— Lecture Notes —

Eliminating Left Recursion from CFGs

1 Left Recursion Writing grammars using Prolog's DCG formalism is very convenient for testing purposes, in particular when we add arguments to our category symbols to keep track of the parse trees and so forth. But we immediately run into the problem of left recursion. Many grammatical categories are left recursive. For example, in English we might make use of the following two rules.

```
s( s/[S1,and,S2] ) --> s( S1 ), [and], s( S2 ).
np( np/[NP,s,N] ) --> np( NP ), [s], n( N ).
```

(Here we are using `s` in place of the harder-to-type `'\'s'` for the possessive marker.)

The left recursion doesn't go away if we decide that possessive NPs are just a subspecies of Det and write instead

```
np( np/[D,N] ) --> det( D ), n( N ).
det( det/[NP,s] ) --> np( NP ), [s].
```

The NP rule is not directly left recursive, but the problem is the same. We just postpone the inevitable.

One solution that immediately suggests itself is simply to transform the grammar into a weakly equivalent grammar which is not left recursive, for example, into Greibach Normal Form (GNF). In GNF, the right hand side of every rule begins with a terminal, so necessarily there is no left recursion.

Such a grammar is unfortunately only weakly equivalent to the grammar we started out with, and we care very much about the derivation trees assigned by a grammar, not just the strings it recognizes. However, there is a way we can "have our cake and eat it too," that is, transform the grammar but still get the parse trees we need. But it's a little tricky! So read carefully.

2 Not Quite Greibach Normal Form Transforming a grammar into GNF actually involves two separate transformations of the grammar rules. One of the transformations deals with the problem of removing directly left recursive rules, like our two original examples. Removing indirect left recursion, as in the revised NP rule, is a little more complicated. Here we only deal with the direct case.

We do not need to transform the grammar all the way into GNF. Our right hand sides do not need to begin with a lexical item. We just need to know that for all rules $A \longrightarrow B\alpha$, there is no derivation of the form $B \stackrel{*}{\Longrightarrow} A\beta$.

3 The Transformation Without Trees (This part follows very closely the presentation in Hopcroft and Ullman (1979), pp. 94ff.) Define an *A-production* to be a production with nonterminal A on the left. Divide the set of A-productions into

$$A \longrightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_r$$

i.e., the (directly) left recursive productions for A, and

$$A \longrightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_s$$

i.e., those that are not (directly) left recursive.

Now add a fresh nonterminal to the grammar. Call it B. We remove all A-productions from the grammar and replace them with

$$\begin{array}{ll} A \longrightarrow \beta_i & B \longrightarrow \alpha_i \\ A \longrightarrow \beta_i B & B \longrightarrow \alpha_i B \end{array}$$

This has the effect of generating the same strings that our original grammar generated with a left-branching derivation with a new right branching derivation.

In terms of recursive programming, the α-rules are the recursive clauses in the definition of A and the β-rules are the base cases. Any use of A in a derivation must produce a sequence of α_i's terminating with a single β_j on the left:

$$A \Longrightarrow A\alpha_{i_1} \Longrightarrow A\alpha_{i_2}\alpha_{i_1} \Longrightarrow \ldots \Longrightarrow A\alpha_{i_p}\ldots\alpha_{i_1} \Longrightarrow \beta_j\alpha_{i_p}\ldots\alpha_{i_1}$$

The new grammar will assign such a sentential form a right branching derivation tree. Compare the two trees in Fig. 1.

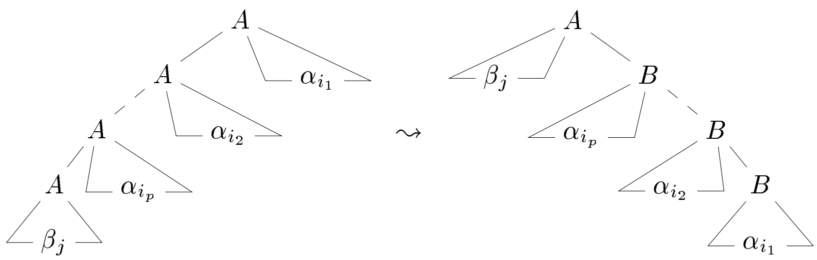

Figure 1: A left recursive derivation tree and the transformed tree.

4 Our Example Lets fill out our example grammar just a little.

```
s --> np, vp.          np --> det, n.
s --> s, [and], s.     np --> np, [s], n.
```

Take the S-rules first. Then

$$A = S,\ \alpha_1 = \textit{and } S,\ \beta_1 = \textit{NP VP}$$

Let us call our new category b_s. Then we replace the two S-rules with the following system.

```
s --> np, vp.         b_s --> [and], s.
s --> np, vp, b_s.    b_s --> [and], s, b_s.
```

We can add another new category b_np and tackle the NP-rules similarly.

```
np --> det, n.          b_np --> [s], n.
np --> det, n, b_np.    b_np --> [s], n, b_np.
```

5 Now for the trees Our new grammar is no longer left-recursive, but it now assigns the wrong trees.

Our approach for reconstructing the intended trees from the altered derivation crucially relies on the use of *difference structures*. Formally, a difference structure is a pair consisting of a term and a variable, where it is assumed that the variable also appears somewhere in the term. We will refer to the variable as having been *extracted* from the term. Obviously, given a term T with extracted variable X, binding X to some other term T2 will have the effect of inserting T2 into T at the position formerly occupied by X. In effect, X acts as a *pointer* to a position (possibly more than one position, in fact) in a given term.

Referring to Fig. 1, consider first what we should do when we are parsing the final B constituent. We collect a sequence of trees referred to in the figure as α_{i_1}. In the final tree, these will be subtrees of the root, with some other tree at the left corner. Also, the root will be labeled A, not B. Take the case of an NP like *the boy's father's father's father.* α_{i_1} in this case is the pair of trees s, n(father). So we know that the tree that should be constructed by the application of the rule

```
b_np --> [s], n.
```

should be of the form np(X,s,n(father)). But we don't know what to do with the X, nor can we know until we are higher up in the tree. So we return the difference structure consisting of that term plus the variable X, as follows.

```
b_np( np(X,s,N), X ) --> [s], n( N ).
```

6 Now consider what we should do with the final application of the recursive b_np rule.

```
b_np --> [s], n, b_np.
```

The call to b_np will return the difference structure we just discussed. If you refer to Fig. 1, and note that we are now working on $\alpha_{i_2}B$, you will see that the stuff in α_{i_2} belongs in the left corner of the tree we just created. That is, it belongs in the position currently occupied by X. But it has a hole in its own left corner, which we need to keep track of. So the result of applying this rule will also be a difference structure. We derive the following.

```
b_np( T, X ) --> [s], n(N), b_np(T,X0), {X0=np(X,s,N)}.
```

We can compile in the explicit unification and get a more compact, but more confusing-looking rule, as follows.

```
b_np( T, X ) --> [s], n( N ), b_np( T, np(X,s,N) ).
```

Notice that the structure returned by the recursive call to `b_np` is passed up to the calling goal unchanged: it remains `T`. All the work is going on on its left corner. Unlike a normal parse tree, it is getting deeper as we go up the derivation tree, not higher.

7 Now, the `np` rule itself should only return a single tree. That is, at this point we have to close off the difference structure. The full `np` complex looks like the following.

```
np( np(D,N) ) --> det( D ), n( N ).
np( T ) --> det( D ), n( N ), b_np( T, np(D,N) ).

b_np( np(X,s,N), X ) --> [s], n( N ).
b_np( T, X ) --> [s], n( N ), b_np( T, np(X,s,N) ).
```

Referring to Fig. 1, observe that the call to `np` is responsible for parsing β_j, which is the actual bottom left corner of the true, intended parse tree. In this case β is the Det N sequence to the left of the `b_np`. We bundle that up into an `np` constituent-term, and plug it into `X`. It contains no variables, so there is nothing left to keep track of. Our difference structure has become an ordinary structure.

8 The same tricks can be applied to the S-rules. The result is as follows.

```
s( s(NP,VP) ) --> np( NP ), vp( VP ).
s( T ) --> np( NP ), vp( VP ), b_s( T, s(NP,VP) ).

b_s( s(X,and,S2), X ) --> [and], s( S2 ).
b_s( T, X ) --> [and], s( S2 ), b_s( T, s(X,and,S2) ).
```

Observe the similarities between these two sets of rules.

9 One can also think of what we are doing in a more top-down fashion. Consider the goal

```
?- s( T, [john,sneezed,and,john,sneezed], [] ).
```

We get the following.

```
?- np(NP_1,[john|...],L1_1), vp(VP_1,L1,L2), b_s(T,s(NP_1,VP_1),L2,[]).
```

Assume that the calls to `np` and `vp` work as expected. Then we are left with the goal

```
?- b_s(T,s(np(john),vp(sneezed))).
```

This expands to

```
?- 'C'(and,[and|...],L1), s(S2,L1,L2),
   b_s(T,s(s(np(john),vp(sneezed)),and,S2),L2,[]).
```

From this perspective what we are doing is to take the original "kernel" S, parsed as the term `s(np(john),vp(sneezed))`, and then, as we work our way down the recursion, we keep wrapping around it new terms of the form `s(X,and,s(...))`. Only when we reach the base of the recursion and use the nonrecursive clause for `b_s` will we actually say what `T` is. In effect, we wrap a parse of "and John sneezed" around what we have in the second argument and this time we put it in the first or primary argument position instead of keeping it in "storage" in the second argument position.

10 Left as an Exercise to the Reader Now consider the rules

```
np --> det, n.      det --> np, [s].
```

How do we handle non-local left recursion? The answer comes again from the techniques for converting to Greibach Normal Form. The basic idea is to *unfold* the call to `det`. Doing so gives you the rule we have seen already:

```
np --> np, [s], n.
```

However, if we started with a grammar with the two rules above, it is likely that we really meant to analyze such NPs as having the structure `np(det(NP,s),N)`, for some `NP` and `N`. So how can we take our unfolded rule and get it to assign the intended tree? That is, take

```
np( T ) --> np( NP2 ), [s], n( N ), {T = ?}.
```

and replace the question mark with a legal Prolog term, presumably made out of `NP2`, `N`, and whatever else you think it needs.