Name:

Parsing in Prolog C'td

Tutorial 2

The Parser Files

- The source code for the parsers you talked about in the lecture can be found at `/proj/courses/comppsych/Tutorial2`
- Copy the files to your own directory.

The Standard LC-Parser

1. How does it work?

The parsing strategy of an LC parser is a bit different from the Shift-Reduce parser you saw last week. Remember that this type of parser combines top-down and bottom-up parsing by finding out which top-down categories a word can start (be the left corner of). The parser then goes on parsing, trying to find proof for the category it expects. The initial assumption is that the input words will form a sentence.

Load the lc-standard parser into Prolog and parse the `easy` sentence. Again, the parser outputs the stack (with phrases it still needs proof for) and the list of words it needs to read yet. Explain why the stack looks the way it does at each step:

2. How is it implemented?

The code for the lc-standard parser looks much more involved, but don't worry. The grammar and lexicon, for example, are reassuringly familiar. Look at the `parse` predicate with the five arguments (for Prolog this is a completely different predicate from the one with the same name that has only one argument!). The arguments are: the phrase that needs verification, the input sentence, the input sentence after a word has been read, the stack before anything has happened and the stack after we've parsed the current word.

Let's step through the parse rule. Reading the current word is done by the `connects` predicate. Can you see how?

The next line looks familiar again: we're looking up a word's category in the lexicon. The `link` predicate from the lecture implements the oracle, which is basically a work-saving trick: if the current word's category can't be at the left corner of the phrase type we want to find, we have made a mistake somewhere and need to go back and try again. This "going back and trying again" happens automatically through Prolog's backtracking.

The last and most interesting predicate is `lc`, which does all the real work. Firstly, if the phrase we've found and the phrase we are looking for are the same, it just links them up.

Otherwise, it looks for the grammar rule that links the current category to its mother phrase and makes sure (via the oracle) that we remain on the right track to finding the phrase we are really after (remember that a determiner can be the left corner of a sentence, but initially, we would find an NP for its mother phrase; the oracle reassures us that we're still moving in a potentially right direction). Then, `lc` calls `parse` again, asking it to find whatever is missing to complete the phrase starting with the current category.

Make a little diagram of how `parse` and `lc` call each other up to the third word of the `easy` sentence. For each `parse` call, please show the phrase and input sentence, and for each `lc` call, please show both phrase types and the input sentence. For both calls, also give the lexical or grammar rule that is instantiated in them.

Calls could look like this:

```
Call: (13) parse(vp, [read, the, book], _L402, [vp, s], _L403) ?

Call: (14) _L460--->[read] ? creep
Exit: (14) vt--->[read] ?
```

Arc-Eager Parsing

1. How is it done?

Arc-eager parsing is just a variant of arc-standard parsing. Look at lc-eager.pl. The only difference to the arc-standard parser is one extra definition for `lc.` Compare it to the (still existing) old definition. The eagerness to accept a found phrase type as soon as possible is implemented as follows: The old procedure was to completely finish each grammar rule before the newly found left-hand side could be linked to anything (first parsing of Right, then the call to `lc`). In the new rule, we check immediately whether the phrase we're working on is the left corner of the phrase we're looking for. If this is the case, we parse just the rest of the rule, and *take the intermediate goal off the stack.* This is done by matching the stack `StIn` to `[_|St]` and calling parse with just St (the stack minus its first element).

Load the lc-eager.pl file and run the four sentences. Trace the easy sentence and see how the stack is handled differently. Here's an example:

Call: (12) lc(np, s, [read, the, book], [], [np, [det, [the]], [n, [man]]], _L363, **[s],** []) ?
Call: (13) s--->np, _G416 ?
Call: (13) parse(vp, [read, the, book], [], _G412, **[vp],** []) ?

Look at the stack (in bold): `lc` is called with [s] on its stack, but calls parse with just [vp] on the stack, assuming that this is the only thing missing to complete the sentence.
Trace a parse and make sure you understand what happens at each step.

2. What's the difference?

Look at the stack size. What is the difference to the arc-standard parser? Why does this happen? How do these two parsers compare to the shift-reduce parser?