

Semantiek 2012

Semantics with Haskell : HW-H2

Jeroen Breteler
Based on materials by Chris Blom

March 30, 2012

0 Introduction and Submission Guidelines

0.1 Aim of the Tutorial

This tutorial is about implementing semantics in Haskell. It is split up into basic and more advanced questions. In order to get a pass, students are expected to do well at the basic questions, and to make some attempt to crack the more advanced problems. For this goal, no prior knowledge of Haskell is necessary, and only basic programming experience is needed. Students with a stronger background in programming may find more depth in the assignment by fully solving the advanced problems.

0.2 Guidelines for Submission

The assignment can be made in pairs. The answers to the questions should be supplied partially in prose, and partially in the form of Haskell code files. If a question asks you to write or edit code files (marked by `(code)`), submit the relevant code files. In the other cases (marked by `(prose)`), such as when you are asked to report on something or to work with the interpreter, write down the answers in a document. Submit your solutions in the form of a ZIP-archive containing all relevant text and code files to *j.m.w.breteler@students.uu.nl*. The topic line of the email message should read:

"[Semantiek HW-H2] Name1_StudentID1_Name2_StudentID2".

The deadline for submission is **April 13, 23:59**.

1 Generalized Quantifiers

Firstly, welcome back to Semantics with Haskell! This tutorial is split up into two parts, with two different versions of the lexicon (and underlying framework). You are advised to work on only part at a time, although both parts are compulsory. To get set up, let's load `Lexicon.hs` from the `Part1` folder into

a text editor, and `Parser.hs` in `ghci`. The lexicon has expanded since the last time we visited. In particular, quantification has been added in the form of `exists` and `forall`, we can use `card` to compute the cardinality of a set (i.e. the arguments of a characteristic function that return `True`), and the boolean operator `.<.` to indicate subsets. These helper functions may make life easier for our first task: implementing generalized quantifiers and determiners. The lexicon contains examples of *everyone*, *someone*, and *no-one* to help get you started.

1. (code) Provide definitions of *every*, *some*, and *no*. (prose) For each of these, give an example sentence that will be `True` and one that will be `False` if we feed it to our parser.

Next, let's implement some generalized quantifiers that take an extra integer argument:

2. (code) Provide definitions of *at_most*, *at_least*, and *exactly*.
Hint: recall that you can use the `card` function to find the cardinality of sets.

Note that since our GQs are Boolean types, we can easily coordinate them. For example, we are able to parse the following sentence:

```
parse "some human and some alien are tall"
```

However, we cannot coordinate entities. For example, the sentence "some human and Chewbacca are tall" will fail to parse. This is because *e* is not a boolean type.

3. (code) Advanced: Define a 'lift' function that turns entities into their corresponding GQs. Make sure to include the type for the definition. Also include a string `"^"` in the lexicon to represent this lift function, and make sure it has the correct syntactic type.

2 Restrictive Modifiers and Adjectives

Another thing we can implement are restrictive modifiers. The lexicon contains a helper function `isRestrictive` that we can use to check if an *(et)et* function is restrictive.

1. (code) Provide definitions of *quickly* and *bravely*. (prose) For each of these, give an example sentence that will be `True` and one that will be `False` if we feed it to our parser.

Hint: You can ask for a specific value of an argument by specifying the argument in the definition. Conversely, you can disregard the value of an argument with `"_"`. For example, to create a function that returns `True` if its argument is the integer 5, we could write:

```
func :: Int -> Bool
func 5 = True
func _ = False
```

Similarly to restrictive modifiers, we may want *(et)et* versions of our current adjectives. This way, we could express sentences that in our current framework cannot be parsed, such as "Vader is a tall human". We will do this by having the syntax recognize adjectives that are in a modifier position, and then assigning them a different denotation from the usual one. To do this, we need a general function that will take an adjective and return a restrictive modifier that we can apply to the noun that is being modified by the adjective.

2. (code) Define a function `to_mod` that turns *et* functions into restrictive modifiers. In the lexicon, add entries for our adjectives in modifier position, using the syntactic category 'adj' and the `to_mod` function. Please put the lexical entries at line 211 and further.

3 Intensionality

The following questions are all advanced.

Save and close the files we have worked with so far. We will now switch to the files in the 'Intensionality' folder. Inside this folder, open `Lexicon.hs`. This is a version of the lexicon that works with the possible-world model, and it has been kept separate from our extensional semantics to keep things a bit more comprehensible.

Take a moment to examine the new lexicon file. It contains an explanation of the possible worlds and entities that are available to us, and examples for types and definitions.

We will model example (7) from the first handout on intensionality in our lexicon file. The handout can be found here:

<http://www.phil.uu.nl/~yoad/semantiek2012/6-handout-intensionality.pdf> .

1. (code) Define the concepts (type *se*) *tina*, *mary*, and *john*. Please start writing from line 165, so that I'll know where to look with checking. Note that you have some freedom in how you assign the entities to these concepts in different worlds.

For these assignments, we will just focus on getting the semantics right, meaning that we will not use the parser. Instead, we can put our formulas straight into the prompt of `Lexicon.hs`, which will still provide us with pretty-printed outputs. For example, try writing just `tina` in the prompt.

Next, we will define the intransitives:

2. (code) Define the properties (type *(se)(st)*) *smile* and *dance* so that they are in accordance with the facts in the handout.

Again, you can check if your code is working by simply feeding the formula into the parser, like this: `smile tina`.

Finally, we will define the transitive verb **believe**.

3. [\(code\)](#) Define believe in accordance with the handout.

We can now test if everything has been put together correctly to provide the proof by checking the following statements:

- `smile tina (I 1)` : Should be True
- `dance mary (I 1)` : Should be True
- `believe (smile tina) john (I 1)` : Should be True
- `believe (dance mary) john (I 1)` : Should be False