

Semantiek 2012

Semantics with Haskell : HW-H1

Jeroen Breteler
Based on materials by Chris Blom

March 10, 2012

1 Introduction and Submission Guidelines

1.1 Aim of the Tutorial

This tutorial is an introduction to 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 solving the advanced problems.

1.2 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, submit the relevant code files. In the other cases, 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-H1] Name1_StudentID1_Name2_StudentID2". The deadline for submission is **March 23rd, 23:59**.

2 Getting acquainted with Haskell Expressions

Boot up GHCi. The interpreter will automatically load a base file, the Prelude, for us. We are now ready to evaluate expressions. Try entering the following expressions and report on your results:

Extra challenge: predict (and still report) the evaluations of these expressions without entering them into GHCi.

1. `True || False`
2. `True && False`
3. `not True`
4. `5*5`
5. `1+2+3`
6. `1/2`
7. `'c'`
8. `"hello world"`
9. `[1,2,3]`
10. `[1..10]`
11. `[1..]`

NB: Use `Ctrl+C` to interrupt Haskell's calculations.

12. `['a'..'z']`
13. `1 == 2`
14. `[1,2] == [1,2]`
15. `[2,3] == [3,2]`

3 Lambda notation

Translating lambda terms to Haskell is fairly straightforward. To translate $\lambda x.y$, we replace λ with `\`, and the dot `.` with `->`.

Some examples:

Lambda calculus	Haskell
$\lambda x.x \wedge 1$	<code>\x -> x && True</code>
$\lambda x.\lambda y.x + y$	<code>\x -> \y -> x + y</code>
$(\lambda x.x)(1)$	<code>(\x -> x) True</code>

Arguments can also be placed left of the `\`-sign, so $plus = \lambda x.\lambda y.x + y$ can also be written as `plus x = \y -> x + y`

1. Convert the following lambda calculus terms to haskell notation:

- (a) $(\lambda x.\lambda f.f(x))(True)(not)$
- (b) $(\lambda x.\lambda y.\lambda f.f(x,y))(True)(False)(and)$

Hint: the function we need for *and* is not called **and** in Haskell. Recall the logical operators in the previous question.

- (c) **Advanced:** Think of as many ways as possible to represent $\lambda x.\lambda y.\lambda f.f(x,y)$ as a Haskell function (excluding infinite variations such as (extra) ((pairs) of parentheses.))

Instead of working in the interpreter directly, we may prefer to work from our own files. Open the file `Lambdas.hs` with a text editor (Notepad++, among others, comes with nice colors for Haskell). This is a Haskell code file that contains names for the two lambda terms above, but the functions are still undefined.

- Using your answers from step 1, fill out the definitions. Define these functions as just the lambda functions, without the arguments. Run the program and test that your definitions work, e.g. by checking that inputting `lambda1a True not` in `ghci` will return `False`.

4 Defining a lexicon

Now we are ready to implement some of the semantics notions learned in class. Run the `Parser.hs` file. This file serves as the 'frontend' for our semantics framework. The program can parse simple strings and return a truth value for those strings. For example, we can evaluate the sentence "Chewbacca hates Vader" by asking the program to parse it: `parse "Chewbacca hates Vader"` will return the value `True`, and a derivation.

The `Parse` module draws on a lexicon to determine the meaning of sentences. This is stored, perhaps unsurprisingly, in the file `Lexicon.hs`. We will explore and expand the lexicon.

Open `Lexicon.hs` in a text editor. For now, ignore the top part of the module and scroll down to the part about "Combinators and Utility Functions". Currently, this holds `charf` and `charf2`. These functions represent the characteristic functions for entities and binary relations over entities, respectively. The set of entities in this case consists of Luke, Leia, Han, Chewbacca, Vader, and Yoda. Take a look at the "Denotations" section to see how `charf` and `charf2` can be used to define one-place and two-place predicates. Finally, scroll down further to see the actual `Lexicon`. This lists all the words with their definitions and syntactic types. Now try defining some predicates of your own as follows:

- Find the definition for "human" in the "Denotations" section. Uncomment the lines and complete the definition according to your intuitions of who is human. If you are not sure whether someone is human, make an assumption or look for fellow classmates that are Star Wars fans. Note: you will also need to uncomment the entry for "human" in the lexicon. After making the additions, save your lexicon file and then reload the `Parser.hs` module (there is a `GHCi` command for this, `:r`). You can now test whether your newly built predicate works by entering expressions such as `parse "Luke is human"`.

2. Similarly to the definition for "human", we can define "loves". This is a two-place predicate, though, so we should deal with sets of pairs of entities. You can take a look at the definition of `hates` for inspiration. Make it so that Luke and Leia love each other, Chewbacca loves Han, and Han loves Leia. Again, a type definition and a lexicon entry have been provided for you in comments.
3. **Advanced:** Define an entity George and a two-place predicate "created" and make it so that George created Luke, Leia, Han, Chewbacca, Vader, and Yoda. Make sure you include type definitions where necessary, and make sure that you include lexicon entries. You may assume that the lexicon knows the meaning of the term *George*.

5 Boolean Semantics

In addition to filling our lexicon with arbitrary relations, we may encode functional elements. Near the top of the `Lexicon.hs` file is a definition for conjunction, disjunction and negation functions. The `class` definition is a very general statement that tells Haskell something about the types of arguments that these functions are involved with. However, it does not do any actual work for us; we need to explain how these functions work for specific instances.

1. Uncomment the lines of the instance for the boolean type `T`. These definitions are for values of type `T`, where `x` and `y` can be either `True` or `False`. Fill in the definitions for `/\`, `\|`, and `compl`. Uncomment the entries in the lexicon that work on sentence coordination. You can now check your answer by attempting to parse coordinated sentences, e.g. `parse "Vader hates Luke and Vader hates Han"`.

This takes care of coordination for type `T`, but we have not yet taken care of the general case where `T` can only be derived once more arguments have been supplied. For example, in "Tina is thin and tall", we want to coordinate "thin" with "tall" before we even know that "Tina" will be supplied as an argument to both these functions.

2. **Advanced:** Uncomment the lines for the other instance and give the definitions. Don't forget to uncomment the lexicon entries. Hint: you can use lambda notation to define an argument that can be fed into the functions.

We can use these coordinators not just in parsing, but also in defining new lexicon items.

3. **Quite Advanced:** Define a predicate "alien" that makes use of the "compl" function and the "human" predicate that you defined earlier.

Furthermore, now that we have coordinators for the general case, we can in principle conjoin not just nouns, but also things such as transitive verbs. However, we are missing the lexicon items that give the coordinators this kind of power.

4. Quite Advanced: Add lexicon entries for the coordinators for transitive verbs. Note that you will have to supply explicit types to the denotations.