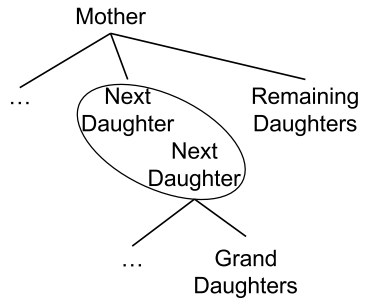
Tutorial 5b
Implementing Left-Corner

1. Review the algorithm as given in the first part of this tutorial, then draw the search tree for the left-corner arc-standard parser for *"John loves Mary"*.
   - Use "simplified" slash notation, in which atoms are used to represent complete elements (e. g. writing `NP` instead of `NP/[]`),
   - Consider the grammar to be binary (*John* and *Mary*: NPs), and
   - Expand only the "shift" operations needed to reach the correct parse.
2. Make a copy of your last Parser Architecture, rename it Left-Corner, and implement a basic version of the left-corner arc-standard parsing rules.
   - Since a Prolog list is used to represent the stack, the newest element will always be on top and the second-newest element second in the list. Therefore merging will turn `[n, np/n]` into `[np]` (note the ordering).
   - Whenever possible, use variable names that indicate relative node positions in the tree (e. g. `Mother, Daughter,`…), as this will make your code easier to understand and expand.
3. Check that your model behaves as expected by following the steps in your search tree. For now, take no heed that there is more than one element on the stack at the end of the sentence.
4. In your tree, modify complete constituents to use full slash notation, including the empty lists which explicitly denote that nothing is missing. Then modify the parse rules in your model accordingly. Note that all changes should be made in the parse rules; it is not necessary to modify either lexicon or grammar.
5. Add the arc-eager derivation for *"John loves Mary"* to your tree, then duplicate the arc-standard "merge" rule, comment out the first and modify the second to obtain the arc-eager variant.
6. Test your parser, and make sure both arc-standard and arc-eager can parse the sentence. You can switch between the two versions by commenting one rule out and uncommenting the other.
7. Draw the search tree for *"John chases the black cat"*.
   - Include both arc-standard and the arc-eager derivations.
   - Use the n-ary grammar from previous tutorials.

- As in the other parsers, the trick here is to group the right-hand side elements of a grammar rule as a list. Therefore, on the stack, missing elements will also become lists: If the grammar contains NP → Det Adj N and we find a determiner, it will be replaced by `NP/[Adj N]`.
- Make sure to use correct bracketing for all the lists, especially the singletons. You may however use "simplified" notation for complete elements.

8. Extend your parser so it can deal with n-ary rules. As before, modify arc-standard first, test the parser, and then go on with arc-eager. In arc-eager, you will need to use "append" to combine all missing elements to one list, as shown in the figure below, in which the two postulated `NextDaughter` nodes are merged, leaving `GrandDaughters` and `RemainingDaughters` to be found, in that order.



9. Now serialize the parser to prevent it from adding several elements to the stack: Add a Possible Operations buffer, the definition for `possible_operations/2` as well as the rule that selects an operation at random, then split each parse rule into two parts as in the previous tutorials. Check your previous models if needed to remind yourself how to obtain a clean "propose-execute" cycle.

10. Load the stimuli, lexicon and grammar provided on the tutorial page, and test the model by parsing each sentence in turn, commenting out unnecessary grammar rules as needed. Both versions of the parser should principally be able to parse all sentences, although it will often make wrong choices and not be able to backtrack.

11. Add backtracking to the model. Load the rules from the tutorial page, change the success condition, fill in the question marks as in previous tutorials, and make the necessary modifications to the input/output rules and to the stimuli. Then try to run an "experiment" with five participants with either parser variant (this may take a while!)

12. You will notice that some sentences still constitute a problem when the parser backtracks. Add "_ is in Current Word" to all parse rules (both to the propose and execute parts) to solve this problem and run the "experiment" again.

13. As you can see, the parser needs to backtrack a lot before finding the correct parse. This was already true under point 11, and the fix we added in the previous step has made the problem only worse, because now "shift" is possible at every choice point, including those where it was not before because no input was available in Current Word. Temporarily make *John* and *Mary* NPs again in the lexicon and comment

out all unnecessary rules in the grammar, then use your parser to expand the missing "shift" branches of the first search tree you drew in this tutorial.

14. If you give the content of the stack along the new branches of your search tree a closer look, you will notice that those paths had no chance in succeeding in the first place. For example, if, after cycle 5 (step 2 in the search tree), we do not use "predict" to make an `s/[vp]` out of the `np/[]`, but instead immediately shift the verb onto the stack to obtain `[np/[], tv/[]]`, there is no way the parser will ever be able to integrate the NP and the VP using "merge".

    One way to reduce the extensive backtracking in our parser is to have it explore the most promising paths first. In order to do this, we are going to incorporate a new mechanism which will introduce precedence between the parse rules, instead of letting chance decide. This will make sure that anytime "predict" is possible, it will be prefered to "shift", and anytime "merge" is possible, it will be prefered over both other operations.

    a) Make a copy of your model.

    b) Add scores to the "propose" part of the three parse rules. Change the actions of the rule that proposes shifting to read:

        add `score(1, shift(Word,Category))` to Possible Operations

        Then modify the other rules in similar fashion, "predict" having a value of 2 and "merge" a value of 3.

    c) Replace possible_operations/2 and the choice rule with the rules from the tutorial page, and make the necessary changes to the execute rules.

    d) Run a new "experiment" with five participants, and compare the total number of cycles needed to parse all sentences with the one under point 12.

15. Unfortunately, the previous mechanism does not reduce useless backtracking as much as might have be hoped. Luckily, we have at least one more trick in our bag: We are going to add an "oracle" to the parser.

    a) Add a buffer called Oracle to your model, and make it ungrounded by unchecking the corresponding box in its properties.

    b) Compile the list of possible left corners according to the grammar, add them to the buffer, and add *`lc(Cat,Cat)`* as the last element of the list.

    c) Initialize the stack with *`s/[s]`* to ensure the parser always has a goal.

    d) Request new stack elements to be potential left-corners of the top-most category by adding "*`lc(Variable,Goal)`* is in Oracle" to the conditions of the "shift" and "predict" rules.

    e) Adapt the value of `<Variable>` and make sure the variable `<Goal>` is bound: It is always the left corner of the stack element preceding the one to be added or modified.

    f) Run a final experiment like the previous ones, and report the numbers for all experiments. Can you tell why the number of cycles in the last model still is not exactly the same every time?