— Lecture Notes —

From Top-Down Recognizers to DCGs

```
rule( s, [np,vp] ).
rule( np, [det,n] ).
rule( np, [pn] ).
rule( vp, [v] ).
rule( vp, [v,np] ).
rule( vp, [v,s] ).

word( the, det ).
word( villain, n ).
word( hero, n ).
word( duck, n ).
word( pursue, v ).
word( saw, v ).
word( duck, v ).
word( her, det ).
word( i, pn ).
word( her, pn ).
```

Figure 1: File `gsimple.pl`.

1 The following programs are all based on the grammar in Fig. 1.

2 First we have the basic top-down grammar interpreter, presented in Fig. 2. It is a general device, in which the predicates `rule/2` and `word/2` are hook predicates which the user defines separately.

3 We intend to transform this interpreter, together with the grammar in Fig. 1, into a Prolog definite clause grammar.

Our basic strategy will be to get rid of all the predicates used in the grammar and the interpreter, eventually replacing them with the nonterminals of the grammar.

4 `td1.pl` This version gets rid of the call to `multipush/3` by calling a variant of the top down interpreter on the right hand side of the rule before continuing with the rest of the stack.

The call to `td_aux` may alter the string, so it needs to return whatever the altered string is, in its 3rd argument.

Of course, now `td_aux/3` does exactly the same work as `td/2` does. The only difference is that it consumes only part of the string, and returns the remainder.

```
rec( String ):-
        td( [s], String ).

%%% accept
td( [], [] ).
%%% scan
td( [PreTermCat|RestStack], [Word|RestString] ):-
        word( Word, PreTermCat ),
        td( RestStack, RestString ).
%%% expand
td( [NonTermCat|RestStack], String ):-
        rule( NonTermCat, RightHandSide ),
        multipush( RightHandSide, RestStack, NewStack ),
        td( NewStack, String ).

multipush( RightHandSide, RestStack, NewStack ):-
        append( RightHandSide, RestStack, NewStack ).
```

Figure 2: File `td0.pl`: A simple top down grammar interpreter.

```
%%% expand
td( [NonTermCat|RestStack], String0 ):-
        rule( NonTermCat, RightHandSide ),
        td_aux( RightHandSide, String0, String1 ).
        td( NewStack, String1 ).

td_aux( [], String, String ).
td_aux( [Cat|Cats], [W|String1], String ):-
        word( W, Cat ),
        td_aux( Cats, String1, String ).
td_aux( [Cat|Cats], String0, String ):-
        rule( Cat, RHS ),
        td_aux( RHS, String0, String1 ),
        td_aux( Cats, String1, String ).
```

Figure 3: File `td1.pl`: Getting rid of `multipush`.

td/2 could do that too, in which case we need only one of these routines, and a slightly modified rec/1. Note that in fact the td/2 itself will never be used, except once at the very beginning and once at the very end. All of the work is actually going to get done by td_aux/3.

```
rec( String ):-
        td( [s], String, [] ).

%%% accept
td( [], String, String ).
%%% scan
td( [PreTermCat|RestStack], [Word|S1], S ):-
        word( Word, PreTermCat ),
        td( RestStack, S1, S ).
%%% expand
td( [NonTermCat|RestStack], S0, S ):-
        rule( NonTermCat, RightHandSide ),
        td( RightHandSide, S0, S1 ),
        td( RestStack, S1, S ).
```

Figure 4: File td2.pl: Combining td_aux and td.

5 Observe that we now have the sentence being treated as a difference list, so that part of the DCG framework is now in place.

6 The crucial step comes now.

Observe that both rule/2 and word/2 are defined by unit clauses. So calls to rule/2 and word/2 don't introduce any deep branches in the search tree. As a result we can *unfold* the calls to these predicates, replacing the calls the the predicates by their definitions. In this case, the definitions of these predicates are all implemented by a few unifications, so we do the unifications directly in the clauses for td. Of course, we have to unfold the calls to rule/2 and word/2 in every possible way, so we now have a lot more clauses for td.

The result of unfolding (also called partial evaluation or partial execution) is illustrated in Fig. 5.

7 We just unfolded away our hook predicates, so note that our program is no longer a general purpose top down grammar interpreter: one particular choice of grammar is now part of the program. We cannot modify the grammar without modifying the program.

8 Now we know exactly how many times we will call the first td/3 subgoal, since its first argument is a constant. If we had a version of td/3 that had an atomic

```
rec( String ):-
        td( s, String, [] ).

%%% accept
td( [], String, String ).
%%% scan
td( [det|RestStack], [the|S1], S ):-
        td( RestStack, S1, S ).
td( [n|RestStack], [villain|S1], S ):-
        td( RestStack, S1, S ).
td( [n|RestStack], [hero|S1], S ):-
        td( RestStack, S1, S ).
td( [n|RestStack], [duck|S1], S ):-
        td( RestStack, S1, S ).
td( [v|RestStack], [pursue|S1], S ):-
        td( RestStack, S1, S ).
```

etc.

```
%%% expand
td( [s|RestStack], S0, S ):-
        td( [np,vp], S0, S1 ),
        td( RestStack, S1, S ).
td( [np|RestStack], S0, S ):-
        td( [det,n], S0, S1 ),
        td( RestStack, S1, S ).
td( [np|RestStack], S0, S ):-
        td( [pn], S0, S1 ),
        td( RestStack, S1, S ).
```

etc.

Figure 5: Unfolding the calls to the grammar predicates.

first argument instead of a list, then we could use it here. So we develop such a predicate. See Fig. 6.

```
%%% expand
td( [s|RestStack], S0, S ):-
        td_aux( np, S0, S1 ),
        td_aux( vp, S1, S2 ),
        td( RestStack, S2, S ).
td( [np|RestStack], S0, S ):-
        td_aux( det, S0, S1 ),
        td_aux( n, S1, S2 ),
        td( RestStack, S2, S ).
td( [np|RestStack], S0, S ):-
        td_aux( pn, S0, S1 ),
        td( RestStack, S1, S ).
```

etc.

```
td_aux( s, S0, S ):-
        td_aux( np, S0, S1 ),
        td_aux( vp, S1, S ).
td_aux( np, S0, S ):-
        td_aux( det, S0, S1 ),
        td_aux( n, S1, S ).
td_aux( np, S0, S ):-
        td_aux( pn, S0, S ).
```

etc.

```
td_aux( det, [the|S], S ).
td_aux( n, [villain|S], S ).
td_aux( n, [hero|S], S ).
td_aux( n, [duck|S], S ).
td_aux( v, [pursue|S], S ).
```

Figure 6: Unrolling the list literals.

9 Once again our "auxiliary" procedure winds up carrying all of the computational burden, and our main procedure only gets called to start things off and finish them up. So we eliminate the old `td/3` and replace it with `td_aux/3`, now renamed. See Fig. 7.

10 In effect, what we have just done is to "compile away" the stack that used to hold our nonterminal categories. We now use the Prolog stack instead.

```
rec( String ):-
        td( s, String, [] ).


td( s, S0, S ):-
        td( np, S0, S1 ),
        td( vp, S1, S ).
td( np, S0, S ):-
        td( det, S0, S1 ),
        td( n, S1, S ).
td( np, S0, S ):-
        td( pn, S0, S ).
td( vp, S0, S ):-
        td( v, S0, S ).
td( vp, S0, S ):-
        td( v, S0, S1 ),
        td( np, S1, S ).
td( vp, S0, S ):-
        td( v, S0, S1 ),
        td( s, S1, S ).

td( det, [the|S], S ).
td( n, [villain|S], S ).
td( n, [hero|S], S ).
td( n, [duck|S], S ).
td( v, [pursue|S], S ).
td( v, [saw|S], S ).
td( v, [duck|S], S ).
td( det, [her|S], S ).
td( pn, [i|S], S ).
td( pn, [her|S], S ).
```

Figure 7: Compiling in the stack.

11 The last step is just to take the constants in argument 1 of `td/3`, which are our nonterminal category symbols, and turn them into Prolog predicate names. The final result is presented in Fig. 8. You can clearly see that this is a DCG,w ith one inessential difference in the way we handle terminal symbols. The Prolog DCG compiler would translate a rule like

```
det --> [the].
```

as

```
det( S0, S ):- 'C'( S0, the, S ).
```

But `'C'/3` is defined so that the above `'C'/3` goal will succeed just in case `S0` equals `[the|S]`, which is also true of our clause

```
det( [the|S], S ).
```

Things are not so straightforward in full-scale DCGs, which may contain calls to Prolog's cut predicate, for example. Nonetheless it should be clear by now how the rather general presentation of top-down recognition in Fig. 2, which is really only "accidentally" written in Prolog as opposed to some other language, relates to a Definite Clause Grammar, which really is a formalism specific to logic programming.

12 Suppose we had started with a parser instead of a recognizer. In that case, our nonterminal category symbols would no longer be atoms, since each one would be associated with a parse tree. That only makes a difference in the last step, since predicate symbols must be atomic. All we have to do is to leave the parse trees as arguments of the DCG clauses, as in the following example.

```
td( STree:s, S0, S ):-
        td( NPTree:np, S0, S1 ),
        td( VPTree:vp, S1, S ).
```

becomes

```
s( STree, S0, S ):-
        np( NPTree, S0, S1 ),
        vp( VPTree, S1, S ).
```

```
rec( String ):-
        s( String, [] ).


s(S0, S ):-
        np( S0, S1 ),
        vp( S1, S ).
np( S0, S ):-
        det( S0, S1 ),
        n( S1, S ).
np( S0, S ):-
        pn( S0, S ).
vp( S0, S ):-
        v( S0, S ).
vp( S0, S ):-
        v( S0, S1 ),
        np( S1, S ).
vp( S0, S ):-
        v( S0, S1 ),
        s( S1, S ).

det( [the|S], S ).
n( [villain|S], S ).
n( [hero|S], S ).
n( [duck|S], S ).
v( [pursue|S], S ).
v( [saw|S], S ).
v( [duck|S], S ).
det( [her|S], S ).
pn( [i|S], S ).
pn( [her|S], S ).
```

Figure 8: Converting to a DCG.