Theory of GF and Parsing, Hints for Efficient Grammars

Krasimir Angelov

Chalmers University of Technology

August 21, 2009
1. Introduction

2. GF Core

3. Optimizations

4. Debugging

5. Conclusion
The GF language is:
- domain specific - Grammars first of all
- declarative - What vs How

The GF system is:
- optimizing compiler and interpreter
- not as smart as you

Although the language is declarative, the compiler needs some help to produce efficient grammars
There are two major operations that someone could do with a grammar:

- **Linearization**
  - **Efficient** - mapping from tree to string
- **Parsing**
  - **Search Problem** - find the tree(s) that produce a given string

Speaking about the efficiency I mean efficient parsing and compact grammar.
The GF language is too complex to be used directly for efficient parsing.

The compiler transforms it into simpler **GF Core** language.

The efficiency of the grammar depends on the GF Core.

Small Core $\equiv$ Efficient Grammar

\[
GF \Rightarrow GF\ Core
\]

\[
GF\ Core \equiv PMCFG
\]
Parallel Multiple Context-Free Grammar (PMCFG)

- Well known grammar formalism (Seki at al., 1991)
- Natural extension of CFG that produces tuples of strings instead of simple strings
- It is trivial to implement classical context-sensitive languages - \( \{a^n b^n c^n | n > 0\} \):

\[
\begin{align*}
\text{fun } z &= <"","",""> \\
        s x &= <"a" ++ x.p1,"b" ++ x.p2,"c" ++ x.p3> \\
c x &= <x.p1 ++ x.p2 ++ x.p3>
\end{align*}
\]
Joshi Aravind. 1991. Tree Adjoining Grammars: How much context-sensitivity is required to provide reasonable structural descriptions?

- The language must be parsable in polynomial time.
- The language must have constant growth.
- The language should admit limited cross-serial dependencies.

Example language: \( \{a^n b^n c^n | n > 0 \} \) and all MCFG
More than Mildly Context-Sensitive Languages

The exponential language $\{a^{2^n} | n > 0\}$:

$$\text{fun } z \text{ = } < "a" >$$
$$s \times = < x.p1 ++ x.p1 >$$

is not Mildly Context-Sensitive (reduplication).
Four non-MCS Natural Languages

- Mandarin Chinese numeral names
- Mandarin Chinese yes/no questions
- Old Georgian case system
- Lindenmayer system
Mandarin Chinese numeral names

- **Concatenation of ten thousands**
  
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>wan</td>
<td>10 000</td>
</tr>
<tr>
<td>wan wan</td>
<td>yi</td>
</tr>
<tr>
<td>wan yi</td>
<td>zhao</td>
</tr>
</tbody>
</table>

- **Composite numbers**
  
  wu zhao zhao wu zhao
  five trillion trillion five trillion

  5 000 000 000 005 000 000 000 000

- **Formally**

  \[ \{ \text{wu zhao}^{k_1} \text{ wu zhao}^{k_2} \ldots \text{ wu zhao}^{k_n} \mid k_1 > k_2 > \ldots > k_n \} \]
Mandarin Chinese yes/no questions

Does Zhangsan like to play basketball?

Zhangsan likes to play basketball, not to play volleyball
Old Georgian case system

Example:

tkuenda micemul ars cnob-ad saidumlo-j
to you given is knowing-Adv mistery-Nom

igi sasupevel-isa m-is ymrt-isa-jsa-j
Art=Nom kingdom-Gen Art-Gen god-Gen-Gen-Nom

Onto You it is given to know the mistery of the kingdom of God
Mark 4:11

Formal

\[ N_1-\text{Nom} \quad N_2-\text{Gen-Nom} \quad N_3-\text{Gen}^2-\text{Nom} \ldots N_k-\text{Gen}^{k-1}-\text{Nom} \]
Lindenmayer system

- Mathematical objects
- The structure of some plants
- The growth of some crystals
- Symmetry in music

\[
\begin{align*}
A & \rightarrow a \\
B & \rightarrow b \\
A & \rightarrow BRARB \\
B & \rightarrow ALBLA
\end{align*}
\]

The reduplication is a norm!
The current status of parsing with GF

Angelov. 2009. Incremental Parsing with PMCFG

Features:
- Very Efficient (polynomial - close to linear)
- Supports PMCFG for free
- PMCFG allows more compact grammars
- It is incremental !!!

Things to consider:
- the GF ⇒ GF Core conversion is often exponential
- The grammar should be carefully written to avoid combinatorial explosion
- In practice careful means linguistically motivated
**The Rules of The Game**

**Initial Predict**

\[
S \rightarrow f[\vec{B}] \\
[0] S \rightarrow f[\vec{B}]; 1 : \bullet \alpha
\]

\(S\) - start category, \(\alpha = \text{rhs}(f, 1)\)

**Predict**

\[
B_d \rightarrow g[\vec{C}] \\
[j^k A \rightarrow f[\vec{B}]; l : \alpha \bullet \langle d; r \rangle \beta] \\
k^k B_d \rightarrow g[\vec{C}]; r : \bullet \gamma
\]

\(\gamma = \text{rhs}(g, r)\)

**Scan**

\[
j^k A \rightarrow f[\vec{B}]; l : \alpha \bullet s \beta
\]

\(s = w_{k+1}\)

**Complete**

\[
j^k A \rightarrow f[\vec{B}]; l : \alpha \bullet
\]

\(N \rightarrow f[\vec{B}] \\
[j^k A]; l; N
\]

\(N = (A, l, j, k)\)

**Combine**

\[
j^u A \rightarrow f[\vec{B}]; l : \alpha \bullet \langle d; r \rangle \beta \\
[k^u B_d; r; N]
\]

\(j^k A \rightarrow f[\vec{B}\{d := N\}]; l : \alpha \langle d; r \rangle \bullet \beta\)
Parsing with the resource library

Note: Much faster with application grammars
Parsing of $\{a^{2^n} \mid n > 0\} - O(n \log n)$
1 Introduction

2 GF Core

3 Optimizations

4 Debugging

5 Conclusion
The parser uses language which is simplified version of GF.

- The linearization types are flat tuples of strings:

  \[ \text{lincat } C = \text{Str} \ast \text{Str} \ast \ldots \ast \text{Str}; \]

- The linearizations are simple concatenations:

  \[ \text{lin } f x y = \langle x.p1, x.p2 ++ y.p3 \rangle; \]

- No operations are allowed
- No variants are allowed
- No parameters and tables
- No pattern matching
- No gluing is allowed (i.e. \( \oplus \) but not \( + \))
GF ⇒ GF Core

- Operations elimination
- Variants elimination
- Parameter types elimination
- Linearization rules transformations
- Common subexpressions optimization
The operations are **NONRECURSIVE** functions. They are evaluated at compile time. *(macroses)*

### GF

```plaintext
oper mkN noun = case noun of {
    _ + "s" ⇒ < noun, noun + "es" >;
    _     ⇒ < noun, noun + "s" >
};

lin apple_N = mkN "apple";
plus_N = mkN "plus";
```

### GF Core

```plaintext
lin apple_N = < "apples" >;
plus_N = < "pluses" >;
```

*Note: the pattern matching in mkN was eliminated*
Hints for Operations

Since the operations are computed at compile time this doesn’t affect the runtime efficiency. However they affect the compilation speed \((slightly)\).
Variants elimination

The variants are just expanded:

\[
\text{lin } \text{girl}_N = \text{mkN} \ (\text{variants } \{"tjej"; "flicka"\});
\]

GF Core

\[
\begin{align*}
\text{lin } \text{girl}_N_1 &= \text{mkN } "tjej"; \\
\text{girl}_N_2 &= \text{mkN } "flicka";
\end{align*}
\]

Note: Appropriate for application specific grammars. Should be avoided in resource grammars.
Variants are not always what you want

GF

```
lin Answer pol verb = "I" ++ pol ++ verb;
    eat = "eat";
    like = "like";
    Pos = "";
    Neg = variants{"do not";"don’t"};

Comp s1 s2 = s1 ++ ";" ++ s2;
```

Comp (Answer Neg like) (Answer Neg eat)

I don’t like; I don’t eat
I don’t like; I do not eat
I do not like; I don’t eat
I do not like; I do not eat
Variants are not always what you want

\[ \text{lin Answer pol verb} = \text{!!style} \Rightarrow "I" + + (pol!style) + + \text{verb}; \]
\[ \text{eat} = "eat"; \]
\[ \text{like} = "like"; \]
\[ \text{Pos} = \text{table}\{\text{Official} \Rightarrow ""; \text{Spoken} \Rightarrow ""\}; \]
\[ \text{Neg} = \text{table}\{\text{Official} \Rightarrow "do not"; \text{Spoken} \Rightarrow "don't" \}; \]

\[ \text{Comp ids1 ids2} = \text{variants}\{\text{comp Official}; \text{comp Spoken}\}; \]

\[ \text{oper comp style} = \text{s1!style} + + ";" + + \text{s2!style}; \]

\[ \text{Comp (Answer Neg like) (Answer Neg eat)} \]

I don’t like; I don’t eat
I do not like; I do not eat
The variants could blow up

When many variants are used in parallel the number of core rules grows exponentially.

**GF**

\[
\text{lin start_word} = \text{variants}\{"open";"start"\} \oplus \text{variants}\{"Word";"Microsoft Word"\};
\]

**GF Core**

\[
\text{lin start_word}_1 = "open" \oplus "Word";
\text{start_word}_2 = "open" \oplus "Microsoft Word";
\text{start_word}_3 = "start" \oplus "Word";
\text{start_word}_4 = "start" \oplus "Microsoft Word";
\]
Variants explosion with tables

\[
\text{lin } \text{close\textunderscore word} = \text{table } Tense \{ \text{"close" } + + \text{ variants}\{\text{"Word" ; } \text{"Microsoft Word"}\} ; \\
\text{oper } \text{close\textunderscore V} = \text{table } Tense \{ \text{"closed" } + + \text{ variants}\{\text{"Word" ; } \text{"Microsoft Word"}\} ; \\
\text{"have closed" } + + \text{ variants}\{\text{"Word" ; } \text{"Microsoft Word"}\} \}
\]

\[
\text{Note: Leads to } 2^3 = 8 \text{ possible combinations although there is only one variant in the original code}
\]
<table>
<thead>
<tr>
<th>Line Number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>close word₁ = &quot;close&quot; ++ &quot;Word&quot;;</td>
</tr>
<tr>
<td></td>
<td>&quot;closed&quot; ++ &quot;Word&quot;;</td>
</tr>
<tr>
<td></td>
<td>&quot;have closed&quot; ++ &quot;Word&quot; &gt;;</td>
</tr>
<tr>
<td>2</td>
<td>close word₂ = &quot;close&quot; ++ &quot;Microsoft Word&quot;;</td>
</tr>
<tr>
<td></td>
<td>&quot;closed&quot; ++ &quot;Word&quot;;</td>
</tr>
<tr>
<td></td>
<td>&quot;have closed&quot; ++ &quot;Word&quot; &gt;;</td>
</tr>
<tr>
<td>3</td>
<td>close word₃ = &quot;close&quot; ++ &quot;Word&quot;;</td>
</tr>
<tr>
<td></td>
<td>&quot;closed&quot; ++ &quot;Microsoft Word&quot;;</td>
</tr>
<tr>
<td></td>
<td>&quot;have closed&quot; ++ &quot;Word&quot; &gt;;</td>
</tr>
<tr>
<td>4</td>
<td>close word₄ = &quot;close&quot; ++ &quot;Microsoft Word&quot;;</td>
</tr>
<tr>
<td></td>
<td>&quot;closed&quot; ++ &quot;Microsoft Word&quot;;</td>
</tr>
<tr>
<td></td>
<td>&quot;have closed&quot; ++ &quot;Word&quot; &gt;;</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>
lin close_word = variants{closeIt "Word"; closeIt "Microsoft Word"};
oper closeIt obj = \ \tense ⇒ close_V!tense ++ obj;
   close_V = table Tense {"close";"closed";"have closed";...};
Hints for Variants

- Variants should be used with care
- A variant in the wrong place could lead too many combinations, which is often not what you want
- A combinatorial explosion could kill the compiler with out of memory

Out of memory

- Unnecessary combinations will slow down the parser
- Hint: use the command ’l -all’
**Parameter Types Elimination**

\[ \text{lincat} \ NP = \{ s : \text{Case} \Rightarrow \text{Str}; \ g : \text{Gender}; \ n : \text{Number}; \ p : \text{Person} \} \]

**param**

\begin{align*}
\text{Case} &= \text{Nom} | \text{Acc} | \text{Dat}; \\
\text{Gender} &= \text{Masc} | \text{Fem} | \text{Neutr}; \\
\text{Number} &= \text{Sg} | \text{Pl}; \\
\text{Person} &= \text{P1} | \text{P2} | \text{P3};
\end{align*}
A value of type $\text{Case } \Rightarrow \text{Str}$ looks like:

$$\text{table } \{ \text{Nom } \Rightarrow s_1; \text{ Acc } \Rightarrow s_2; \text{ Dat } \Rightarrow s_3 \}$$

We could replace it with tuple:

$$\langle s_1, s_2, s_3 \rangle$$

Then in general type like $A \Rightarrow \text{Str}$ is equivalent to:

$$\underbrace{\text{Str} \ast \text{Str} \ast \ldots \ast \text{Str}}_{n \text{ times}}$$

where $n$ is the number of values in the parameter type $A$. 
Parameter Fields Elimination

**GF**

\[
\text{lincat } NP = \{ s : \ldots ; g : \text{Gender}; n : \text{Number}; p : \text{Person} \}
\]

**GF Core**

\[
\begin{align*}
\text{lincat } NP_1 &= \text{Str} \ast \text{Str} \ast \text{Str}; & \quad & \text{– Masc; Sg, P1} \\
NP_2 &= \text{Str} \ast \text{Str} \ast \text{Str}; & \quad & \text{– Masc; Sg, P2} \\
NP_3 &= \text{Str} \ast \text{Str} \ast \text{Str}; & \quad & \text{– Masc; Sg, P3} \\
NP_4 &= \text{Str} \ast \text{Str} \ast \text{Str}; & \quad & \text{– Masc; Pl, P1} \\
\ldots \\
NP_{18} &= \text{Str} \ast \text{Str} \ast \text{Str}; & \quad & \text{– Neutr; Pl, P3}
\end{align*}
\]

*Note: The number of categories doesn’t immediately affect the size of the compiled grammar*
It is important to know how many possible values a given parameter type has because:

- This determines the number of fields in the core:
  \[ P \Rightarrow Str \]

- This determines the number of categories in the core:
  \[ \{ \ldots ; p : P \} \]
Counting the number of parameter values

**Parameter Definition**

\[
\text{param } P = P_1 \quad Q_{11} \quad Q_{12} \ldots Q_{1m_1} \\
| \quad P_2 \quad Q_{21} \quad Q_{22} \ldots Q_{2m_2} \\
\ldots \\
| \quad P_n \quad Q_{n1} \quad Q_{n2} \ldots Q_{nm_n}
\]

**Values Count**

\[
C(P) = C(Q_{11}) \times C(Q_{12}) \ldots C(Q_{1m_1}) \\
+ C(Q_{21}) \times C(Q_{22}) \ldots C(Q_{2m_2}) \\
\ldots \\
+ C(Q_{n1}) \times C(Q_{n2}) \ldots C(Q_{nm_n})
\]
Counting Parametric Tables and Records

### Parametric Records

\[ \mathbb{C}(\{q_1 : Q_1; q_2 : Q_2 \ldots q_n : Q_n\}) = \mathbb{C}(Q_1) \ast \mathbb{C}(Q_2) \ldots \mathbb{C}(Q_n) \]

### Parametric Tables

\[ \mathbb{C}(P \Rightarrow Q) = \mathbb{C}(Q)^{\mathbb{C}(P)} \]

**Warning:** Exponentials should be avoided!!!
Hints for Parameters

- Keep the lexicon compact:

  \[
  \text{lincat } N = \{ s : NForm \Rightarrow \text{Str}; \ g : DGender \};
  \]

- **param** NForm
  
  \[
  = NF \ \text{Number Species} \\
  | \ NFSgDefNom \\
  | \ NFPICount \\
  | \ NFVocative
  \]

- **param** NForm
  
  \[
  = NF \ \text{Number Species Case} \\
  | \ NFPICount
  \]

- Linguistically accurate
- The irregularity is obvious
- Mathematically elegant
- Linguistically overgenerating

**Comment**

The lexical items are inflection tables. Duplication means overhead for every entry in the lexicon.
Hints for Parameters

- Keep the syntax elegant:

  \[
  \text{lincat } CN = \{ s : \text{Number} \implies \text{Species} \implies \text{Case} \implies \text{Str} \};
  \]

Comment

The syntactic rules are closed set. Compared to the lexicon this is a small set so it is not so important to make them compact. It is much more important to have clear easy to manipulate structure.

The efficiency of the parser is not affected by the number of fields in the linearization types.
Hints for Parameters

- Minimize the number of inherent parameters

\[
\text{lincat } N = \{ s : N\text{Form} \Rightarrow \text{Str}; \ g : \text{DGender} \};
\]

\[
\begin{align*}
\text{param } \text{DGender} &= \text{DMasc Animacy} \\
&= \text{DFem} \\
&= \text{DNeutr}
\end{align*}
\]

\[
\begin{align*}
\text{oper } \text{DGender} &= \{ g : \text{Gender}; \ a : \text{Animacy} \} \\
\text{param } \text{Gender} &= \text{Masc} \\
&= \text{Fem} \\
&= \text{Neutr};
\end{align*}
\]

\[
\begin{align*}
\text{param } \text{Animacy} &= \text{Animate} | \text{Inanimate};
\end{align*}
\]

- Animacy matters only for Masc
- Animacy given for all genders
Linearization Rules Transformation

GF

fun AdjCN : AP → CN → CN;
lin AdjCN ap cn = {
  s = ap.s!cn.g ++ cn.s;
  g = cn.g
};

GF Core

fun AdjCN₁ : AP → CN₁ → CN₁; – Masc
lin AdjCN₁ ap cn = < ap.p1 ++ cn.p1 >

fun AdjCN₂ : AP → CN₂ → CN₂; – Fem
lin AdjCN₂ ap cn = < ap.p2 ++ cn.p1 >

fun AdjCN₃ : AP → CN₃ → CN₃; – Neutr
lin AdjCN₃ ap cn = < ap.p3 ++ cn.p1 >
In general linearization rule like:

$$\textbf{fun } f : A_1 \rightarrow A_2 \rightarrow \ldots \rightarrow A_n \rightarrow A;$$

produces $\mathcal{C}(f)$ rules in the core

$$\mathcal{C}(f) = \mathcal{C}(A_1) \ast \mathcal{C}(A_2) \ast \ldots \ast \mathcal{C}(A_n)$$

**Comment**

The number of rules could be reduced by reducing the number of parameters in the linearization types. The count is also reduced by the optimizations in the compiler.
No pattern matching

**Allowed**

```haskell
oper mkN noun = case noun of {
  _ + "+"s" ⇒ < noun, noun + "es" >;
  _ ⇒ < noun, noun + "s" >
};
```

**Not Allowed**

```haskell
lin DetCN det cn = case det.s of {
  "" ⇒ . . .
  _ ⇒ . . .
};
```

*Hint: use parameter which says whether the string is empty*
No gluing

<table>
<thead>
<tr>
<th>Allowed</th>
</tr>
</thead>
</table>
|\[
\text{lin } \text{DetCN} \ \text{det} \ \text{cn} = \text{case } \text{det}\.\text{spec} \ \text{of} \ \{ \\
\quad \ldots \\
\quad \text{Indefinite } \Rightarrow \text{case } \text{cn}\.\text{g} \ \text{of} \ \{ \text{Utr } \Rightarrow "\text{en}" ; \text{Neutr } \Rightarrow "\text{ett}" \} \ + \ \text{cn}\.s \\
\}\] |

<table>
<thead>
<tr>
<th>Not Allowed</th>
</tr>
</thead>
</table>
|\[
\text{lin } \text{DetCN} \ \text{det} \ \text{cn} = \text{case } \text{det}\.\text{spec} \ \text{of} \ \{ \\
\quad \text{Definite } \Rightarrow \text{cn}\.s \ + \ \text{case } \text{cn}\.\text{g} \ \text{of} \ \{ \text{Utr } \Rightarrow "\text{en}" ; \text{Neutr } \Rightarrow "\text{et}" \}; \\
\quad \ldots \\
\}\] |

*Hint: for agglutinative languages (Turkish, Finnish, Estonian, Hungarian, ...) use custom lexer*
Some languages have potentially infinite set of words:

Turkish:

anlamiyorum = anla(root) -mi(negation) -yor(continuous) -um(first person)
I don’t understand

The grammar could be based on roots and suffixes instead of words:

"anla" ++ "&+" ++ "mi" ++ "&+" ++ "yor" ++ "&+" ++ "um"

The lexer/unlexer are responsible to produce the real words
Three main optimizations reduce the exponential size of the grammar:

- Common Subexpressions Optimization
- Common Functions Optimization
- Coercion Rules

Note: the optimizations cannot be expressed in GF Core. PMCFG is needed.
Common Subexpressions Optimization

**GF Core**

\[
\text{lin } u \times y = <x.p1, x.p2 + + y.p1 >
\]

\[
v \times y = <"a", x.p2 + + y.p1 >
\]

**PMCFG**

\[
F_1 := (S_1, S_2) \quad [u]
\]

\[
F_2 := (S_3, S_2) \quad [v]
\]

\[
S_1 := \langle 0; 0 \rangle
\]

\[
S_2 := \langle 0; 1 \rangle \langle 1; 0 \rangle
\]

\[
S_3 := "a"
\]
Common Subexpressions Optimization in the Lexicon

**GF Core**

\[
\text{lin } \text{good}_A = < "\text{dobár}" , "\text{dobra}" , "\text{dobro}" , "\text{dobre}" > \\
\text{beautiful}_A = < "\text{hubav}" , "\text{hubava}" , "\text{hubavo}" , "\text{hubavo}" >
\]

**PMCFG**

\[
F_1 := (S_1, S_2, S_3, S_4) \ [\text{good}_A] \\
F_2 := (S_5, S_6, S_7, S_7) \ [\text{beautiful}_A]
\]

\[
S_1 := "\text{dobár}" \quad S_5 := "\text{hubav}"
\]
\[
S_2 := "\text{dobra}" \quad S_6 := "\text{hubava}"
\]
\[
S_3 := "\text{dobro}" \quad S_7 := "\text{hubavo}"
\]
\[
S_4 := "\text{dobre}"
\]
The function symbols in PMCFG could be reused in different productions

\[
C_1 \leftarrow F_1[C_2, C_3]
\]

\[
C_1 \leftarrow F_1[C_4, C_5]
\]

\[
F_1 := (S1, S2, S3, S4) \quad [u]
\]
Coercion Rules

PMCFG

\[ C_1 \leftarrow F_1[C_2, C_{31}, C_{41}, C_5] \]
\[ C_1 \leftarrow F_1[C_2, C_{32}, C_{41}, C_5] \]
\[ C_1 \leftarrow F_1[C_2, C_{33}, C_{41}, C_5] \]
\[ C_1 \leftarrow F_1[C_2, C_{31}, C_{42}, C_5] \]
\[ C_1 \leftarrow F_1[C_2, C_{32}, C_{42}, C_5] \]
\[ C_1 \leftarrow F_1[C_2, C_{33}, C_{42}, C_5] \]
\[ C_1 \leftarrow F_1[C_2, C_{31}, C_{43}, C_5] \]
\[ C_1 \leftarrow F_1[C_2, C_{32}, C_{43}, C_5] \]
\[ C_1 \leftarrow F_1[C_2, C_{33}, C_{43}, C_5] \]
Coercion Rules

PMCFG

\[
\begin{align*}
C_1 & \leftarrow F_1[C_2, C_3, C_4, C_5] \\
C_3 & \leftarrow [C_{31}] \\
C_3 & \leftarrow [C_{32}] \\
C_3 & \leftarrow [C_{33}] \\
C_4 & \leftarrow [C_{41}] \\
C_4 & \leftarrow [C_{42}] \\
C_4 & \leftarrow [C_{43}]
\end{align*}
\]
1. Introduction
2. GF Core
3. Optimizations
4. Debugging
5. Conclusion
You can dump the PMCFG representation of the grammar with the following command:

c:\gf> gf -make -output-format=pmcfg_pretty LangMy.gf
Reading Lang.pgf...
Refusing to overwrite Lang.pgf
Writing LangEng.pmcfg...

This will produce one file with extension .pmcfg with four interesting sections:
- productions
- functions
- sequences
- startcats
Thank You and Have Fun !!!