User documentation for CCPC

July 29, 2013

1 Getting started

This brief tutorial will walk through some basic uses of what can be thought of as the three "core" tools provided by CCPC:

- `visualize` provides an indication of what is likely to be generated by a given grammar.
- `parse` finds derivations of a given string licensed by a given grammar.
- `intersect` constructs new grammars by combining the constraints imposed by an existing grammar with further "external" constraints, such as compatibility with a given prefix.

We assume that you have checked out a working copy of the repository (or equivalent) and compiled successfully with `make`. All the commands that follow are to be run in the root directory of this working copy (i.e. the directory that contains `Makefile`).

1.1 A simple context-free grammar

We'll start by playing with the example grammar `strauss.wmcfg`:

```
$ cat grammars/wmcfg/strauss.wmcfg
1 / 1 S  --> np vp [0,0;1,0]
1 / 1 vp --> v1    [0,0]
1 / 4 v1 --> v1 pp [0,0;1,0]
3 / 4 v1 --> v  np [0,0;1,0]
1 / 1 pp --> p  np [0,0;1,0]
1 / 4 np --> pn    [0,0]
3 / 4 np --> dt n1 [0,0;1,0]
3 / 4 n1 --> n     [0,0]
1 / 4 n1 --> n1 pp [0,0;1,0]
1 / 1 v  --> "hit"
1 / 1 pn --> "Jon"
1 / 2 n  --> "dog"
1 / 2 n  --> "stick"
1 / 1 p  --> "with"
1 / 1 dt --> "the"
```

This defines a simple context-free phrase-structure grammar. There is one rule on each line. Each line begins with a fraction (e.g. `3 / 4`) that is the rule's weight. Nonterminals and terminals are distinguished by whether or not they are surrounded by double quotes (e.g. `"hit"` is a terminal, `np` is a nonterminal). For now, we will ignore the numbers in square brackets that come after the first nine rules.

We can ask for the 20 most likely derivations that are licensed by this grammar (i.e. the 20 derivations that have the highest weights) using `visualize`. In this command, `-g` specifies a grammar file to use and `-n` specifies the number of derivations to report.

```
$ ./visualize -g grammars/wmcfg/strauss.wmcfg -n 20
0.0593262   the dog hit the dog
0.0593262   the dog hit the stick
0.0593262   the stick hit the stick
0.0593262   the stick hit the dog
0.0527344   Jon hit the dog
0.0527344   the stick hit Jon
0.0527344   the dog hit Jon
0.0527344   Jon hit the stick
0.046875    Jon hit Jon
0.00417137  the dog hit the dog with the dog
0.00417137  the dog hit the dog with the stick
0.00417137  the dog hit the stick with the stick
0.00417137  the dog hit the stick with the dog
0.00417137  the dog hit the dog with the dog
0.00417137  the dog hit the dog with the stick
0.00417137  the dog hit the stick with the stick
0.00417137  the dog hit the stick with the dog
0.00417137  the stick hit the stick with the dog
0.00417137  the stick hit the stick with the stick
0.00417137  the stick hit the dog with the stick
```

These are the strings that are produced by the 20 most likely derivations, with the weight of the corresponding derivation at the start of each line.

Notice that the string `the dog hit the stick with the stick` appears twice.

We can ask for the two distinct derivation structures like this (`-g` specifies the grammar, `-i` specifies the input):

```
$ ./parse -g grammars/wmcfg/strauss.wmcfg -i "the dog hit the stick with the ▶
    ▶ stick"
0.00417137145996    (S (np (dt "the") (n1 (n "dog"))) (vp (v1 (v "hit") (np (dt ▶
    ▶ "the") (n1 (n1 (n "stick")) (pp (p "with") (np (dt "the") (n1 (n ▶
    ▶ "stick")))))))))
0.00417137145996    (S (np (dt "the") (n1 (n "dog"))) (vp (v1 (v1 (v "hit") (np ▶
    ▶ (dt "the") (n1 (n "stick")))) (pp (p "with") (np (dt "the") (n1 (n ▶
    ▶ "stick")))))))
```

The labelled bracketings here descibe tree structures in the usual way, with nonterminals from the grammar (e.g. `S`, `np`) labelling each constituent.

Considering prefixes

Suppose now we are interested not in *all* the derivations that are licensed by the grammar `strauss.wmcfg`, but rather in the more limited set of derivations that are both (a) licensed by the grammar, and (b) consistent with a certain initial portion of the derived string. For example, suppose we are interested in all those derivations that are licensed by the grammar `strauss.wmcfg` that produce sentences beginning with `Jon hit`. Even this more limited set of derivations is infinite (like the set of all derivations licensed by `strauss.wmcfg`), so there is no way to write down the entire list. In order to get our hands on it, we need some finite representation; we'll use a grammar as a finite representation of this set. The grammar that encapsulates this set — in the same way that `strauss.wmcfg` encapsulates a certain less restricted set of derivations — can be produced as follows:

```
$ ./intersect -g grammars/wmcfg/strauss.wmcfg -prefix "Jon hit"
(* original grammar: grammars/wmcfg/strauss.wmcfg *)
(* intersected with prefix: Jon hit *)
1 / 1      S_0-2 --> np_0-1 vp_1-2 [0,0;1,0]
1 / 4      np_0-1 --> pn_0-1 [0,0]
1 / 1      vp_1-2 --> v1_1-2 [0,0]
1 / 1      pn_0-1 --> "Jon"
3 / 4      v1_1-2 --> v_1-2 np_2-2 [0,0;1,0]
1 / 4      v1_1-2 --> v1_1-2 pp_2-2 [0,0;1,0]
1 / 1      pp_2-2 --> p_2-2 np_2-2 [0,0;1,0]
1 / 1      v_1-2 --> "hit"
1 / 4      np_2-2 --> pn_2-2 [0,0]
3 / 4      np_2-2 --> dt_2-2 n1_2-2 [0,0;1,0]
1 / 1      p_2-2 --> "with"
1 / 1      dt_2-2 --> "the"
3 / 4      n1_2-2 --> n_2-2 [0,0]
1 / 4      n1_2-2 --> n1_2-2 pp_2-2 [0,0;1,0]
1 / 1      pn_2-2 --> "Jon"
1 / 2      n_2-2 --> "stick"
1 / 2      n_2-2 --> "dog"
```

The first two lines (enclosed by `(*` and `*)`) are comments, useful for human users but not part of the grammar itself. The rest of the output is a grammar in the same format as the contents of `strauss.wmcfg`. Symbols like `np_0-1` are just atomic nonterminals.[1] (Again, we'll ignore for now the numbers in square brackets that appear to the right of some of the rules.)

Since the output of `intersect` is a grammar just like the contents of `strauss.wmcfg`, we can use `visualize` and `parse` to explore this new grammar. First we save it to a file:

```
./intersect -g grammars/wmcfg/strauss.wmcfg -prefix "Jon hit" > ▶
    ▶ strauss.Jon-hit.wmcfg
```

The best 20 derivations licensed by *this* grammar can be produced just like before, using the new file `strauss.Jon-hit.wmcfg`:

```
$ ./visualize -g strauss.Jon-hit.wmcfg -n 20
0.0527344   Jon hit the stick
```

[1]Can you guess what they "mean"?

```
0.0527344   Jon hit the dog
0.046875    Jon hit Jon
0.00370789  Jon hit the stick with the stick
0.00370789  Jon hit the stick with the dog
0.00370789  Jon hit the dog with the dog
0.00370789  Jon hit the dog with the stick
0.00370789  Jon hit the stick with the stick
0.00370789  Jon hit the stick with the dog
0.00370789  Jon hit the dog with the dog
0.00370789  Jon hit the dog with the stick
0.0032959   Jon hit Jon with the stick
0.0032959   Jon hit the dog with Jon
0.0032959   Jon hit the stick with Jon
0.0032959   Jon hit Jon with the dog
0.0032959   Jon hit the stick with Jon
0.0032959   Jon hit the dog with Jon
0.00292969  Jon hit Jon with Jon
0.000260711 Jon hit the stick with the stick with the stick
0.000260711 Jon hit the stick with the stick with the dog
```

Notice that all of these strings begin with `Jon hit`, as we would hope.

We can ask for the structures licensed by this new grammar that correspond to the string `Jon hit the dog with the stick`:

```
$ ./parse -g strauss.Jon-hit.wmcfg -i "Jon hit the dog with the stick"
0.00370788574219    (S_0-2 (np_0-1 (pn_0-1 "Jon")) (vp_1-2 (v1_1-2 (v_1-2 "hit") ▶
   ▶ (np_2-2 (dt_2-2 "the") (n1_2-2 (n1_2-2 (n_2-2 "dog")) (pp_2-2 (p_2-2 ▶
   ▶ "with") (np_2-2 (dt_2-2 "the") (n1_2-2 (n_2-2 "stick"))))))))))
0.00370788574219    (S_0-2 (np_0-1 (pn_0-1 "Jon")) (vp_1-2 (v1_1-2 (v1_1-2 ▶
   ▶ (v_1-2 "hit") (np_2-2 (dt_2-2 "the") (n1_2-2 (n_2-2 "dog")))) (pp_2-2 ▶
   ▶ (p_2-2 "with") (np_2-2 (dt_2-2 "the") (n1_2-2 (n_2-2 "stick")))))))
```

Notice that the nonterminal labels in the output are the "annotated" nonterminals that appear in the new grammar, so this is not identical to what we would get from parsing `Jon hit the dog with the stick` with the original `strauss.wmcfg`. But by ignoring the underscore and everything after it in the nonterminal names, we can "get back" the derivation in terms of the original grammar.

Since this new grammar only licenses sentences that begin with `Jon hit`, it is unable to derive the string `the dog hit the stick with the stick` that we parsed above with `strauss.wmcfg`:

```
$ ./parse -g strauss.Jon-hit.wmcfg -i "the dog hit the stick with the stick"
No derivations found
```

1.2 Minimalist Grammars

So far we have just been dealing with a context-free phrase structure grammar, but we can do all the same things with a minimalist grammar involving movement. Here are (the strings produced by) the top 10 derivations licensed by a certain minimalist grammar:

```
$ ./visualize -g grammars/wmcfg/larsonian1.wmcfg
0.00379689  he matter -ed
```

```
0.00315572  David matter -ed
0.00236679  Sally matter -ed
0.00221223  they matter -ed
0.00221223  I matter -ed
0.00144425  the treat be -s clever
0.00129323  the treat be -s young
0.00129323  the treat be -s right
0.00129323  the treat be -s poor
0.000991171 the treat be -s strange
```

(If we leave out the `-n` option to `visualize`, it defaults to 10.)

The set of derivations that these 10 were drawn from is, of course, defined by the grammar in `grammars/wmcfg/larsonian1.wmcfg`. The beginning of that file looks like this:

```
$ head grammars/wmcfg/larsonian1.wmcfg
12286 / 12286           S --> t158 [0,0;0,1;0,2]
12286 / 12286           t158 --> t0 t137 [0,0][0,1][0,2;1,0;1,1;1,2]
12286 / 12286           t0 --> E t0_tmp2 [0,0][1,0][1,1]
12286 / 12286           t0_tmp2 --> t0_tmp1 E [0,0][1,0]
12286 / 12286           t0_tmp1 --> " "
23639 / 23639           t137 --> t136 [0,3;0,0][0,1][0,2]
9855 / 9889           t133 --> t47 t86 [0,0][0,1][0,2;1,0;1,1;1,2][1,3]
10788 / 10788           t47 --> E t47_tmp2 [0,0][1,0][1,1]
10788 / 10788           t47_tmp2 --> t47_tmp1 E [0,0][1,0]
10788 / 10788           t47_tmp1 --> "be"
```

What the ...?! What on earth is that? It certainly doesn't look much like any of the stuff we know and love from minimalist syntax.

Indeed, this grammar is *not* a minimalist grammar. It is a *weighted multiple context-free grammar* (or WMCFG), which is the kind of grammar that the CCPC tools work with. It is, however, strongly equivalent to a certain minimalist grammar — i.e. it licenses, in effect, the same set of derivations — and so the top 10 derivations licensed by `larsonian1.wmcfg` are the same as the top 10 derivations licensed by the corresponding minimalist grammar.

To see the actual minimalist grammar to which `larsonian1.wmcfg` is equivalent, look at the contents of the file `grammars/mg/larsonian1.pl`. Understanding the full details of what's in this file requires some familiarity with the formulation of minimalist syntax developed by Ed Stabler,[2] but the basic idea is simple: this file defines a number of lexical entries, each of which has a surface form (e.g. `he`) along with a collection of features (e.g. `['D',-case]`) that comes after it, separated by a double-colon.

We will not go into any more detail here, but given the minimalist grammar in `larsonian1.pl` it is possible to automatically generate the strongly-equivalent WMCFG that the CCPC tools work with, and this is precisely how `larsonian1.wmcfg` was produced. In a way, one can think of `larsonian1.wmcfg` as a CCPC-readable version of the human-readable minimalist grammar `larsonian1.pl`.

[2]See for example "Derivational Minimalism" (1997) and "Computational perspectives on minimalism" (2011), both available at `http://www.linguistics.ucla.edu/people/stabler/writing.html`.

We can therefore use the other tools like `intersect` with `larsonian1.wmcfg` just as we did with `strauss.wmcfg` above. For example, we can find the 10 most likely derivations that produce sentences beginning with `Sally` as follows:

```
$ ./intersect -g grammars/wmcfg/larsonian1.wmcfg -prefix "Sally" > ▶
    ▶  larsonian1.Sally.wmcfg
```

```
$ ./visualize -g larsonian1.Sally.wmcfg
0.00236679  Sally matter -ed
0.000463883 Sally doesnt matter
0.000345626 Sally matter -s
0.000338299 Sally matter
0.000149766 Sally pay -ed for the treat
0.000101355 Sally sell -ed the treat
0.000101079 Sally tell -ed the treat
9.87318e-05 Sally get -ed the treat
9.87318e-05 Sally leave -ed the treat
9.53583e-05 Sally have -ed pay -en for the treat
```

Suppose we would like to look more closely at some of these top 10 derivations. Earlier, we used `parse` to display a labelled-bracketing illustrating hierarchical structure. Let's try the equivalent here for the sentence which yields `Sally sell -ed the treat`.

```
$ ./parse -g larsonian1.Sally.wmcfg -i "Sally sell -ed the treat"
0.000101355397899   (S_0-1 (t158_eps_eps_0-1 (t0_eps_eps_eps (E_eps " ") ▶
    ▶  (t0_tmp2_eps_eps (t0_tmp1_eps " ") (E_eps " "))) (t137_0-1_1-1_1-1 ▶
    ▶  (t136_1-1_1-1_1-1_0-1 (t42_1-1_1-1_1-1 (E_1-1 " ") (t42_tmp2_1-1_1-1 ▶
    ▶  (t42_tmp1_1-1 "-ed") (E_1-1 " "))) (t80_1-1_1-1_1-1_0-1 (t69_1-1_1-1_1-1 ▶
    ▶  (t37_1-1_1-1_1-1 (E_1-1 " ") (t37_tmp2_1-1_1-1 (t37_tmp1_1-1 " ") (E_1-1 " ▶
    ▶  "))) (t65_1-1_1-1_1-1 (t56_1-1_1-1_1-1_1-1 (t26_1-1_1-1_1-1 (E_1-1 " ") ▶
    ▶  (t26_tmp2_1-1_1-1 (t26_tmp1_1-1 "sell") (E_1-1 " "))) (t79_1-1_1-1_1-1 ▶
    ▶  (t4_1-1_1-1_1-1 (E_1-1 " ") (t4_tmp2_1-1_1-1 (t4_tmp1_1-1 "the") (E_1-1 " ▶
    ▶  "))) (t53_1-1_1-1_1-1 (t5_1-1_1-1_1-1 (E_1-1 " ") (t5_tmp2_1-1_1-1 ▶
    ▶  (t5_tmp1_1-1 " ") (E_1-1 " "))) (t10_1-1_1-1_1-1 (E_1-1 " ") ▶
    ▶  (t10_tmp2_1-1_1-1 (t10_tmp1_1-1 "treat") (E_1-1 " ")))))))) ▶
    ▶  (t7_eps_0-1_1-1 (E_eps " ") (t7_tmp2_0-1_1-1 (t7_tmp1_0-1 "Sally") (E_1-1 ▶
    ▶  " ")))))))
```

This isn't as useful as we might have hoped. The reason is that `parse` displays the structure of derivations in terms of the CCPC-readable WMCFG, not in terms of the human-readable minimalist grammar. In the earlier example with `strauss.wmcfg`, displaying the structure in WMCFG terms provided a relatively human-readable result, because `strauss.wmcfg` was itself a human-readable grammar. But for most intended uses of CCPC, human-readable results will generally come from `visualize`, not from `parse`.[3]

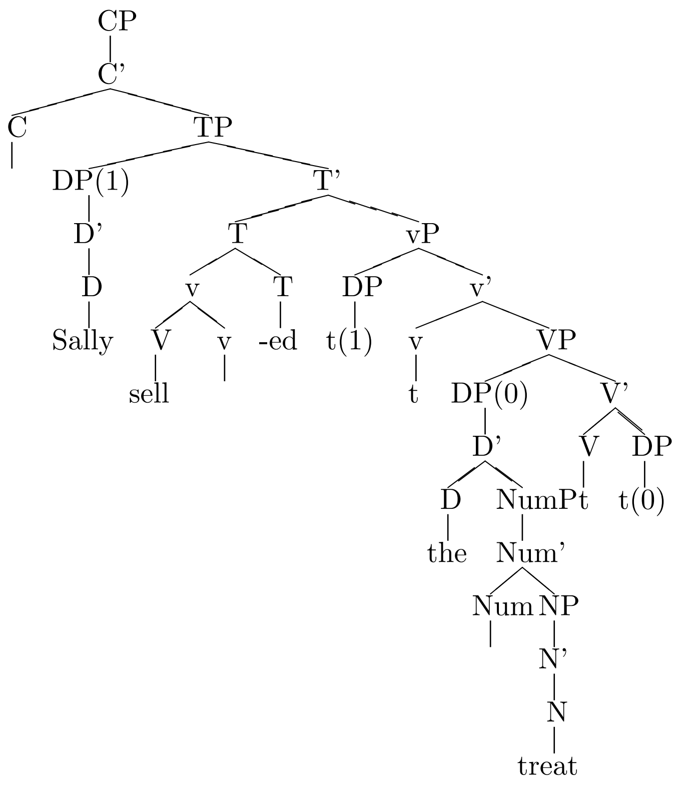
To ask `visualize` for a human-readable display of *structures* (as well as just the derived strings that it prints out), we provide the location for a LaTeX source file as an extra argument with `-o`:

```
$ ./visualize -g larsonian1.Sally.wmcfg -o larsonian1.Sally.top10.tex
0.00236679  Sally matter -ed
```

[3]`parse` is used mainly for debugging, or for other occasions where one really does need to look at the derivations from the point of view of the CCPC internals.

```
0.000463883 Sally doesnt matter
0.000345626 Sally matter -s
0.000338299 Sally matter
0.000149766 Sally pay -ed for the treat
0.000101355 Sally sell -ed the treat
0.000101079 Sally tell -ed the treat
9.87318e-05 Sally get -ed the treat
9.87318e-05 Sally leave -ed the treat
9.53583e-05 Sally have -ed pay -en for the treat
*** Output from Prolog tree-drawing: Given 10 derivations; found trees for 10 of ▶
    ▶ those
```

This will create the requested `larsonian1.Sally.top10.tex` file, which you can then compile (for example with `pdflatex`). At the beginning of the resulting document is some book-keeping information and a table that repeats the simple list of derived strings with their weights. This is followed by the relevant extra information, namely X-bar structures for the retrieved derivations. For example, it includes this X-bar structure corresponding to `Sally sell -ed the treat`:



In order to construct this X-bar tree, `visualize` consults both the WMCFG file that we provided on the command line (i.e. `larsonian1.Sally.wmcfg`) *and* the corresponding original minimalist grammar file `larsonian1.pl`; only the latter file talks about notions such as DPs and CPs and movement for Case, so without this file there would be no way to reconstruct the X-bar trees. For this reason, it is not possible to use the `-o` option with the grammar `strauss.wmcfg` used in the earlier examples: this is a "standalone" WMCFG that was not derived from a minimalist grammar, so if we ask `visualize` to produce a LaTeX output file showing derivations of this grammar, it can't:

```
$ ./visualize -g grammars/wmcfg/strauss.wmcfg -o out.tex
0.0593262   the dog hit the dog
0.0593262   the dog hit the stick
0.0593262   the stick hit the stick
0.0593262   the stick hit the dog
0.0527344   Jon hit the dog
0.0527344   the stick hit Jon
0.0527344   the dog hit Jon
0.0527344   Jon hit the stick
0.046875    Jon hit Jon
0.00417137  the dog hit the dog with the dog
Couldn't open dict file grammars/mcfgs/strauss.dict (Perhaps there is no MG file ▶
   ▶ from which grammars/wmcfg/strauss.wmcfg was derived?)
Couldn't write derivations to latex file
```

2 Generating and organizing grammars

All of the grammars used in §**??** were weighted multiple context-free grammars (WMCFGs), as indicated by the suffix `.wmcfg`. In this section we'll explain where these `.wmcfg` files come from. An overview of the relevant dependencies and workflow is given in Figure **??**.

We'll proceed in two steps: (i) how to come up with an unweighted MCFG file, and (ii) how to add weights to this file. (Of course, it is also possible to simply write the `.wmcfg` file by hand from scratch, or use any other tools one likes.)

2.1 How to come up with an unweighted MCFG file

MCFG files have a `.mcfg` suffix by convention.

Option A: Write the MCFG from scratch

The file format is described in Matthieu Guillaumin's report.

For users unfamiliar with MCFGs, it is relatively straightforward to encode a "normal" context-free grammar in this format too. The example grammar `strauss.wmcfg` is an illustration of this. The only parts of that file that look strange should be the zeros and ones in brackets on the right hand side of some of the rules; these are what "turn the CFG into an MCFG", so they must be included in order for the file to be valid, but here's the simple recipe for doing it correctly:

- For rules that introduce a terminal symbol (e.g. `dt --> "the"`), you do not need to add anything.
- For rules that introduce *one* nonterminal symbol (e.g. `n1 --> b`), add `[0,0]`.
- For rules that introduce *two* nonterminal symbols (e.g. `v1 --> v pp`), add `[0,0;1,0]`.
- For rules that introduce *three* nonterminal symbols, add `[0,0;1,0;2.0]`.

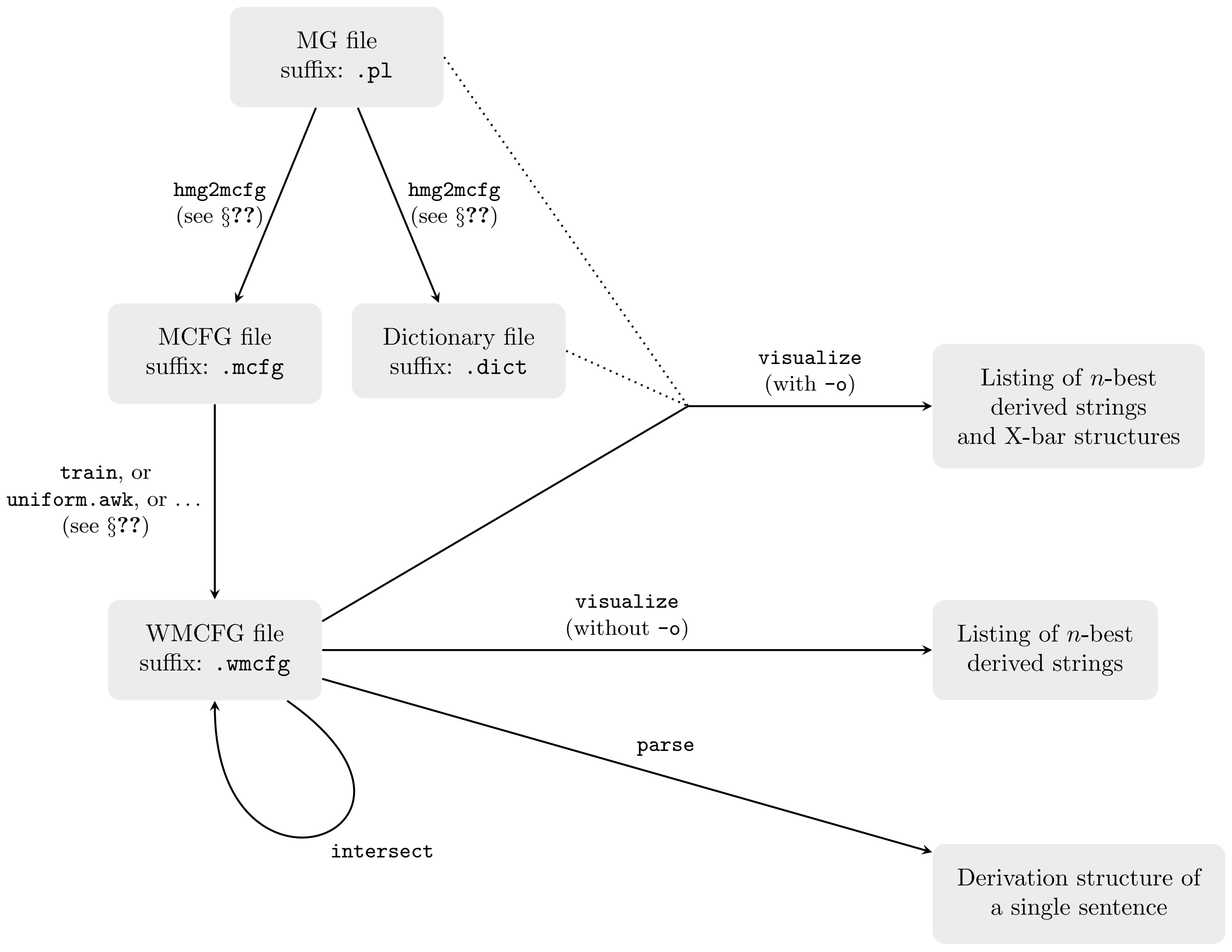


Figure 1: Overview of the dependencies among various kinds of inputs and outputs. Dotted lines indicate "implicit input" dependencies: `visualize` (with `-o`) relies on appropriate MG and dictionary files being present in the expected locations, but does not take these as command-line options.

- And so on …

Notice that this file format does not allow rules that have a combination of terminals and nonterminals on the right hand side. (Or, in general MCFG terms, string constants can only be introduced at terminating rules.)

Option B: Produce an MCFG from a Minimalist Grammar

An alternative is to use Guillaumin's translator to convert a Minimalist Grammar (MG) to an equivalent MCFG. See Guillaumin's report for details.

For the purposes of CCPC, we assume that all MG files are in "prolog format" (suffix `.pl`). See the included MGs `larsonian1.pl` and `chomskyan.pl` (in `grammars/mg`) for examples. Supposing that you have written a new MG, save this file in that same directory, say as `grammars/mg/newgrammar.pl`, and then run the following command.

```
$ /path/to/guillaumin/hmg2mcfg/hmg2mcfg -pl grammars/mg/newgrammar.pl -o ▶
   ▶ grammars/mcfgs/newgrammar.mcfg -dict grammars/mcfgs/newgrammar.dict
```

This will create two new files: the MCFG file itself, `grammars/mcfgs/newgrammar.mcfg`, and also a "dictionary file" `grammars/mcfgs/newgrammar.dict` that records the relationship between the nonterminals in this MCFG and the features in the original MG.[4]

It is important that the relative paths of these three files (`.pl`, `.mcfg` and `.dict`) are as indicated in these examples. You can use any directory you like in place of the `grammars` directory that comes with the CCPC bundle, but the structure inside this directory must be the same.

2.2 How to add weights to an unweighted MCFG file

The required format for `.wmcfg` files is based on that of `.mcfg` files, the only difference being that each rule is preceded by a weight (and then some whitespace): this weight is specified in fractional form, as two integers separated by a slash (e.g. `2 / 3`).

Two automated ways of adding such weights to an unweighted MCFG are provided, i.e. ways to generate `grammars/wmcfg/newgrammar.wmcfg` from an existing file `grammars/mcfgs/newgrammar.wmcfg`.

Option A: Locally uniform distibution

As a particularly quick-and-dirty method, `uniform.awk` adds weights to an MCFG in a way that uniformly distributes a total weight of one across all of the alternative ways of rewriting each nonterminal. For example, if there are three rules that have VP as their left-hand side, then each of these rules will be assigned a weight of $\frac{1}{3}$; if there are two rules that have NP as their left-hand side, then each will be assigned a weight of $\frac{1}{2}$; etc.

[4]Creating the MCFG file without the dictionary file (simply leave off the `-dict` option) will still generate a working MCFG file, but without the dictionary file it will not be possible for `visualize` to connect the MCFG to the MG in the way that it needs to for displaying the X-bar tree output.

```
$ awk -f uniform.awk grammars/mcfgs/newgrammar.mcfg > ▶
    ▶ grammars/wmcfg/newgrammar.wmcfg
```

Option B: Relative frequency estimation from a corpus

A more involved method is to compute weights by relative frequency estimation from a corpus.

The required corpus file has a very simple format: each line consists of a sentence preceded by a frequency (the "number of times it occurs"). Here is a small example:

```
$ cat grammars/train/sleep.train
20 mary sleeps
20 john sleeps
10 mary sleeps on tuesday
10 john sleeps on tuesday
2 mary sleeps on tuesday on tuesday
2 john sleeps on tuesday on tuesday
```

This can be used to add weights to an unweighted MCFG `sleep.mcfg` (itself derived from the MG `sleep.pl`, as described above) as follows:

```
$ ./train grammars/mcfgs/sleep.mcfg grammars/train/sleep.train > ▶
    ▶ grammars/wmcfg/sleep.wmcfg
```

This method has an unfortunate limitation. Notice that the training corpus takes the form of unparsed sentences, whereas relative frequency estimation requires counting uses of rules in parsed structures. The `train` tool simply parses the sentences in the corpus (using the specified unweighted grammar) in order to construct the relevant structures, but it therefore has no way to handle unambiguous strings: the corpus has no way to specify that one or another of the compatible derivations was intended. If ambiguous strings are encountered, `train` produces some warnings but also produces a not-very-sensible weighted grammar anyway. In the future `train` should accept as input a corpus of derivation trees, which would avoid this problem.

2.3 Using the Makefile to generate grammars

As a convenience, the CCPC's Makefile can be used to automate some of the steps described in the previous two subsections.

The `make` variable `GRAMMARS` defaults to the `grammars` directory, but other options can be specified on the command line. (See the `man` page for `make`.)

Here's an informal description of the relevant targets, using `%` as a wildcard as in Makefile syntax.

- `make $(GRAMMARS)/mcfgs/%.mcfg`
  Depends on: `$(GRAMMARS)/mg/%.pl`
  Uses Guillaumin's `hmg2mcfg` translator to produce an MCFG file.

- `make $(GRAMMARS)/mcfgs/%.dict`
  Depends on: `$(GRAMMARS)/mg/%.pl`
  Uses Guillaumin's `hmg2mcfg` translator to produce a dictionary file.
- `make $(GRAMMARS)/wmcfg/%.wmcfg`
  Depends on: `$(GRAMMARS)/mcfgs/%.mcfg`
  Uses `train` with corpus file `$(GRAMMARS)/train/%.train`; or, if this file does not exist, uses `uniform.awk`.[5]

[5] This is actually a simplification. The names of the MCFG file and the training corpus need not match precisely: any prefix of the string matching `%`, plus the `.train` suffix will work. So for example, one can have a single training corpus `english.train` that will be used by distinct grammars `english1.mcfg`, `english2.mcfg`, etc. If there are multiple training files that share a prefix with the MCFG file, then the one with the longest matching prefix is used. (If there are none, then the Makefile uses `uniform.awk`.)