Incremental Parsing with Parallel Multiple Context-Free Grammars

Krasimir Angelov
Chalmers University of Technology
Göteborg, Sweden
krasimir@chalmers.se

Abstract

Parallel Multiple Context-Free Grammar (PMCFG) is an extension of context-free grammar for which the recognition problem is still solvable in polynomial time. We describe a new parsing algorithm that has the advantage to be incremental and to support PMCFG directly rather than the weaker MCFG formalism. The algorithm is also top-down which allows it to be used for grammar based word prediction.

1 Introduction

Parallel Multiple Context-Free Grammar (PMCFG) (Seki et al., 1991) is one of the grammar formalisms that have been proposed for the syntax of natural languages. It is an extension of context-free grammar (CFG) where the right hand side of the production rule is a tuple of strings instead of only one string. Using tuples the grammar can model discontinuous constituents which makes it more powerful than context-free grammar. In the same time PMCFG has the advantage to be parseable in polynomial time which makes it attractive from computational point of view.

A parsing algorithm is incremental if it reads the input one token at the time and calculates all possible consequences of the token, before the next token is read. There is substantial evidence showing that humans process language in an incremental fashion which makes the incremental algorithms attractive from cognitive point of view.

If the algorithm is also top-down then it is possible to predict the next word from the sequence of preceding words using the grammar. This can be used for example in text based dialog systems or text editors for controlled language where the user might not be aware of the grammar coverage. In this case the system can suggest the possible continuations.

A restricted form of PMCFG that is still stronger than CFG is Multiple Context-Free Grammar (MCFG). In Seki and Kato (2008) it has been shown that MCFG is equivalent to string-based Linear Context-Free Rewriting Systems and Finite-Copying Tree Transducers and it is stronger than Tree Adjoining Grammars (Joshi and Schabes, 1997). Efficient recognition and parsing algorithms for MCFG have been described in Nakanishi et al. (1997), Ljunglöf (2004) and Burden and Ljunglöf (2005). They can be used with PMCFG also but it has to be approximated with overgenerating MCFG and post processing is needed to filter out the spurious parsing trees.

We present a parsing algorithm that is incremental, top-down and supports PMCFG directly. The algorithm exploits a view of PMCFG as an infinite context-free grammar where new context-free categories and productions are generated during parsing. It is trivial to turn the algorithm into statistical by attaching probabilities to each rule.

In Ljunglöf (2004) it has been shown that the **Grammatical Framework** (GF) formalism (Ranta, 2004) is equivalent to PMCFG. The algorithm was implemented as part of the GF interpreter and was evaluated with the **resource grammar library** (Ranta, 2008) which is the largest collection of grammars written in this formalism. The incrementality was used to build a help system which suggests the next possible words to the user.

Section 2 gives a formal definition of PMCFG. In section 3 the procedure for "linearization" i.e. the derivation of string from syntax tree is defined. The definition is needed for better understanding of the formal proofs in the paper. The algorithm introduction starts with informal description of the idea in section 4 and after that the formal rules are given in section 5. The implementation details are outlined in section 6 and after that there are some comments on the evaluation in section 7. Section 8 gives a conclusion.

2 PMCFG definition

Definition 1 *A parallel multiple context-free grammar is an 8-tuple* $G = (N, T, F, P, S, d, r, a)$ *where:*

- N *is a finite set of categories and a positive integer* $d(A)$ *called dimension is given for each* $A \in N$.
- T *is a finite set of terminal symbols which is disjoint with* N.
- F *is a finite set of functions where the arity* $a(f)$ *and the dimensions* $r(f)$ *and* $d_i(f)$ $(1 \leq i \leq a(f))$ *are given for every* $f \in F$. *For every positive integer* d, $(T^*)^d$ *denote the set of all* d*-tuples*

Proceedings of the 12th Conference of the European Chapter of the ACL, pages 69–76,
Athens, Greece, 30 March – 3 April 2009. ©2009 Association for Computational Linguistics

of strings over T. Each function $f \in F$ is a total mapping from $(T^*)^{d_1(f)} \times (T^*)^{d_2(f)} \times \cdots \times (T^*)^{d_{a(f)}(f)}$ to $(T^*)^{r(f)}$, defined as:

$f := (\alpha_1, \alpha_2, \ldots, \alpha_{r(f)})$

Here α_i is a sequence of terminals and $\langle k; l\rangle$ pairs, where $1 \leq k \leq a(f)$ is called argument index and $1 \leq l \leq d_k(f)$ is called constituent index.

- *P is a finite set of productions of the form:*

  $A \to f[A_1, A_2, \ldots, A_{a(f)}]$

  where $A \in N$ is called result category, $A_1, A_2, \ldots, A_{a(f)} \in N$ are called argument categories and $f \in F$ is the function symbol. For the production to be well formed the conditions $d_i(f) = d(A_i)$ $(1 \leq i \leq a(f))$ and $r(f) = d(A)$ must hold.

- *S is the start category and $d(S) = 1$.*

We use the same definition of PMCFG as is used by Seki and Kato (2008) and Seki et al. (1993) with the minor difference that they use variable names like x_{kl} while we use $\langle k; l\rangle$ to refer to the function arguments. As an example we will use the $a^n b^n c^n$ language:

$$
\begin{aligned}
S &\to c[N] \\
N &\to s[N] \\
N &\to z[]
\end{aligned}
$$

$$
\begin{aligned}
c &:= (\langle 1;1\rangle\ \langle 1;2\rangle\ \langle 1;3\rangle) \\
s &:= (a\ \langle 1;1\rangle, b\ \langle 1;2\rangle, c\ \langle 1;3\rangle) \\
z &:= (\epsilon, \epsilon, \epsilon)
\end{aligned}
$$

Here the dimensions are $d(S) = 1$ and $d(N) = 3$ and the arities are $a(c) = a(s) = 1$ and $a(z) = 0$. ϵ is the empty string.

3 Derivation

The derivation of a string in PMCFG is a two-step process. First we have to build a syntax tree of a category S and after that to linearize this tree to string. The definition of a syntax tree is recursive:

Definition 2 *$(f\ t_1 \ldots t_{a(f)})$ is a tree of category A if t_i is a tree of category B_i and there is a production:*
$A \to f[B_1 \ldots B_{a(f)}]$
The abstract notation for "t is a tree of category A" is $t : A$. When $a(f) = 0$ then the tree does not have children and the node is called leaf.

The linearization is bottom-up. The functions in the leaves do not have arguments so the tuples in their definitions already contain constant strings. If the function has arguments then they have to be linearized and the results combined. Formally this can be defined as a function $\mathcal{L}$ applied to the syntax tree:

$$
\begin{aligned}
&\mathcal{L}(f\ t_1\ t_2 \ldots t_{a(f)}) = (x_1, x_2 \ldots x_{r(f)}) \\
&\quad \text{where } x_i = \mathcal{K}(\mathcal{L}(t_1), \mathcal{L}(t_2) \ldots \mathcal{L}(t_{a(f)}))\ \alpha_i \\
&\quad \text{and } f := (\alpha_1, \alpha_2 \ldots \alpha_{r(f)}) \in F
\end{aligned}
$$

The function uses a helper function $\mathcal{K}$ which takes the already linearized arguments and a sequence α_i of terminals and $\langle k; l\rangle$ pairs and returns a string. The string is produced by simple substitution of each $\langle k; l\rangle$ with the string for constituent l from argument k:

$$\mathcal{K}\ \sigma\ (\beta_1\langle k_1; l_1\rangle\beta_2\langle k_2; l_2\rangle \ldots \beta_n) = \beta_1\sigma_{k_1 l_1}\beta_2\sigma_{k_2 l_2} \ldots \beta_n$$

where $\beta_i \in T^*$. The recursion in $\mathcal{L}$ terminates when a leaf is reached.

In the example $a^n b^n c^n$ language the function z does not have arguments and it corresponds to the base case when $n = 0$. Every application of s over another tree $t : N$ increases n by one. For example the syntax tree $(s\ (s\ z))$ will produce the tuple (aa, bb, cc). Finally the application of c combines all elements in the tuple in a single string i.e. $c\ (s\ (s\ z))$ will produce the string $aabbcc$.

4 The Idea

Although PMCFG is not context-free it can be approximated with an overgenerating context-free grammar. The problem with this approach is that the parser produces many spurious parse trees that have to be filtered out. A direct parsing algorithm for PMCFG should avoid this and a careful look at the difference between PMCFG and CFG gives an idea. The context-free approximation of $a^n b^n c^n$ is the language $a^* b^* c^*$ with grammar:

$$
\begin{aligned}
S &\to ABC \\
A &\to \epsilon \mid aA \\
B &\to \epsilon \mid bB \\
C &\to \epsilon \mid cC
\end{aligned}
$$

The string "$aabbcc$" is in the language and it can be derived with the following steps:

$$
\begin{aligned}
& S \\
\Rightarrow\ & ABC \\
\Rightarrow\ & aABC \\
\Rightarrow\ & aaABC \\
\Rightarrow\ & aaBC \\
\Rightarrow\ & aabBC \\
\Rightarrow\ & aabbBC \\
\Rightarrow\ & aabbC \\
\Rightarrow\ & aabbcC \\
\Rightarrow\ & aabbccC \\
\Rightarrow\ & aabbcc
\end{aligned}
$$

The grammar is only an approximation because there is no enforcement that we will use only equal number of reductions for A, B and C. This can be guaranteed if we replace B and C with new categories B' and C' after the derivation of A:

$$B' \to bB'' \qquad C' \to cC''$$
$$B'' \to bB''' \qquad C'' \to cC'''$$
$$B''' \to \epsilon \qquad C''' \to \epsilon$$

In this case the only possible derivation from $aaB'C'$ is $aabbcc$.

The PMCFG parser presented in this paper works like context-free parser, except that during the parsing it generates fresh categories and rules which are specializations of the originals. The newly generated rules are always versions of already existing rules where some category is replaced with new more specialized category. The generation of specialized categories prevents the parser from recognizing phrases that are otherwise withing the scope of the context-free approximation of the original grammar.

5 Parsing

The algorithm is described as a deductive process in the style of (Shieber et al., 1995). The process derives a set of items where each item is a statement about the grammatical status of some substring in the input.

The inference rules are in natural deduction style:

$$\frac{X_1 \ldots X_n}{Y} < \text{side conditions on } X_1, \ldots, X_n >$$

where the premises X_i are some items and Y is the derived item. We assume that $w_1 \ldots w_n$ is the input string.

5.1 Deduction Rules

The deduction system deals with three types of items: active, passive and production items.

Productions In Shieber's deduction systems the grammar is a constant and the existence of a given production is specified as a side condition. In our case the grammar is incrementally extended at runtime, so the set of productions is part of the deduction set. The productions from the original grammar are axioms and are included in the initial deduction set.

Active Items The active items represent the partial parsing result:
$[^k_j A \to f[\vec{B}]; l : \alpha \bullet \beta]\,, \quad j \le k$

The interpretation is that there is a function f with a corresponding production:

$$A \to f[\vec{B}]$$
$$f := (\gamma_1, \ldots \gamma_{l-1}, \alpha\beta, \ldots \gamma_{r(f)})$$

such that the tree $(f\ t_1 \ldots t_{a(f)})$ will produce the substring $w_{j+1} \ldots w_k$ as a prefix in constituent l for any sequence of arguments $t_i : B_i$. The sequence α is the part that produced the substring:

$$\mathcal{K}(\mathcal{L}(t_1), \mathcal{L}(t_2) \ldots \mathcal{L}(t_{a(f)}))\ \alpha = w_{j+1} \ldots w_k$$

and β is the part that is not processed yet.

Passive Items The passive items are of the form:
$[^k_j A; l; N]\,, \quad j \le k$
and state that there exists at least one production:

$$A \to f[\vec{B}]$$
$$f := (\gamma_1, \gamma_2, \ldots \gamma_{r(f)})$$

and a tree $(f\ t_1 \ldots t_{a(f)}) : A$ such that the constituent with index l in the linearization of the tree is equal to $w_{j+1} \ldots w_k$. Contrary to the active items in the passive the whole constituent is matched:

$$\mathcal{K}(\mathcal{L}(t_1), \mathcal{L}(t_2) \ldots \mathcal{L}(t_{a(f)}))\ \gamma_l = w_{j+1} \ldots w_k$$

Each time when we complete an active item, a passive item is created and at the same time we create a new category N which accumulates all productions for A that produce the $w_{j+1} \ldots w_k$ substring from constituent l. All trees of category N must produce $w_{j+1} \ldots w_k$ in the constituent l.

There are six inference rules (see figure 1).

The INITIAL PREDICT rule derives one item spanning the $0 - 0$ range for each production with the start category S on the left hand side. The $\text{rhs}(f, l)$ function returns the constituent with index l of function f.

In the PREDICT rule, for each active item with dot before a $\langle d; r \rangle$ pair and for each production for B_d, a new active item is derived where the dot is in the beginning of constituent r in g.

When the dot is before some terminal s and s is equal to the current terminal w_k then the SCAN rule derives a new item where the dot is moved to the next position.

INITIAL PREDICT

$$\frac{S \to f[\vec{B}]}{[^0_0 S \to f[\vec{B}]; 1 : \bullet\alpha]} \quad S \text{ - start category}, \alpha = \text{rhs}(f, 1)$$

PREDICT

$$\frac{B_d \to g[\vec{C}] \qquad [^k_j A \to f[\vec{B}]; l : \alpha \bullet \langle d; r \rangle\ \beta]}{[^k_k B_d \to g[\vec{C}]; r : \bullet\gamma]} \quad \gamma = \text{rhs}(g, r)$$

SCAN

$$\frac{[^k_j A \to f[\vec{B}]; l : \alpha \bullet s\ \beta]}{[^{k+1}_j A \to f[\vec{B}]; l : \alpha\ s \bullet \beta]} \quad s = w_{k+1}$$

COMPLETE

$$\frac{[^k_j A \to f[\vec{B}]; l : \alpha\bullet]}{N \to f[\vec{B}] \qquad [^k_j A; l; N]} \quad N = (A, l, j, k)$$

COMBINE

$$\frac{[^u_j A \to f[\vec{B}]; l : \alpha \bullet \langle d; r \rangle\ \beta] \qquad [^k_u B_d; r; N]}{[^k_j A \to f[\vec{B}\{d := N\}]; l : \alpha\ \langle d; r \rangle \bullet \beta]}$$

Figure 1: Deduction Rules

When the dot is at the end of an active item then it is converted to passive item in the COMPLETE rule. The category N in the passive item is a fresh category created for each unique (A, l, j, k) quadruple. A new production is derived for N which has the same function and arguments as in the active item.

The item in the premise of COMPLETE was at some point predicted in PREDICT from some other item. The COMBINE rule will later replace the occurence A in the original item (the premise of PREDICT) with the specialization N.

The COMBINE rule has two premises: one active item and one passive. The passive item starts from position u and the only inference rule that can derive items with different start positions is PREDICT. Also the passive item must have been predicted from active item where the dot is before $\langle d; r\rangle$, the category for argument number d must have been B_d and the item ends at u. The active item in the premise of COMBINE is such an item so it was one of the items used to predict the passive one. This means that we can move the dot after $\langle d; r\rangle$ and the d-th argument is replaced with its specialization N.

If the string β contains another reference to the d-th argument then the next time when it has to be predicted the rule PREDICT will generate active items, only for those productions that were successfully used to parse the previous constituents. If a context-free approximation was used this would have been equivalent to unification of the redundant subtrees. Instead this is done at runtime which also reduces the search space.

The parsing is successful if we had derived the $[^n_0 S; 1; S']$ item, where n is the length of the text, S is the start category and S' is the newly created category.

The parser is incremental because all active items span up to position k and the only way to move to the next position is the SCAN rule where a new symbol from the input is consumed.

5.2 Soundness

The parsing system is sound if every derivable item represents a valid grammatical statement under the interpretation given to every type of item.

The derivation in INITIAL PREDICT and PREDICT is sound because the item is derived from existing production and the string before the dot is empty so:

$$\mathcal{K}\ \sigma\ \epsilon = \epsilon$$

The rationale for SCAN is that if

$$\mathcal{K}\ \sigma\ \alpha = w_{j-1} \ldots w_k$$

and $s = w_{k+1}$ then

$$\mathcal{K}\ \sigma\ (\alpha\ s) = w_{j-1} \ldots w_{k+1}$$

If the item in the premise is valid then it is based on existing production and function and so will be the item in the consequent.

In the COMPLETE rule the dot is at the end of the string. This means that $w_{j+1} \ldots w_k$ will be not just a prefix in constituent l of the linearization but the full string. This is exactly what is required in the semantics of the passive item. The passive item is derived from a valid active item so there is at least one production for A. The category N is unique for each (A, l, j, k) quadruple so it uniquely identifies the passive item in which it is placed. There might be many productions that can produce the passive item but all of them should be able to generate $w_{j+1} \ldots w_k$ and they are exactly the productions that are added to N. From all this arguments it follows that COMPLETE is sound.

The COMBINE rule is sound because from the active item in the premise we know that:

$$\mathcal{K}\ \sigma\ \alpha = w_{j+1} \ldots w_u$$

for every context σ built from the trees:

$$t_1 : B_1;\ t_2 : B_2;\ \ldots t_{a(f)} : B_{a(f)}$$

From the passive item we know that every production for N produces the $w_{u+1} \ldots w_k$ in r. From that follows that

$$\mathcal{K}\ \sigma'\ (\alpha\langle d; r\rangle) = w_{j+1} \ldots w_k$$

where σ' is the same as σ except that B_d is replaced with N. Note that the last conclusion will not hold if we were using the original context because B_d is a more general category and can contain productions that does not derive $w_{u+1} \ldots w_k$.

5.3 Completeness

The parsing system is complete if it derives an item for every valid grammatical statement. In our case we have to prove that for every possible parse tree the corresponding items will be derived.

The proof for completeness requires the following lemma:

Lemma 1 *For every possible syntax tree*

$$(f\ t_1 \ldots t_{a(f)}) : A$$

with linearization

$$\mathcal{L}(f t_1 \ldots t_{a(f)}) = (x_1, x_2 \ldots x_{d(A)})$$

where $x_l = w_{j+1} \ldots w_k$, the system will derive an item $[^k_j A; l; A']$ if the item $[^k_j A \rightarrow f[\vec{B}]; l : \bullet\alpha_l]$ was predicted before that. We assume that the function definition is:

$$f := (\alpha_1, \alpha_2 \ldots \alpha_{r(f)})$$

The proof is by induction on the depth of the tree. If the tree has only one level then the function f does not have arguments and from the linearization definition and from the premise in the lemma it follows that $\alpha_l = w_{j+1} \ldots w_k$. From the active item in the lemma

by applying iteratively the SCAN rule and finally the COMPLETE rule the system will derive the requested item.

If the tree has subtrees then we assume that the lemma is true for every subtree and we prove it for the whole tree. We know that

$$\mathcal{K}\ \sigma\ \alpha_l = w_{j+1} \dots w_k$$

Since the function $\mathcal{K}$ does simple substitution it is possible for each $\langle d; s\rangle$ pair in α_l to find a new range in the input string $j' - k'$ such that the lemma to be applicable for the corresponding subtree $t_d : B_d$. The terminals in α_l will be processed by the SCAN rule. Rule PREDICT will generate the active items required for the subtrees and the COMBINE rule will consume the produced passive items. Finally the COMPLETE rule will derive the requested item for the whole tree.

From the lemma we can prove the completeness of the parsing system. For every possible tree $t : S$ such that $\mathcal{L}(t) = (w_1 \dots w_n)$ we have to prove that the $[^n_0 S; 1; S']$ item will be derived. Since the top-level function of the tree must be from production for S the INITIAL PREDICT rule will generate the active item in the premise of the lemma. From this and from the assumptions for t it follows that the requested passive item will be derived.

5.4 Complexity

The algorithm is very similar to the Earley (1970) algorithm for context-free grammars. The similarity is even more apparent when the inference rules in this paper are compared to the inference rules for the Earley algorithm presented in Shieber et al. (1995) and Ljunglöf (2004). This suggests that the space and time complexity of the PMCFG parser should be similar to the complexity of the Earley parser which is $\mathcal{O}(n^2)$ for space and $\mathcal{O}(n^3)$ for time. However we generate new categories and productions at runtime and this have to be taken into account.

Let the $\mathcal{P}(j)$ function be the maximal number of productions generated from the beginning up to the state where the parser has just consumed terminal number j. $\mathcal{P}(j)$ is also the upper limit for the number of categories created because in the worst case there will be only one production for each new category.

The active items have two variables that directly depend on the input size - the start index j and the end index k. If an item starts at position j then there are $(n - j + 1)$ possible values for k because $j \leq k \leq n$. The item also contains a production and there are $\mathcal{P}(j)$ possible choices for it. In total there are:

$$\sum_{j=0}^{n} (n - j + 1)\mathcal{P}(j)$$

possible choices for one active item. The possibilities for all other variables are only a constant factor. The $\mathcal{P}(j)$ function is monotonic because the algorithm only adds new productions and never removes. From that follows the inequality:

$$\sum_{j=0}^{n} (n - j + 1)\mathcal{P}(j) \leq \mathcal{P}(n) \sum_{i=0}^{n} (n - j + 1)$$

which gives the approximation for the upper limit:

$$\mathcal{P}(n)\frac{n(n+1)}{2}$$

The same result applies to the passive items. The only difference is that the passive items have only a category instead of a full production. However the upper limit for the number of categories is the same. Finally the upper limit for the total number of active, passive and production items is:

$$\mathcal{P}(n)(n^2 + n + 1)$$

The expression for $\mathcal{P}(n)$ is grammar dependent but we can estimate that it is polynomial because the set of productions corresponds to the compact representation of all parse trees in the context-free approximation of the grammar. The exponent however is grammar dependent. From this we can expect that asymptotic space complexity will be $\mathcal{O}(n^e)$ where e is some parameter for the grammar. This is consistent with the results in Nakanishi et al. (1997) and Ljunglöf (2004) where the exponent also depends on the grammar.

The time complexity is proportional to the number of items and the time needed to derive one item. The time is dominated by the most complex rule which in this algorithm is COMBINE. All variables that depend on the input size are present both in the premises and in the consequent except u. There are n possible values for u so the time complexity is $\mathcal{O}(n^{e+1})$.

5.5 Tree Extraction

If the parsing is successful we need a way to extract the syntax trees. Everything that we need is already in the set of newly generated productions. If the goal item is $[^n_0 S; 0; S']$ then every tree t of category S' that can be constructed is a syntax tree for the input sentence (see definition 2 in section 3 again).

Note that the grammar can be erasing; i.e., there might be productions like this:

$$S \to f[B_1, B_2, B_3]$$
$$f := (\langle 1; 1\rangle\langle 3; 1\rangle)$$

There are three arguments but only two of them are used. When the string is parsed this will generate a new specialized production:

$$S' \to f[B'_1, B_2, B'_3]$$

Here S,B_1 and B_3 are specialized to S', B'_1 and B'_3 but the B_2 category is still the same. This is correct

because actually any subtree for the second argument will produce the same result. Despite this it is sometimes useful to know which parts of the tree were used and which were not. In the GF interpreter such unused branches are replaced by meta variables. In this case the tree extractor should check whether the category also exists in the original set of categories N in the grammar.

Just like with the context-free grammars the parsing algorithm is polynomial but the chart can contain exponential or even infinite number of trees. Despite this the chart is a compact finite representation of the set of trees.

6 Implementation

Every implementation requires a careful design of the data structures in the parser. For efficient access the set of items is split into four subsets: $\mathbb{A}$, $\mathbb{S}_j$, $\mathbb{C}$ and $\mathbb{P}$. $\mathbb{A}$ is the agenda i.e. the set of active items that have to be analyzed. $\mathbb{S}_j$ contains items for which the dot is before an argument reference and which span up to position j. $\mathbb{C}$ is the set of possible continuations i.e. a set of items for which the dot is just after a terminal. $\mathbb{P}$ is the set of productions. In addition the set $\mathbb{F}$ is used internally for the generatation of fresh categories. The sets $\mathbb{C}$, $\mathbb{S}_j$ and $\mathbb{F}$ are used as association maps. They contain associations like $k \mapsto v$ where k is the key and v is the value. All maps except $\mathbb{F}$ can contain more than one value for one and the same key.

The pseudocode of the implementation is given in figure 2. There are two procedures *Init* and *Compute*.

Init computes the initial values of $\mathbb{S}$, $\mathbb{P}$ and $\mathbb{A}$. The initial agenda $\mathbb{A}$ is the set of all items that can be predicted from the start category S (INITIAL PREDICT rule).

Compute consumes items from the current agenda and applies the SCAN, PREDICT, COMBINE or COMPLETE rule. The case statement matches the current item against the patterns of the rules and selects the proper rule. The PREDICT and COMBINE rules have two premises so they are used in two places. In both cases one of the premises is related to the current item and a loop is needed to find item matching the other premis.

The passive items are not independent entities but are just the combination of key and value in the set $\mathbb{F}$. Only the start position of every item is kept because the end position for the interesting passive items is always the current position and the active items are either in the agenda if they end at the current position or they are in the $\mathbb{S}_j$ set if they end at position j. The active items also keep only the dot position in the constituent because the constituent definition can be retrieved from the grammar. For this reason the runtime representation of the items is $[j; A \to f[\vec{B}]; l; p]$ where j is the start position of the item and p is the dot position inside the constituent.

The *Compute* function returns the updated $\mathbb{S}$ and $\mathbb{P}$ sets and the set of possible continuations $\mathbb{C}$. The set of continuations is a map indexed by a terminal and the values are active items. The parser computes the set of continuations at each step and if the current terminal is one of the keys the set of values for it is taken as an agenda for the next step.

Language	Productions	Constituents
Bulgarian	3516	75296
English	1165	8290
German	8078	21201
Swedish	1496	8793

Table 1: GF Resource Grammar Library size in number of PMCFG productions and discontinuous constituents

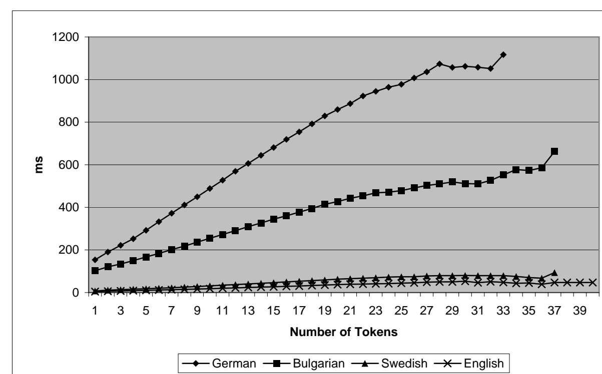

Figure 3: Parser performance in miliseconds per token

7 Evaluation

The algorithm was evaluated with four languages from the GF resource grammar library (Ranta, 2008): Bulgarian, English, German and Swedish. These grammars are not primarily intended for parsing but as a resource from which smaller domain dependent grammars are derived for every application. Despite this, the resource grammar library is a good benchmark for the parser because these are the biggest GF grammars.

The compiler converts a grammar written in the high-level GF language to a low-level PMCFG grammar which the parser can use directly. The sizes of the grammars in terms of number of productions and number of unique discontinuous constituents are given on table 1. The number of constituents roughly corresponds to the number of productions in the context-free approximation of the grammar. The parser performance in terms of miliseconds per token is shown in figure 3. In the evaluation 34272 sentences were parsed and the average time for parsing a given number of tokens is drawn in the chart. As it can be seen, although the theoretical complexity is polynomial, the real-time performance for practically interesting grammars tends to be linear.

8 Conclusion

The algorithm has proven useful in the GF system. It accomplished the initial goal to provide suggestions

```
procedure Init() {
   k = 0
   𝕊_i = ∅,    for every i
   ℙ = the set of productions P in the grammar

   𝔸 = ∅
   forall S → f[B⃗] ∈ P do    // INITIAL PREDICT
      𝔸 = 𝔸 + [0; S → f[B⃗]; 1; 0]

   return (𝕊, ℙ, 𝔸)
}


procedure Compute(k, (𝕊, ℙ, 𝔸)) {
   ℂ = ∅
   𝔽 = ∅
   while 𝔸 ≠ ∅ do {
      let x ∈ 𝔸,    x ≡ [j; A → f[B⃗]; l; p]
      𝔸 = 𝔸 − x
      case the dot in x is {
         before s ∈ T ⇒ ℂ = ℂ + (s ↦ [j; A → f[B⃗]; l; p + 1])    // SCAN

         before ⟨d; r⟩  ⇒ if ((B_d, r) ↦ (x, d)) ∉ 𝕊_k then {
                              𝕊_k = 𝕊_k + ((B_d, r) ↦ (x, d))
                              forall B_d → g[C⃗] ∈ ℙ do    // PREDICT
                                 𝔸 = 𝔸 + [k; B_d → g[C⃗]; r; 0]
                           }
                           forall (k; B_d, r) ↦ N ∈ 𝔽 do    // COMBINE
                              𝔸 = 𝔸 + [j; A → f[B⃗{d := N}]; l; p + 1]

         at the end     ⇒ if ∃N.((j, A, l) ↦ N ∈ 𝔽) then {
                              forall (N, r) ↦ (x′, d′) ∈ 𝕊_k do       // PREDICT
                                 𝔸 = 𝔸 + [k; N → f[B⃗]; r; 0]
                           } else {
                              generate fresh N       // COMPLETE
                              𝔽 = 𝔽 + ((j, A, l) ↦ N)
                              forall (A, l) ↦ ([j′; A′ → f′[B⃗′]; l′; p′], d) ∈ 𝕊_j do    // COMBINE
                                 𝔸 = 𝔸 + [j′; A′ → f′[B⃗′{d := N}]; l′; p′ + 1]
                           }
                           ℙ = ℙ + (N → f[B⃗])
      }
   }

   return (𝕊, ℙ, ℂ)
}
```

Figure 2: Pseudocode of the parser implementation

in text based dialog systems and in editors for controlled languages. Additionally the algorithm has properties that were not envisaged in the beginning. It works with PMCFG directly rather that by approximation with MCFG or some other weaker formalism.

Since the Linear Context-Free Rewriting Systems, Finite-Copying Tree Transducers and Tree Adjoining Grammars can be converted to PMCFG, the algorithm presented in this paper can be used with the converted grammar. The approach to represent context-dependent grammar as infinite context-free grammar might be applicable to other formalisms as well. This will make it very attractive in applications where some of the other formalisms are already in use.

References

Håkan Burden and Peter Ljunglöf. 2005. Parsing linear context-free rewriting systems. In *Proceedings of the Ninth International Workshop on Parsing Technologies (IWPT)*, pages 11–17, October.

Jay Earley. 1970. An efficient context-free parsing algorithm. *Commun. ACM*, 13(2):94–102.

Aravind Joshi and Yves Schabes. 1997. Tree-adjoining grammars. In Grzegorz Rozenberg and Arto Salomaa, editors, *Handbook of Formal Languages. Vol 3: Beyond Words*, chapter 2, pages 69–123. Springer-Verlag, Berlin/Heidelberg/New York.

Peter Ljunglöf. 2004. *Expressivity and Complexity of the Grammatical Framework*. Ph.D. thesis, Department of Computer Science, Gothenburg University and Chalmers University of Technology, November.

Ryuichi Nakanishi, Keita Takada, and Hiroyuki Seki. 1997. An Efficient Recognition Algorithm for Multiple ContextFree Languages. In *Fifth Meeting on Mathematics of Language*. The Association for Mathematics of Language, August.

Aarne Ranta. 2004. Grammatical Framework: A Type-Theoretical Grammar Formalism. *Journal of Functional Programming*, 14(2):145–189, March.

Aarne Ranta. 2008. GF Resource Grammar Library. `digitalgrammars.com/gf/lib/`.

Hiroyuki Seki and Yuki Kato. 2008. On the Generative Power of Multiple Context-Free Grammars and Macro Grammars. *IEICE-Transactions on Info and Systems*, E91-D(2):209–221.

Hiroyuki Seki, Takashi Matsumura, Mamoru Fujii, and Tadao Kasami. 1991. On multiple context-free grammars. *Theoretical Computer Science*, 88(2):191–229, October.

Hiroyuki Seki, Ryuichi Nakanishi, Yuichi Kaji, Sachiko Ando, and Tadao Kasami. 1993. Parallel Multiple Context-Free Grammars, Finite-State Translation Systems, and Polynomial-Time Recognizable Subclasses of Lexical-Functional Grammars. In *31st Annual Meeting of the Association for Computational Linguistics*, pages 130–140. Ohio State University, Association for Computational Linguistics, June.

Stuart M. Shieber, Yves Schabes, and Fernando C. N. Pereira. 1995. Principles and Implementation of Deductive Parsing. *Journal of Logic Programming*, 24(1&2):3–36.