{w}) = \epsilon$:
      $\forall$ fathers $\dot{v}$ of $\dot{w}$:
         MAKEBROTHER ($\dot{v}$)
2) $\forall$ cursors $\dot{w} \in \dot{C}_{i-1}$, $\varsigma(\dot{w}) = a\delta$:
   IF $a = a_i$: /* input matches */
      replace $\varsigma(\dot{w}) := \delta$
      add $\dot{w}$ to $\dot{C}_i$ /* Let only matching nodes survive */
3) IF $\dot{C}_i = \emptyset$:
   reject the input word and stop.
4) IF $a_i = \dashv$:
   accept the input word and stop.

Procedure MAKEBROTHER ($\dot{w}$):
/* Apply rule (5) of definiton 2.2 for those nodes where this has not been done yet. */
IF $\dot{w}$ has not been considered in step i for this test:
   $\forall$ fathers $\dot{v}$ of $\dot{w}$:
      IF $\sigma(\dot{w}) = \epsilon$:
         MAKEBROTHER ($\dot{v}$) /* Don't generate ϵ-brothers */
      ELSE
         GENERATE ($\dot{v}$, $\sigma(\dot{w})$) /* Generate brother of $\dot{w}$ */

Procedure GENERATE ($\dot{v}$, α):
/* Generate a new node with label α and connect it to father $\dot{v}$ */
/* If such a node was already generated in step i, use it instead. */
IF $\nexists$ $\dot{w}$ generated in step i with $\varsigma(\dot{w}) = \alpha$:
   generate a new node $\dot{w}$.
   $\varsigma(\dot{w}) := \alpha$
   Add $\dot{w}$ to $\dot{C}_{i-1}$
ELSIF MAKEBROTHER ($\dot{w}$) had been completed previously in step i:
   /* Make up leeway */ let $\alpha = X\delta$ for the following section:
   IF $\delta = \epsilon$:
      MAKEBROTHER ($\dot{v}$) /* Don't generate ϵ-brothers */
   ELSE
      GENERATE ($\dot{v}$, δ) /* Generate/Connect also to brother of $\dot{w}$ */
add a new edge $(\dot{v}, \dot{w})$ to $\dot{E}$

Explanations:

- The two steps of theorem 3.7, generating $\dot{G}_{i+1}$ and therefrom $\dot{G}'_{i+1}$, are combined. Procedure GENERATE automatically unifies all nodes which would be generated in $\dot{G}_{i+1}$ in one input step, so we directly produce $\dot{G}'_{i+1}$.

- GENERATE will never produce an ϵ-node, because it is always called with a non-empty argument for α. Instead, rule (5) of definition 2.2 is applied at once whenever a call with ϵ would be possible.

- The quantifier $\forall$ denotes a loop with nondeterministic choice of the quantified elements. In case 1) of MAIN, we can select any element from the cursor set until all elements have been processed. Note that while processing an element, new members may be added to the cursor set by GENERATE.

- MAKEBROTHER will "generate" a brother of a node or call itself recursively in case of ϵ-brothers only if this has not been done in the same input step before. In this way, cycles in the graph and/or multiple copies caused by different paths to the same node will not matter[3]. Whenever a brother of some $\dot{w}$ has to be "generated", it has to be connected to all of $\dot{w}$'s fathers, which is done automatically by the unification of GENERATE. This results in (at most) one brother connected to all fathers. However, in some cases not all fathers are known to MAKEBROTHER, because new fathers may be added later. If new fathers are added later to such a node $\dot{w}$ via GENERATE, we have to make up this leeway by doing the same things as had been done previously in MAKEBROTHER.

- All really new generated nodes are added to the set of cursor nodes $\dot{C}_{i-1}$ without regard, whether their total prefix really matches all read input symbols. When the generated γ's begin with another nonterminal, they are treated again and will therefore be removed from $\dot{C}_{i-1}$. At the end of step 1) of MAIN, all nodes in $\dot{C}_{i-1}$ begin with a terminal. The subsequent step 2) will take over to $\dot{C}_i$ only the matching nodes.

5 Complexity Considerations

By construction, algorithm 4.1 accepts any context-free grammar[4] G. First we estimate the space and time complexities in the worst case, when the grammar is unrestricted:

In worst case, procedure GENERATE can produce at most c different nodes in any input step, where c is a constant bounded by the grammar size. So the total number of nodes is $O(n)$, and the total space complexity is $O(n^2)$ for the edges, assuming that nodes may have $O(n)$ predecessors.

(a) To see the general case time complexity at once is a little more difficult. First, we leave out the ELSIF part of GENERATE which complicates things. When we have shown the complexity without it, we will find that its presence does not change anything. We can state the following:

- The inner complexity of GENERATE (assuming to leave out the ELSIF part) is bounded by a constant, for it has to compare α with at most c nodes generated in the same input step. In a smart implementation, this comparison may be achieved by a simple table lookup.

- The sets $\dot{C}_{i-1}$ never contain more than $2c$ elements, because c new elements may be added by GENERATE to at most c old members.

- Step 1) of MAIN may execute the subcases a) and b) at most $2c$ times in every pass of the i-loop. The inner complexity of case b) is $O(n)$, yielding an overall time complexity of $O(n^2)$ caused by step 1).

- Regardless of the number of times MAKEBROTHER will be called, it performs its inner body only for at most $O(n)$ different nodes in one input step. When the node has been tested before, the inner complexity is bounded by a constant. Otherwise, when the inner body is executed, its inner complexity is $O(n)$ because $\dot{w}$ has at most $O(n)$ fathers. So the complexity caused by MAKEBROTHER in one input step is $O(n^2)$, in spite of the recursivity. The trick is to prevent cycles and multiple visits by the test.

- Thus follows the overall time complexity of $O(n^3)$.

[3]In fact, this is basically a transitive closure algorithm.

[4]There are no restrictions on the grammar; it need not be reduced, may have ϵ-productions and may contain useless and unit productions, or cycles $X \stackrel{+}{\Rightarrow} X$

Now we have to release our assumption that the ELSIF part of GENERATE is left out. The only purpose of the ELSIF part is to do the same things which would have been done by MAKEBROTHER if all fathers had been present from the beginning. So our argumentation will hold also for taking the ELSIF part into account.

(b) Now we are interested in the time complexities when the grammar is restricted in some way. First we investigate the case of bounded ambiguous grammars. A grammar is called to be of bounded ambiguity k when all possible derivations lead to at most k different derivation trees.

The following arguments show that the overall complexity is $O(n^2)$ in this case: By definition of the bounded ambiguity, all nodes $\dot{\mathrm{v}}$ in the CTPL have at most k successors which will "survive" until rule (5) of definiton 2.2 is applied the last time to $\dot{\mathrm{v}}$. Note that as long as the subgraph $\dot{\mathrm{S}}(\dot{\mathrm{v}})$ is not completed, there may be more successors, but they are guaranteed to produce not more than k brothers of $\dot{\mathrm{v}}$ in at most k different input steps, because the sublanguage $L(\iota(\dot{\mathrm{v}}))$ is also of bounded ambiguity k. Therefore the maximum number of brothers $\dot{\mathrm{v}}$ can get by rule (5) is k.

As a consequence, the total number of successors of each node is bounded by some constant k'. This is because successors are generated either in step 1)a) of MAIN (at most $|P|$) or by MAKEBROTHER, which will be called at most k times for each differently labeled brother[5], so the maximum number of successors depends only on the grammar.

Thus follows that the total number of edges in our graph is now $O(n)$. Each time MAKEBROTHER examines all fathers of $\dot{\mathrm{w}}$, it goes through the incoming edges of $\dot{\mathrm{w}}$. In one input step any edge can be touched only once by MAKEBROTHER (or substitutively by GENERATE), so it has complexity $O(n)$ in this input step, resulting in an overall complexity of the algorithm of $O(n^2)$.

(c) Finally we investigate the case where the grammar is $LR(k)$. By definiton of the $LR(k)$ property, multiple branches occuring in the CTPL will lead to a dead end after at most k input steps. Now we argue in a simple way: Let $\dot{\mathrm{v}}$ be in $\dot{\mathrm{C}}_{i-1}$, then for $i < j < i+k$ there may occur multiple descendants of $\dot{\mathrm{v}}$ in $\dot{\mathrm{C}}_{j-1}$, but in $\dot{\mathrm{C}}_{i+k-1}$ there will remain at most one descendant of $\dot{\mathrm{v}}$. So the number of brothers produced by MAKEBROTHER in each input step is bounded by some constant k', for each brother generated at a certain input step j must be a descendant of a bounded number of cursors of previous cursor sets.

As can be seen easily, MAKEBROTHER now has a constant bounded complexity in each input step; the same is true for step 1) of MAIN and for all other parts of the algorithm. So we have an overall time complexity of $O(n)$.

As a result of this section, we find that the worst case asymptotic time complexity of our algorithm is as good as Earley's. We are currently investigating the practical aspects of our algorithm, to see the behaviour of constant factors. Also, the nondeterministic choices in step 1) of MAIN give the opportunity to use several heuristic selection strategies. We recommend a queue structure for the $\dot{\mathrm{C}}_{i-1}$, where elements are removed at the front and added at the end. However, we have currently no idea whether other strategies (like sorting by number of successors etc.) will give some gains at least with certain grammars. This may be a wide research area.

6 Possible Modifications, Conclusion

Our Algorithm could be classified as an efficient general directional top-down method with maximally restricted breadth-first search. There are several ways to implement our it; one has been carried out by the author. However, there are more interesting points regarding possible modifications of the algorithm itself.

Several techniques known from improvements of Earley's algorithm (see, e. g., [GHR80], [GJ90], [Lei90]) could be applied to our algorithm in a modified form. For example, we could introduce lookahead in several ways. A simple way would be for example to compute FIRST and FOLLOW sets to restrict case 1)a) of MAIN to those nodes matching the lookahead. Similarly, the reduction in case 2) could be restricted by lookahead techniques. Other types of precomputing could be useful, too.

On the lowest level, it would be interesting to compare the running time of our algorithm to the other known general context-free parsers. This seems to be very difficult, for there are no commonly defined micro operations allowing a hardware independent comparison.

Earley's algorithm has been adapted for many grammar classes used in the field of natural language processing. So it is interesting whether adapting our algorithm to, say, ID/LP grammars or various

[5] However, each generated brother may produce new brothers in turn, but they have shorter labels. This process will terminate in at most c' generations of brothers, where c' is the maximum number of nonterminals at the righthand side of the productions

unification methods would give any gains.

Also, it can be seen easily that there are efficient bit-vector implementations: Just view the graph as adjacency matrix, then operate on sets of edges by representing them as bit vectors. We expect this will increase speed on real machines and on real problems considerably.

The interrelations to Earley's algorithm and varieties will be explained in a separate paper. To our current knowledge, there is an argument that gives evidence our algorithm is at least as good as Earley's: Just partition Earley's state sets S_i into groups where all group members have the same dotted rule $X \to \alpha.\beta$ (but different backward pointers). Then each group roughly corresponds to a node in our algorithm, labeled with β. The main difference is, that Earley's pointers point to state sets, wheras our edges point to "states" respective such "groups". At first glance, Earley's structure seems to be less expensive than our's, because one particular Earley pointer pointing to a state set may address multiple states, while we need an extra edge for each of them. However, Earley's completer has to do exhaustive search to find all of these nodes within the addressed state set, while our algorithm simply follows an edge backward. Another difference to Earley's algorithm is that it only can unify dotted rules resulting from the same production, while our algorithm takes into account only the remainder behind the dot for melting nodes. Conversely, we have therefore discovered an improvement of Earley's algorithm: Just view states $< X \to \alpha.\beta, j >$ and $< X \to \alpha'.\beta, j >$ as equal even if $\alpha \neq \alpha'$. However, our algorithm would allow to unify $< X \to \alpha.\beta, j >$ and $< Y \to \alpha'.\beta, j >$ where $X \neq Y$, speaking analogously in terms of Earley.

The interrelations to Tomita's algorithms are not clear at present. When applying graph structured stacks [To88] to a top-down algorithm (e. g. apply graph structuring to the $LL(1)$ method), we will get an algorithm which seems similar to ours. The main difference will be that labels are symbols rather than strings. A similar behaviour could be seen in our algorithm when the grammar is transformed to a special left-recursion free chomsky normal form where some redundant information is thrown out by sharing common suffixes of the original grammar by the same chains of rules. But we feel our algorithm will be more practical than this.

References

[BPS75] Bouckaert, M., Pirotte, A., Snelling, M.: *Efficient Parsing Algorithms for General Context-free Parsers.* Inf. Sciences, vol. 8, no. 1, p. 1-26 (Jan 1975)

[Ea70] Earley, J.: *An Efficient Context-Free Parsing Algorithm.* Comm. ACM 13, 2 (Feb. 1970), 94-102

[GJ90] Grune, D., Jacobs, C. J. H.: *Parsing Techniques. a practical guide.* Ellis Horwood 1990, ISBN 0-13-651431-6

[GHR80] Graham, S. L., Harrison, M. L., Ruzzo, W. L.: *An Improved Context-Free Recognizer.* ACM Trans. Prog. Lang. Syst. 2, 3 (1980), 415-462

[HU79] Hopcroft, J. E., Ullman, J. D.: *Introduction to Automata Theory, Languages, and Computation.* Addison Wesley 1979, ISBN 0-201-02988-X

[KA89] Kruseman Aretz, F. E. J.: *A new approach to Earley's parsing algorithm.* Sci. Comput. Program (Netherlands), vol. 12, no. 2, p. 105-121 (July 1989)

[Lei90] Leiss, H.: *On Kilbury's Modification of Earley's Algorithm.* ACM Trans. Programming Languages and Systems, Vol. 12, No. 4, p. 610-640 (Oct. 1990)

[To86] Tomita, M.: *Efficient parsing for natural languages.* Kluwer Academic Publishers, Boston, 1986, p. 201

[To88] Tomita, M.: *Graph-structured stack and natural language parsing.* 26th Annual Meeting of the Association for Computational Linguistics. Proceedings of the Conference, Buffalo, NY, 7-10 June 1988 (Morristown, NJ: Assoc. Comput. Linguistics 1988) p. 249-257