1.5.2 Extending the DCG

Let's see how to use this notation in DCGs. We'll use a small DCG with e.g. an intransitive verb and a proper name as well as the necessary rules to use them in sentences. To use the resulting DCG for semantic construction, we have to specify the semantic representation for each phrasal and lexical item. We do this by giving additional arguments to the phrase markers of the DCG.

The resulting grammar is found in See file `semanticDCG.pl`.. Let's have a look at the phrasal rules first:

```
s(NP@VP) --> np(NP),vp(VP).

np(DET@N) --> det(DET),n(N).
np(PN) --> pn(PN).

vp(TV@NP) --> tv(TV),np(NP).
vp(IV) --> iv(IV).
```

The unary phrasal rules just percolate up their semantic representation (here coded as Prolog variables `NP`, `VP` and so on), while the binary phrasal rules use `@` to build a semantic representation out of their component representations. This is completely transparent: we simply apply function to argument to get the desired result.

1.5.3 The Lexicon

The real work is done at the lexical level. Nevertheless, the lexical entries for nouns and intransitive verbs practically write themselves:

```
n(lambda(X, witch(X))) --> [witch], {vars2atoms(X)}.
n(lambda(X, wizard(X))) --> [wizard], {vars2atoms(X)}.
n(lambda(X, broomstick(X))) --> [broomstick], {vars2atoms(X)}.
n(lambda(X, man(X))) --> [man], {vars2atoms(X)}.
n(lambda(X, woman(X))) --> [woman], {vars2atoms(X)}.

iv(lambda(X, fly(X))) --> [flies], {vars2atoms(X)}.
```

If you do not remember the somewhat difficult representation of transitive verbs, look at Section 1.4.4 again. Here's the lexical rule for our only transitive verb form, 'curses':

```
tv(lambda(X, lambda(Y, X@lambda(Z, curse(Y,Z))))) --> [curses], {vars2atoms
tv(lambda(X, lambda(Y, X@lambda(Z, love(Y,Z))))) --> [loves], {vars2atoms(X
```

Recall that the λ-expressions for the determiners 'every' and 'a' are $\lambda P.\lambda Q.\forall x.(P@x \rightarrow Q@x)$ and $\lambda P.\lambda Q.\exists x.(P@x \wedge Q@x)$. We express these in Prolog as follows:

```
det(lambda(P, lambda(Q, exists(X, ((P@X) & (Q@X)))))) --> [a], {vars2atoms(
det(lambda(P, lambda(Q, forall(X, ((P@X) > (Q@X)))))) --> [every], {vars2at
```

Finally, the 'role-reversing' (Section 1.4.4) representation for our only proper name:

```
pn(lambda(P, P@harry)) --> [harry], {vars2atoms(P)}.
pn(lambda(P, P@john)) --> [john], {vars2atoms(P)}.
pn(lambda(P, P@mary)) --> [mary], {vars2atoms(P)}.
```


Prolog Variables?

Note that we break our convention (page 6) of representing variables by constants in these lexical rules. All the λ-bound variables are written as Prolog variables instead of atoms. This is the reason why we have to add the calls to `vars2atoms/1` in some of our phrasal rules (included in curly brackets - curly brackets allow us to include further Prolog calls with DCG-rules). Whenever a lexical entry is retrieved, `vars2atoms/1` replaces all Prolog variables in it by new atoms. Distinct variables are replaced by distinct atoms. We won't go into how exactly this happens - if you're interested, have a look at the code of the predicate. After this call, the retrieved lexical entry is in accord with our representational conventions again.

This sounds complicated - so why do we do it? If you have read the sidetracks in the previous section (Section 1.4.7 and Section 1.4.8), you've heard about the possibility of accidental binding and the need for α-conversion during the semantic construction process. Now by using Prolog variables in lexical entries and replacing them by atoms on retrieval, we make sure that no two meaning representations taken from the lexicon ever contain the same λ-bound variables. In addition, the atoms substituted by `vars2atoms/1` are distinct from the ones that we use for quantified variables. Finally, no other rules in our grammar ever introduce any variables or double any semantic material. In result accidental bindings just cannot happen. So using Prolog variables in the lexicon may be a bit of a hack, but that way we get away without implementing α-conversion.

1.5.4 A First Run

Semantic construction during parsing is now extremely easy. Here is an example query:

?- `s(Sem,[harry,flies],[]).`

```
Sem = Sem=lambda(v1, v1@harry)@lambda(v2, fly(v2))
```

Or generate the semantics for 'Harry curses a witch.': `s(Sem,[harry,curses,a,witch],[]).`

The variables `v1,v2` etc. in the output come from the calls to `vars2atoms` during lexical retrieval. The predicate generates variable names by concatenating the letter `v` to a new number each time it is called.

So now we can construct λ-terms for natural language sentences. But of course we need to do more work *after* parsing, for we certainly want to reduce these complicated λ-expressions into readable first-order formulas by carrying out β-conversion. For this purpose we will now implement the predicate `betaConvert/2`.

1.5.5 Beta-Conversion

The first argument of `betaConvert/2` is the expression to be reduced and the second argument will be the result after reduction. Let's look at the two clauses of the predicate in detail. You find them in the file See file `betaConversion.pl`..

```
betaConvert(Functor@Arg,Result):-
      betaConvert(Functor,lambda(X,Formula)),
```

```
        !,
        substitute(Arg,X,Formula,BetaConverted),
        betaConvert(BetaConverted,Result).
```

The first clause of `betaConvert/2` is for the cases where 'real' β-conversion is done, i.e. where a λ is thrown away and all occurences of the respective variable are replaced by the given argument. In such cases

1. The input expression must be of the form Functor@Arg,
2. The functor must be (recursively!) reducible to the form `lambda(X,Formula)` (and is indeed reduced to that form before going on).

If these three conditions are met, the required substitution is made and the result can be further β-converted recursively.

This clause of `betaConvert/2` makes use of a predicate `substitute/4` (originally implemented by Sterling and Shapiro) that we won't look at in any more detail. It is called like this:

```
substitute(Substitute,For,In, Result).
```

`Substitute` is substituted for `For` in `In`. The result is returned in `Result`.

1.5.6 Beta-Conversion Continued

Second, there is a clause of `betaConvert/2` that deals with those expressions that do not match the first clause. Note that the first clause contains a cut. So, the second clause will deal with all *and only* those expressions whose functor is *not* (reducible to) a λ-abstraction. The only well-formed expressions of that kind are formulas like `walk(john) & (lambda(X,talk(X))@john)` and atomic formulas with arguments that are possibly still reducible. Apart from that, this clause also applies to predicate symbols, constants and variables (remember that they are all represented as Prolog atoms). It simply returns them unchanged.

```
betaConvert(Formula,Result):-
        compose(Formula,Functor,Formulas),
        betaConvertList(Formulas,ResultFormulas),
        compose(Result,Functor,ResultFormulas).
```

The clause breaks down `Formula` using the predicate `compose/3`. This predicate decomposes complex Prolog terms into the functor and a list of its arguments (thus in our case, either the subformulas of a complex formula or the arguments of a predication). For atoms (thus in particular for our representations of predicate symbols, constants and variables), the atom is returned as `Functor` and the list of arguments is empty.

If the input is not an atom, the arguments or subformulas on the list are recursively reduced themselves. This is done with the help of:


```
betaConvertList([],[]).
betaConvertList([Formula|Others],[Result|ResultOthers]):-
        betaConvert(Formula,Result),
        betaConvertList(Others,ResultOthers).
```

After that, the functor and the reduced arguments/subformulas are put together again using `compose/3` the other way round. Finally, the fully reduced formula is returned as `Result`.

If the input is an atom, the calls to `betaConvertList/2` and `compose/3` trivially succeed and the atom is returned as `Result`.

Here is an example query with β-conversion:

```
?- s(Sem,[harry,flies],[]), betaConvert(Sem,Reduced).

Sem = lambda(A,A@mary)@lambda(B,walk(B)), Reduced = fly(harry)
```

Try it for 'Harry curses a witch.': `s(Sem,[harry,curses,a,witch],[]), betaConvert(Sem,Res).`

?- Question!

Above, we said that complex formulas like `fly(harry) & (lambda(x,fly(x))@harry)` are split up into their subformulas (which are then in turn β-converted) by the last clause of `betaConvert/2`. Explain how this is achieved at the example of this particular formula!

1.5.7 Running the Program

We've already seen a first run of our semantically annotated DCG, and we've now implemented a module for β-conversion. So let's plug them together in a driver predicate `go/0` to get our first real semantic construction system:

```
go :-
        readLine(Sentence),
        resetVars,
        s(Formula,Sentence,[]),
        nl, print(Formula),
        betaConvert(Formula,Converted),
        nl, print(Converted).
```

This predicate first converts the keyboard input into a list of Prolog atoms. Next, it does some cleaning up that is needed to manage the creation of variable names during lexicon retrieval (see Section 1.5.3). Then it uses the semantically annotated DCG from See file `semanticDCG.pl`. and tries to parse a sentence.

Next, it prints the unreduced λ-expression produced by the DCG. Finally, the λ-expression is β-converted by our predicate `betaConvert/2` and the resulting formula is printed out, too.

In order to run the program, consult `runningLambda.pl` at a Prolog prompt: