Übung 1

November 5, 1999

The object of today's exercise is to write a predicate which will

1. prompt the user to enter a sentence, terminated by a period;
2. convert the string of characters typed in by the user to a list of Prolog atoms;
3. assert a collection of unit clauses representing that list into the Prolog database.

If you have time you can also look into ways to eliminate step 2, for example by asserting the relevant clauses as each word is read.

You will probably need to refer to the SICStus manual from time to time. Remember that there is a link to it from the main course web page.

1 The first step is trivial. It is just a matter of writing some instructions to the console. If you want to experiment with `prompt/2`, then this would be a good place to call it, to set your new prompt.

2 Name this predicate `getSentence/1`. The `getSentence` predicate can be defined in two ways. First, you can read the entire string of characters into a list, and then break the list up into words in a second stage. Alternatively, you can tokenize the user's input into words as it is being read.

Either way, your main tool will be the SICStus builtin `get0/1`. The goal `get0(C)` will succeed if `C` can be unified with the character code of the next character typed in by the user. The first call to `get0/1` will cause the interpreter to print its input prompt, either `|:` or whatever you changed that to in step 1. The prompt will thereafter only be printed when the user types return (and the program has not yet terminated).

There is also a SICStus builtin called `get/1`, which is like `get0/1` except that it ignores whitespace. You will not want to ignore whitespace: that's what tells you when you have finished getting one word and should prepare to get the next.

Keep in mind that `get0/1` has a side effect: it advances the input stream over the character that has been read. This side effect is not undone on backtracking! So be very sure that it is only called in a deterministic part of your program. All nondeterminism (i.e., decision making) should be postponed until after the call to `get0`.

If you choose to tokenize the input stream while you are reading it, you will probably want to define an auxiliary predicate `getWord`, which `getSentence` calls until `getWord` recognizes a period in the user's input.

Remember that you can use the expression '`0'.`' to represent the character code for a period. You can even use '`0' `' to represent the character code for a space, but your code will be more readable if you use the integer 32 instead.

For converting a word (represented as a list of character codes) from a string to an atom, you will want to use the builtin `name/2`. The goal `name(Atom,String)` works in both $(+,-)$ and $(-,+)$ modes. That is, you can give it an atom and get a string back in the second argument or you can give it a string and get an atom back in the first argument. Obviously, it is this second mode that will be useful here.

Optional: What happens if the user starts the sentence with a capital letter? You don't want to have to have in your lexicon entries for both *this* and *This*. So you might consider converting the string representation of a word to all lower case letters before calling `name/2`.

3 That was the hard part. Now that you have a list of atoms, you need to assert for each atom a goal of the form `connects(Word,P0,P)`, where `P0` and `P` are integers. You should also assert a single unit clause defining `sentencelength/1` so that calls to `sentencelength(N)` will bind `N` to the last position in the sentence (the sentence position immediately to the right of the last word).

You will, of course, be running this program many times during your development and testing of it. And if it were part of a real parser (as it will be, a month from now), you would be running it every time a user wants to parse a new sentence. So you must also take care to delete from the database all old clauses in the definitions of `connects/3` and `sentencelength/1` before you assert any new ones. Should you use `retractall/1` or `abolish/1` for this? Or does it matter? (It's an empirical question: try experimenting.)

Remember that `abolish/1` takes a predicate spec for its argument (e.g., "`connects/3`"), while `retractall/1` takes a goal (e.g., "`connects(_,_,_)`").

You can use calls to `listing/0` or `listing/1` to see whether your program works. `listing/0` will list all the (dynamic) clauses in the current database. `listing/1` takes a predicate spec as its argument and lists only the clauses defining that predicate.

It is unnecessary to declare `connects/3` and `sentencelength/1` to be dynamic, but it is good practice to do so, so you might want to include

```
:- dynamic connects/3.
:- dynamic sentencelength/1.
```

at the beginning of your program.

Optional: Some things to think about. What about allowing the user to enter questions? If you convert everything to lower case, you wind up with expressions like `cia` or `dfg`, and people's names wind up as `john` or `wolfgang`. Is there anything you can do about that? And speaking of names, what about periods that appear after people's initials, or in abbreviations, e.g., "e.g." Try and think about all the other stuff that appears in text that is not so easy to tokenize (numbers, for example).