Assignment1 – SymNLProj: Semantics Construction in ELPI

In the previous assignment, you have implemented a grammar in GF for different fragments of English. In this assignment, the goal is to translate the abstract syntax trees (ASTs) generated by the GF grammar into logical expressions. As a programming language, we will use λProlog, and more specifically ELPI.

As the name suggests, it is a combination of λ-calculus and Prolog, which is very useful for our purposes. ELPI is a niche language with limited documentation (though there are resources for λProlog in general). Therefore:

- make sure you understand the basics of Prolog and the simply typed λ-calculus because ELPI is based on these concepts,
- we will provide a few examples of ELPI code and relevant ideas at the end of this assignment,
- we will provide a mini fragment with semantics construction in ELPI as an example (see the assignment repository),
- you should feel encouraged to ask questions whenever you are stuck or something is unclear.

Tasks

For each fragment from the previous assignment, you should implement a semantics construction in ELPI. You can modify your GF grammar to make the translation easier if necessary.

Fragment 1

For fragment 1, the semantics construction should result in PLNQ expressions as described in the lecture notes. For example,

Ethel poisoned the cake and Prudence laughed

should be translated into

$$poison(ethel, thecake) \wedge laugh(prudence)$$

Note that you do not have to produce that specific output. Instead, you can produces an ELPI term like this:

```
and (poison ethel thecake) (laugh prudence)
```

NP fragment

This fragment includes complex noun phrases. The semantics construction should correspond to fragment 4 from the lecture. For example,

Ethel poisons the cake and every happy dog laughs

should be translated to

$$poison(ethel, \iota cake) \wedge \forall x.(dog(x) \wedge happy(x)) \rightarrow laugh(x)$$

PP fragment

For this fragment, you should use event semantics (which will be explained in the lecture very soon). For example,

Ethel poisons Peter with Prudence

should be translated to

$$\exists e.tp(e, poison) \wedge ag(e, ethel) \wedge with(e, prudence))$$

Submission and Points

At the deadline, you have to submit:

1. All your code,
2. a README file that briefly explains how to use your code (what files are relevant for what, etc.).

You can get up to 100 points for this assignment.

ELPI

Basics

ELPI syntax is similar to Prolog, but the notation of function application is based on the λ-calculus. Additionally, ELPI is typed.

For example, we can write the following Prolog code

```
member(X, [X|_]).
member(X, [_|T]) :- member(X, T).
```

in ELPI as follows:

```
type member A -> list A -> prop.
member X [X|_].
member X [_|T] :- member X T.
```

The type declaration has a variable *A* to make the predicate polymorphic (i.e. it can work on integer lists, string lists, etc.). If we only wanted lists of integers, we could write **type** member **int** −> list **int** −> **prop**.. The type declaration states that the member predicate gets two arguments: an *A* and a list of *A*s. As it is a predicate, it returns a proposition (**prop**).

Specifying a logic

In ELPI, we can define our own types and functions, which lets us specify the syntax of a logic. We will use the type oo for propositions to keep it short (o is already used by ELPI).

```
% oo is a type (the type of propositions). We use kind to define a new type.
kind oo type.
% neg is a function that takes a proposition and returns a proposition.
type neg oo −> oo.
% and is a function that takes two propositions and returns a proposition.
type and oo −> oo −> oo.
% etc.
```

To get an impression on how to use this, let's define a predicate for removing double negations:

```
type noDoubleNeg oo −> oo −> prop.
noDoubleNeg (neg (neg P)) P1 :− !, noDoubleNeg P P1.
noDoubleNeg (and P Q) (and P1 Q1) :− !, noDoubleNeg P P1, noDoubleNeg Q Q1.
noDoubleNeg (neg P) (neg P1) :− !, noDoubleNeg P P1.
noDoubleNeg P P.

% let's define some propositions so we can test our logic
type phi oo.   % phi is a proposition.
type psi oo.   % psi is a proposition.

% a test case
type test prop.
test :− noDoubleNeg (and (neg (neg phi)) (neg psi)) X, print "Result:", print X.
```

To run this code, enter test. in the ELPI shell.

Quantifiers via Higher-Order Abstract Syntax (HOAS)

Quantifiers like ∀ bind variables. There is a trick to represent this in systems ELPI: higher-order abstract syntax (HOAS).

Here is the relevant part of the code:

```
kind ind type.   % individuals
type forall (ind −> oo) −> oo.
```

So forall is a function that takes a function from individuals to propositions and returns a proposition. This may seem a bit odd at first, but it is a very powerful concept. There are two different ways to think about this:

1. $\forall$ is a binder, i.e. it binds a variable. In ELPI, we already have a way to bind variables: λ functions. In HOAS, we simply re-use this binding mechanism for other binders. The ELPI notation for $\lambda x.p$ is `x \ p`. If want to represent the expression $\forall x.p(x) \wedge q(x)$, we can write `forall (x \ and (p x) (q x))`.
2. forall takes a predicate as an argument. So `forall p` states that `p` evaluates to true for all possible arguments.

GF and ELPI

The assignment repository has code for calling GF from ELPI. Specifically, the parse predicate takes the path to the GF concrete syntax and a string to parse and (on the third argument) returns the AST as a string. Example call:

```
parse "path/to/MiniFrag1Eng.gf" "it is not the case that we detected a pit" AST.
```

If this does not work for you, you can just hard-code a few example ASTs in ELPI:

```
type parse string -> string -> string -> prop.
parse "MiniFrag1Eng.gf" "it is not the case that we detected a pit" "negate (detect pit)".
parse "MiniFrag1Eng.gf" "we detected a pit" "detect pit".
% ...
```

The same way that you can represent the syntax of a logic in ELPI, you can also represent the syntax of the abstract syntax trees generated by GF. Writing this down is very tedious and closely mirrors the GF abstract syntax. Therefore, the assignment repository has a script that generates this code for you. It prefixes all GF function symbols and categories with `g/` to avoid name clashes (ELPI identifiers may contain slashes). It also generates predicates for parsing the string representation of the ASTs into ELPI terms.

Tips

1. There is an example in the assignment repository – take a look at it.
2. Ask questions if you are stuck or something is unclear. Some of the involved concepts are tricky to understand the first time, so do not hesitate to ask for help.