Getting started with Constraint Grammar

Kevin Donnelly*

Abstract

Once you have got CG installed, as described in Chapter 3 of the manual, you will want to start using it. This note describes how to do this, using Welsh as the target language. Bear in mind that it only scratches the surface of what is a very elegant and versatile system, about which I myself have a great deal still to learn.

1 Preparing input text

The first step is to take each surface form in your text, and make a list of the possible lemmas (lexemes in CG terminology) it could derive from, along with relevant morphological tags. For instance, in Welsh, the surface form **mae** could derive from the verb **bod** (*be*), or it could derive from the noun **bae** (*bay*). Setting these facts out in the default CG format gives:

```
"<mae>"
    "bod" vfle 3s present :be:
    "bae" n nm m s :bay:
```

The format lists the surface form in angle brackets and quotes, followed by a newline (**n**). Then the "readings" (i.e. lemma + tags) are listed on separate lines - first a tab (**t**), then the lemma in quotes, and then any morphological tags you have assigned, and finally a newline (**n**). In the above case, **mae** could be either an inflected verb (the third person singular, present tense of **bod**), or a noun (a nasally-mutated form of the masculine singular noun **bae**). CG does not enjoin any specific morphological tags - you are free to choose whatever ones best suit your goals. In the above sample, I have chosen to include an English gloss for the lemma as one of the tags, surrounded by colons to set it apart from the other tags.

Each surface form in your text must be treated in the same way, so for the sentence:
Mae Brian yn gweithio yn ofnadwy o galed yn y swyddfa.
Brian is working terribly hard in the office.

*I am grateful to Tino Didriksen for comments on earlier drafts of this tutorial.

you should end up with something like the following (which is referred to as *cysample.txt* in the rest of the tutorial):

```
"<Mae>"
    "bod" vfle 3s present :be:
    "bae" n nm m s :bay:
"<Brian>"
    "Brian" unk
"<yn>"
    "yn" part stative
    "yn" p :in:
"<gweithio>"
    "gweithio" vinf :work:
    "gweithio" vfle 3s subjunctive :work:
"<yn>"
    "yn" part stative
    "yn" p :in:
"<ofnadwy>"
    "ofnadwy" a :terrible:
"<o>"
    "o" p :from:
    "o" p :of:
"<galed>"
    "caled" a sm :hard:
"<yn>"
    "yn" part _stative
    "yn" p :in:
"<y>"
    "y" part indrel
    "y" t :the:
"<swyddfa>"
    "swyddfa" n f s :office:
"<$.>"
```

Preparing your texts in this way can of course be done manually, but it is much easier to use an automated system of some kind to generate the possible lemmas and morphological tags for each surface form. In the above case, the output is created by having a PHP script read each word of the text, look it up in a dictionary database table that includes the various morphological tags, and write everything out in the CG format.

The tags I use here are self-explanatory: p=preposition, vinf=verb infinitive, a=adjective, f=feminine, t=definite article, c=conjunction, part=particle, indrel-=indirect relative, unk=unknown. The period or full-stop at the end of the sentence is given its own entry.

CG passes along untouched any text that is in non-CG format - anything that will not parse will be left as-is, so you can mix (for instance) plain text or HTML with CG text. We could have done this with the name **Brian** above, but this

means that you lose context, since you cannot query this surface form. So the script outputs unknown words in CG format, but with the default tag **unk**.

2 The first grammar rule

In the above text, you can see that there are several instances of ambiguous readings, which is what we hope CG will solve for us. Apart from **mae** already mentioned, there are ambiguities with **yn** (stative particle or preposition), **gweithio** (infinitive or subjunctive), **o** (two possible meanings), and **y** (definite article or indirect relative particle).

The next step is to write grammatical rules which CG will use to select one of these forms rather than the other, and provide us with at least partially-disambiguated text. In a text-editor, tell CG first of all what delimiters will be used for sentence boundaries. For this example text, we only need one – a period or full-stop:

```
delimiters = "<$.>";
```

Keywords such as DELIMITERS are usually capitalised, but I find the text easier to read if they are not.

The next part of the grammar is for convenience - we list sets of tags to delineate particular grammatical features, which makes it easier to zero in on specific groups of morphological features. In this example, most of our set definitions will do no more than expand the tags so that they are easier to read, as in the first 7 lines below.

```
list noun = n;
list inflected = vfle;
list infinitive = vinf;
list preposition = p;
list particle = part;
list adjective = a;
list conjunction = c;
list nmnoun = (n nm);
```

The last set definition, however, combines two tags to make a new set, that of nasally mutated nouns. Note that when we use two or more tags like this to create a set we must put them in parentheses, which are optional for single tags.

The format is: the keyword LIST, then the name of the set, then an equals sign, then the tags which will be included in the set, and finally a semicolon. For instance, we could declare a set of third-person singular, present tense verbs with aspirate mutation as follows:

```
list asp_3s_pres = (am 3s present);
```

Sets can also be manipulated by using the keyword SET. So if we wish to define a new set of feminine nouns:, we can write the following declaration:

```
set fem_noun = noun + (f);
```

adding the tag **f** to the previously-defined set **noun**. By enclosing the tag in parentheses we create an on-the-fly inline set, which the **+** function can work with. Alternatively, we could define a new set for feminine items:

```
list feminine = f;
```

and then combine that with the noun set:

```
set fem_noun = noun + feminine;
```

The next part of the grammar is another keyword, SECTION. Basically, this says that we are now starting the actual rules. You can have multiple rule sections, each of which can be given an optional name, and they can be run sequentially, or in isolation, or in repeated groups. For this example, we have only one section.

Now come the actual grammar rules which will disambiguate examples such as **mae**. We can say (with some slight simplification for this example) that nasally-mutated nouns will only ever occur after a few specific words like the preposition **yn** (in), or the possessive **fy** (my). So we can write a rule that says, for this instance: 'remove from consideration any reading relating to **bae** (*bay*) if the preceding word is not a form of the lemmas **yn** or **fy**:

```
remove ("bae") if (not -1 ("yn" "fy"));
```

The rule uses the keyword REMOVE, and then specifies what should be removed, and under what conditions. The keyword IF is optional, but improves readability. The condition is placed in parentheses, and can be negated (**not**), or use numbers to refer to position: **–n** means *n places to the left*, **n** means *n places to the right*. Note that lemmas must be quoted and placed in parentheses – this is because rules only take sets as targets, and (as noted above) parentheses must be used to create *ad hoc* sets. Note also that, like the set definitions, the rule must end with a semicolon.

We can use any tag attached to the word we want to remove; any of the following will also work:

```
remove (:bay:) if (not -1 ("yn" "fy"));
remove (n) if (not -1 ("yn" "fy"));
remove (nm) if (not -1 ("yn" "fy"));
```

However, although these will work in this context, they will also apply in other contexts, which may not be what we want – the second in particular would be excessive, since it would remove all nouns unless they were preceded by **yn** or **fy**! It may also be useful to generalise the rule that a nasally-mutated noun should only be expected after **yn** and **fy**. So we will rewrite the rule to apply to such nouns specifically, using the set we defined earlier:

```
remove nmnoun if (not -1 ("yn" "fy"));
```

3 Applying the grammar

We can now test whether the grammar works. Save the grammar file as *smallcygrm*, and in a terminal run:

```
./cg3-autobin.pl -g smallcygrm -I cysample.txt
```

where *cysample.txt* is the formatted text we looked at earlier. I have saved both the grammar and sample files in the same directory as the **vislcg3** executables, where I am running this command, but obviously you can choose another location. The above command uses **cg3-autobin.pl** instead of **vislcg3** itself – this Perl program is a wrapper that takes the same arguments and will compile the grammar to binary format if it has changed since last run. This enables the ease of development of text grammars to be combined with the speed of binary grammars for testing and use. The switch **-g** specifies the grammar file to use, and the switch **-I** (capital i) specifies the speech file you wish to disambiguate.

The output is encouraging:

```
"<Mae>"
    "bod" vfle 3s present :be:
Brian
"<yn>"
    "yn" part stative
    "yn" p :in:
"<gweithio>"
    "gweithio" vinf :work:
    "gweithio" vfle 3s subjunctive :work:
"<yn>"
    "yn" part stative
    "yn" p :in:
"<ofnadwy>"
    "ofnadwy" a :terrible:
"<o>"
    "o" p :from:
    "o" p :of:
"<galed>"
```

```
    "caled" a sm :hard:
"<yn>"
    "yn" part stative
    "yn" p :in:
"<y>"
    "y" part indrel
    "y" t :the:
"<swyddfa>"
    "swyddfa" n f s :office:
"<$.>"
```

Mae has been correctly disambiguated to show derivation from the verb **bod** only.

An alternative way of running the grammar is:

```
cat cysample.txt | ./cg3-autobin.pl -g smallcygrm
```

Or you can pass the entire text to the program as one string, using **n** to represent newlines and **t** to represent tabs:

```
echo -e '"<Mae>"\n \t"bod" vfle 3s present :be:\n\t"bae" n nm m sg
\nBrian\n"<yn>"\n\t"yn" part stative\n\t"yn" p :in:\n"<gweithio>"
\n\t"gweithio" vinf :work:\n\t"gweithio" vfle 3s subjunctive
:work:\n"<yn>"\n\t"yn" part stative\n\t"yn" p :in:\n"<ofnadwy>"
\n\t"ofnadwy" a :terrible:\n"<o>"\n\t"o" p :from:\n\t"o" p :of:
\n"<galed>"\n\t"caled" a sm :hard:\n"<yn>"\n\t"yn" part stative
\n\t"yn" p :in:\n"<y>"\n\t"y" part indrel\n\t"y" t :the:
\n"<swyddfa>"\n\t"swyddfa" n f s :office:\n"<\$.>"\n' |
./cg3-autobin.pl -g cygrammar/smallcygrm
```

Note that there should be no **t** before the surface form, and that the **n** and **t** should not be separated from (respectively) the surface form and the lemma.

4 Completing the rules

We can now write some more rules to deal with the other surface forms that need to be disambiguated. Looking at **gweithio** first, the infinitive reading should be chosen, since it occurs after **yn** – which in this case is a stative marker, and not the homonymous preposition **yn** (*in*). This fact can be reflected in this rule:

```
select infinitive if (-1 ("yn" part));
```

Note that a directly-quoted lemma must be in quotes, and both it and any of its related tags must be in parentheses, so that they make an inline set, as noted earlier. We are here using another keyword, SELECT, which specifies which

reading should be preferred, unlike REMOVE, which specifies which reading to discard.

We can use the same information to disambiguate **yn** – where it occurs before an infinitive (like **gweithio**) or an adjective (like **caled**), it is a stative. We can therefore write another rule:

```
select ("yn" part) if ((1 infinitive) or (1 adjective));
```

If we save *smallcygrm* and run the grammar again, the output now looks much better:

```
"<mae>"
    "bod" vfle 3s present :be:
Brian
"<yn>"
    "yn" part stative
"<gweithio>"
    "gweithio" vinf :work:
"<yn>"
    "yn" part stative
"<ofnadwy>"
    "ofnadwy" a :terrible:
"<o>"
    "o" p :from:
    "o" p :of:
"<galed>"
    "caled" a sm :hard:
"<yn>"
    "yn" part stative
    "yn" p :in:
"<y>"
    "y" part indrel
    "y" t :the:
"<swyddfa>"
    "swyddfa" n f s :office:
"<$.>"
```

Four of the seven original ambiguous surface forms have now been resolved.

The surface form **yn** is still ambiguous in one instance, where it appears before the definite article **y** (*the*). In this location it will never be a stative marker, so let's reflect that in another rule:

```
select ("yn" p) if (1 (t));
```

I am here using the preposition tag in the rule, but you could use any tag; for instance,

```
select ("yn" :in:) if (1 (t));
```

will work just as well. For consistency, though, it is probably best to use meaning tags only for cases where senses need to be distinguished (see **o** below).

Let's deal with **y** too. It will only ever be the indirect relative particle when it precedes an inflected verb, so this rule encapsulates that:

```
select ("y" t) if (not 1 inflected);
```

Note again the use of **1** to indicate "next word to the right" – the condition here therefore reads "if the next word to the right is not an inflected verb".

Only one item remains to be dealt with – the alternative senses of the preposition **o** (*of, from*). Preceding an adjective, the sense is much more likely to be *of*, though that condition does not rule out *from* entirely. So our initial rule here might be:

```
select ("o" :of:) if (1 adjective);
```

This will work, but leaves something to be desired – it is too broad, and may apply when the real sense is *from*. It is in fact better to make the rule narrower, so that it applies only to this context – if it applies more widely, it may create difficulties later which will cost time and effort to debug. So we will rewrite the rule to make it apply only in those cases where we have a prequalifier – **ofnadwy** (*terribly*), **andros** (*really*), etc.

First, we add a new set definition (using quotes because we are referring to lemmas and not tags):

```
list prequal = "ofnadwy" "andros";
```

We can, of course, add more examples as we come across them. Note that since we are referring to lemmas, we need to surround them with quotes.

We can then rewrite the rule to refer to this new set, saying that the *of* sense should be chosen when **o** is preceded by a prequalifier and followed by an adjective:

```
select ("o" :of:) if (-1 prequal)(1 adjective);
```

If we run the grammar again, the output is perfect:

```
"<Mae>"
    "bod" vfle 3s present :be:
Brian
"<yn>"
    "yn" part stative
```

```
"<gweithio>"
    "gweithio" vinf :work:
"<yn>"
    "yn" part stative
"<ofnadwy>"
    "ofnadwy" a :terrible:
"<o>"
    "o" p :of:
"<galed>"
    "caled" a sm :hard:
"<yn>"
    "yn" p :in:
"<y>"
    "y" t :the:
"<swyddfa>"
    "swyddfa" n f s :office:
"<$.>"
```

The final grammar looks like this:

```
DELIMITERS = "<$.>";

LIST noun = n;
LIST inflected = vfle;
LIST infinitive = vinf;
LIST preposition = p;
LIST particle = part;
LIST adjective = a;
LIST conjunction = c;
LIST nmnoun = (n nm);
LIST prequal = "ofnadwy" "andros";

SECTION

remove (nm) if (not -1 ("yn" "fy"));
select infinitive if (-1 ("yn" part));
select ("yn" part) if ((1 infinitive) or (1 adjective));
select ("yn" p) if (1 (t));
select ("y" t) if (not 1 inflected);
select ("o" :of:) if (-1 prequal)(1 adjective);
```

5 Tracing which rules were applied

It can be useful to see what rules were applied to a particular piece of text. To enable this, use the **–trace** switch:

```
./cg3-autobin.pl --trace -g smallcygrm -I cysample.txt
```

This gives the following output:

```
"<Mae>"
        "bod" vfle 3s present :be:
;       "bae" n nm m s :bay: REMOVE:17
Brian
"<yn>"
        "yn" part stative SELECT:25
;       "yn" p :in: SELECT:25
"<gweithio>"
        "gweithio" vinf :work: SELECT:21
;       "gweithio" vfle 3s subjunctive :work: SELECT:21
"<yn>"
        "yn" part stative SELECT:25
;       "yn" p :in: SELECT:25
"<ofnadwy>"
        "ofnadwy" a :terrible:
"<o>"
        "o" p :of: SELECT:31
;       "o" p :from: SELECT:31
"<galed>"
        "caled" a sm :hard:
"<yn>"
        "yn" p :in: SELECT:27
;       "yn" part stative SELECT:27
"<y>"
        "y" t :the: SELECT:29
;       "y" part indrel SELECT:29
"<swyddfa>"
        "swyddfa" n f s :office:
"<$.>"
```

Each reading line shows the line-number of the grammar rule applied, and a semicolon is placed at the beginning of readings that were struck out. This can be very useful when trying to debug your grammar, and see which rules are firing, and when.

To avoid having to refer constantly to the grammar file, you can name the rules by adding a colon and then a chosen name after the rule's keyword. For instance, we can rewrite the **mae** rule to read:

```
remove:DeleteNmNoun nmnoun if (not -1 ("yn" "fy"));
```

If we add names to all the rules, and then use another switch to see only surviving readings after the rules have been applied:

```
./cg3-autobin.pl --trace-no-removed -g smallcygrm -I cysample.txt
```

we get the following output:

```
"<Mae>"
        "bod" vfle 3s present :be:
Brian
"<yn>"
        "yn" part stative SELECT:25:ChooseStativeYn
"<gweithio>"
        "gweithio" vinf :work: SELECT:21:ChooseInfin
"<yn>"
        "yn" part stative SELECT:25:ChooseStativeYn
"<ofnadwy>"
        "ofnadwy" a :terrible:
"<o>"
        "o" p :of: SELECT:31:ChooseO_Of
"<galed>"
        "caled" a sm :hard:
"<yn>"
        "yn" p :in: SELECT:27:ChoosePrepYn
"<y>"
        "y" t :the: SELECT:29:ChooseArtY
"<swyddfa>"
        "swyddfa" n f s :office:
"<$.>"
```

Note that the **–trace-no-removed** switch, although providing output that is easier to read, has the drawback that REMOVE rules are not shown, because they are attached to the reading that is removed.

6 A note on rule coverage

The rules in the CG grammar are strictly applied in the order they occur in in the grammar file, but they are re-run multiple times. So if a rule was unable to do anything first time around, it may be able to do something on subsequent iterations due to later rules having cleared the way during previous iteration. Sections are re-run until no rule fires, then the next section is added to the pool and run until nothing fires; this is repeated until all the sections or ambiguities are exhausted.

You may see instances where the output differs depending on the position of a specific rule. In my (limited!) experience, this behaviour is almost always due to the fact that the rule is not defined tightly enough. If you rewrite it more strictly, it should have the desired effect no matter where it comes in the grammar file. If you then find that several rules are covering the same ground, you can combine them into one – in other words, it is probably better to go bottom-up than top-down.

7 A note on parentheses

Parentheses are widely used in CG, but they have different meanings in different contexts:

- With the keyword LIST, they create composite tags:
  `list nmnoun = (n nm);`
  will create a set of nasally-mutated nouns.
- With the keyword SET, they create on-the-fly inline sets:
  `set fem_noun = noun + (f);`
  where the **f** tag is converted to a set that can then be combined with the *noun* set.
- In the target section of a rule (the bit you want the rule to act on), parentheses again create a set:
  `select ("yn" p) if (1 (t));`
  where the target is a set consisting of readings with surface form **yn** and tag **p**.
- In the condition section of a rule (the bit after the **if**), they create an on-the-fly template:
  `select ("yn" part) if ((1 infinitive) or (1 adjective));`
  See Chapter 14 of the manual for more information on this.