Übung 6

1 You are to implement (or, to be precise, to complete the implementation) of an *Arc Eager Generalized Left Corner Recognizer*. That sounds like a lot, but it's not nearly as bad as it sounds, if you pay attention to the hints that follow.

2 The rules for the arc eager GLC recognizer are given in Fig. 1.

$$\frac{(aw \mid \Gamma)}{(w \mid a\Gamma)}\,\textit{shift} \qquad\qquad \frac{A \longrightarrow \alpha^r \,_\vartriangle\, \beta \quad (w \mid \alpha\Gamma)}{(w \mid \overline{\beta} A\Gamma)}\,\textit{glc}$$

$$\frac{(aw \mid \overline{a}\Gamma)}{(w \mid \Gamma)}\,\textit{shift+attach} \qquad\qquad \frac{A \longrightarrow \alpha^r \,_\vartriangle\, \beta \quad (w \mid \alpha\overline{A}\Gamma)}{(w \mid \overline{\beta}\Gamma)}\,\textit{glc+attach}$$

Figure 1: Arc Eager GLC Inference Rules.

3 Predicted and Completed Categories I recommend using the prefix operators '`-`' and '`+`' to mark predicted and completed categories (respectively). This way you can tell by unification whether a given stack element is predicted or completed. If you just use '`-`', to mark predicted categories, then you need a negative test ('`\+C=-(_)`') to tell if something is a completed category.

4 You will need to define a relation `glc_proof` that holds (minimally) between strings and stacks (both implemented as Prolog lists). You will find it helpful to add a third argument, to use as a code for the grammar rule marking strategy you will use (e.g., `lc` for the left corner strategy). That way you can define F_{td}, F_{lc} and F_{bu} (and any others you want to try) in the same program file, as a single predicate `f`, whose first argument is a strategy code (e.g., `td`, `lc`, etc.). (Using a predicate `f(StratCode, A, Alpha, Beta)` also means that you can use a single grammar for all your different strategies: marker placement can be computed on-line.)

5 Hint GLC proofs are *linear*: the state of the machine is always one of the assumptions, for every rule. And the other assumption, if there is one, is always an axiom (a grammar rule).

This means you will be able to implement your GLC prover as a recursion on the current state of the machine. The initial state, as noted in class, is $(w \mid \overline{S})$, where w is the input string. The final state (the base of the recursion) is the empty state $(\epsilon \mid \epsilon)$. Each step of the recursion applies an inference rule to the current state to derive a new state.

6 Very Important Hint The *glc* rule can be easily abused. In particular, in strategies like F_{td}, when α is empty, *glc* can always apply, making the search space infinite even when the grammar generates a finite number of sentences (so, *a fortiori*, is not left recursive).

On the other hand, in the top down strategy, *glc* should never be used at all. So at least for the purposes of today's Übung it is safe to place a guard on your implementation of the *glc* rule that requires α to be nonempty.

Note: I do not know if it is safe to always have this guard on the *glc* rule. There may be strategies that are slightly less top down than F_{td}, where α is only sometimes empty, where it may be necessary to use *glc* with an empty α. Hausarbeit for the mathematically inclined ...

7 Some useful bits of code The following "shell program" can be found in

```
~cornell/glc_ae_shell.pl
```

Feel free to copy this file to your working directory.

Also in that file is a bunch of grammar rules. Most of them are commented out. While you are debugging, you will want to have as few rules as possible! In the grammar, the possessive *'s* is written `$`.This is to distinguish it from the sentence category `s`.

```
%%% glc_ae_shell.pl
%%%
%%% Arc Eager Generalized Left Corner Recognition
%%% (Some of it)
%%%
%%% Tom Cornell
%%% December 9, 1999

:- use_module( library(lists), [append/3,reverse/2] ).


%%% User Interface Predicates
%%% -------------------------
glc( Strategy, String ):-
        glc_proof( String, [-s], Strategy ).

td( String ):-
        glc_proof( String, [-s], td ).

lc( String ):-
        glc_proof( String, [-s], lc ).

bu( String ):-
        glc_proof( String, [-s], bu ).
```

```
%%% glc_proof( String, Stack, StrategyCode )
%%% ---------------------------------------
%%% Your code goes here.
%%%




%%% f( StrategyCode, A, Alpha_r, Beta )
%%% ----------------------------------
%%% Interface to the grammar.
%%%
f( td, A, [], RHS ):- rule( A, RHS ).
f( bu, A, RHS, [] ):- rule( A, RHS ).
f( lc, A, [LC], RestRHS ):- rule( A, [LC|RestRHS] ).



%%% predict( CatList, PredictedCatList )
%%% -----------------------------------
%%% This just puts a minus sign in front
%%% of each category in CatList. This is
%%% used to form 'beta bar' from beta.
%%%
predict( [], [] ).
predict( [Cat|Cats], [-Cat|PredCats] ):-
        predict( Cats, PredCats ).


%%% This coroutines the following three goals:
%%%
%%%       append( Alpha, Gamma1, Gamma0 ),
%%%       strip( Alpha, Alpha0 ),
%%%       reverse( Alpha0, Alpha_r )
%%%
multipop_complete( Gamma0, Gamma, Alpha_r ):-
        multipop_aux( Gamma0, Gamma, [], Alpha_r ).

multipop_aux( Gamma, Gamma, Alpha_r, Alpha_r ).
multipop_aux( [+A|Gamma1], Gamma, As, Alpha_r ):-
        multipop_aux( Gamma1, Gamma, [A|As], Alpha_r ).


/* No longer used...
strip( [], [] ).
```

```
strip( [+Cat|Cats], [Cat|Stripped] ):-
        strip( Cats, Stripped ).
*/
```

Note: If you uncomment the left recursive possessive NP rule, you will probably get infinite searches with the top down strategy. (This depends on what VP rules you are using, I think.)