Semantics 3: Computational Formal Semantics with Haskell

Course Description & Syllabus

Fall 2013

1 Organizational matters

- Class: TuTh 2:00PM–3:45PM, Crwn Comp Lab 201
- Instructor: Adrian Brasoveanu

Office hours

- Adrian Brasoveanu: TuTh 3:45PM–4:30PM, Crwn Comp Lab 201 / Stevenson 259, and by email appt. (`abrsvn@ucsc.edu`)

Web access

- For private files, we will work in a shared Dropbox folder. Email the instructor if you do not have access to it already.
- Public files will be available on the LaLoCo lab web page[1] under appropriate Sem 3 related headings. E.g., the syllabus, lecture notes, problem sets, their solutions etc. will be posted there. Check regularly, e.g., before / after each class, every day an assignment is scheduled to be posted etc.

Biweekly schedule

- Assignments will generally be posted on Tuesdays.
- Assignments are due two weeks later (on Tuesday) unless otherwise specified, at the beginning of class. Please print all your code (if applicable) and also paste in all the output you get. The assignments should be submitted in hard copy, not electronically.
- Assignments have to be typed (obviously).
- Graded assignments will be returned in about a week after their due date.

[1] Just in case, this is the actual web address:
http://people.ucsc.edu/~abrsvn/category/laloco-lab-log-announcements-materials-etc.html.

Important note

If you qualify for classroom accommodations because of a disability, please get an Accommodation Authorization from the Disability Resource Center (DRC) and submit it to me in person outside of class (e.g., office hours) within the first two weeks of the quarter. Contact DRC at 459-2089 (voice), 459-4806 (TTY), or http://drc.ucsc.edu for more information on the requirements and/or process.

2 General Description

The course will introduce both compositional formal semantics (as introduced in chapters 2-4 of Dowty, Wall & Peters 1981) and show how to implement formal semantics theories in functional languages, Haskell in particular. More precisely:

- For the introduction to the compositional formal semantics part, we will use Dowty, Wall & Peters (1981), chapter 4 in particular, plus various extensions to be introduced in class by the instructor.
- As a general intro to Haskell, we will use lecture notes, to be provided by the instructor. These notes will be primarily based on the book *Learn You a Haskell for Great Good!: A Beginners Guide*, by Miran Lipovača, http://learnyouahaskell.com/. Various other online and offline resources will also be used.
- We will also use lecture notes for the computational formal semantics part, again to be provided by the instructor. The code and notes for this part will be primarily based on the textbook *Computational Semantics with Functional Programming*, by Jan van Eijck & Christina Unger, http://www.computational-semantics.eu/.

But why implement formal semantics theories, and more generally linguistic theories, in a functional language in the first place? For at least two reasons.

The first reason is that building programs in a functional language (Haskell in particular) and building meanings for natural language expressions (more generally, building models of human higher cognition) are very closely related endeavors and are fundamentally using the same kind of approach and mathematics (λ-calculus among other things). So learning both things at the same time provides two different perspectives on what is fundamentally the same theoretical approach.

Here's a quote from a foundational computer science textbook that gives a glimpse of this:

> "[...] another crucial idea about languages and program design [...] is the approach of stratified design, the notion that a complex system should be structured as a sequence of levels that are described using a sequence of languages. Each level is constructed by combining parts that are regarded as primitive at that level, and the parts constructed at each level are used as primitives at the next level. The language used at each level of a stratified design has primitives, means of combination, and means of abstraction appropriate to that level of detail." H. Abelson & G. Sussman, "The Structure and Interpretation of Computer Programs", http://mitpress.mit.edu/sicp/full-text/book/book-Z-H-15.html#%_sec_2.2.4.

This is very similar to the way the subdisciplines of linguistics are hierarchically layered: phonetics and phonology at the bottom, then morphology, then syntax and finally semantics and pragmatics. And within each layer / level, the linguistic representations at that level are compositionally assembled out of atomic parts according to specific rules of composition.

Even within the level of semantics, we can distinguish between various sub-layers. For example, there is a basic layer that encodes the elementary meaning operations contributed by single morphemes, e.g., a quantifier like *every* or *no* contributes multiple meaning operations – a new variable is introduced, then its values are constrained by a restrictor property and finally, they are checked against a nuclear scope property.

All of these meaning operations, i.e., the fact that a new variable is introduced and then threaded through a restrictor and a nuclear scope predicate in a particular way, are then encapsulated in a single function that hides / abstracts away all these details.

This single function is delivered to a higher layer that manages meaning composition. At this level, we simply require the (syntactic) structure of the larger sentence to be such that it can deliver appropriate restrictor and nuclear scope predicates. At this higher level of meaning composition, there is no need to know how the quantifier will semantically manipulate the predicates – we only need to know that the quantifier requires these predicates.

One major advantage of learning about types and compositionality using Haskell is that you can play with every new concept you learn in its interactive interpreter (`ghci`) and get instant feedback. Haskell is an ideal language to support learning about logic and discrete mathematics more generally. You can try out new ideas, explore various consequences of your hypotheses, refine them incrementally based on the feedback you receive etc. The kind of high-level, abstract work we do in formal semantics becomes more concrete and much more engaging in the interactive environment provided by `ghci`.

More on Haskell and logic / semantics. Here are two textbooks that use Haskell to introduce logic and discrete mathematics, a blog post giving 11 reasons to use Haskell for the kind of (discrete) math we use in formal semantics and generative linguistics more generally, and finally the website of a graduate level course that has the same general goals as ours :

- Discrete Mathematics Using a Computer 2nd ed., by John O'Donnell, Cordelia Hall & Rex Page
- The Haskell Road to Logic, Maths and Programming 2nd ed., by Kees Doets & Jan van Eijck (both formal semanticists)
- Eleven Reasons to use Haskell as a Mathematician
- What Philosophers and Linguists Can Learn from Theoretical Computer Science but Didn't Know to Ask, cotaught by Chris Barker and Jim Pryor

The second reason is that theoretical linguistics is changing. Concerns about how our theories of linguistic competence should be embedded in more general (performance) theories of human higher cognition are becoming more important. Practically speaking, this means that more and more research will be concerned with how formal semantic theories can be interfaced with some form of plausible / probabilistic inference. And understanding the theoretical import and empirical consequences of such probabilistic semantic theories is very difficult or straight-up impossible just using traditional pencil-and-paper methods. Thus, implementing non-probabilistic semantic theories of the kind we regularly develop as formal semanticists is a necessary first step towards this goal.

Haskell is a good choice for that because in addition to its interactive interpreter (`ghci`), which makes learning easier and implementation a lot more fun, it also has an excellent native-code compiler (`ghc`) that can make more complex simulations run (much) faster. Suggestive evidence for this is provided by these comparisons:

- Haskell vs. C
- Haskell vs. Python

And here's a recent blog post comparing Haskell and C with respect to a common numerical task. The upshot of this post is that (deep knowledge of) Haskell "lets us write high-level code without sacrificing high performance. The need for any compromise between the two is becoming more rare every day."

3 Course Requirements

The learning (and grading) tools are: lectures, more or less biweekly problem sets, and a final research project.

- Class attendance is a necessary part of this course. Reading the posted notes and doing the assigned readings is crucial but does not substitute for class attendance. Speaking up in class is strongly encouraged. If you cannot make a class, it is your responsibility to find out from a classmate what happened in the class you missed. Handouts will be posted on the LaLoCo site and / or in the Dropbox folder the night before class or the day of class. Reading them is important but does not substitute for class attendance.
- Written work for the course consists of more or less biweekly problem sets, posted and due according to the biweekly schedule above, and the final research project. Homeworks will NOT be handed out in class. Homeworks should **not** be submitted by email—they must be submitted in person at the beginning of the class on the day they are due. Again: please type all your hw assignments.
- It is an excellent idea to form study groups and discuss the problem sets in your meetings. However, **you should write up the assignments on your own**. If you do form study groups, please list the people you discussed the problem set with at the beginning of the assignment. Turning in identical hw assignments could count as plagiarism for all students involved.
- Homework policy: no late homework is accepted unless by prior arrangement (or because of a health problem properly documented to the satisfaction of the instructor). A student who misses more than 2 assignments automatically fails the course.
- There will be no midterm for this class, only a final research project.

Grade calculation

- Course attendance and participation: around 10%.
- Completing around 5 problem sets: around 60-70% (around 12-14% per problem set).

- Completing a final research project: around 20-30% (worth about 2 problem sets); the seemingly low percentage associated with the final project should not be taken as an indication that the final project is easy, but that the problem sets are generally (very) hard; in fact, they might be too hard for some of the participants, so the instructor might post example solutions for some of the problems sets even before they are due so that everyone can get started coding in Haskell; we will discuss this possibility as the course proceeds.

You should start working on your final project as early as the 7th week of classes. Either individual or team projects (in teams of 2-3 people) are acceptable. Part of the reason we're using Dropbox is that it makes this kind of small-team collaboration much easier and more streamlined.

The final projects for this class can vary a lot:

- purely theoretical projects, e.g., formulating a new research problem and writing a paper describing the phenomena and proposing an analysis; for this kind of projects, a correct formal account is required while an implementation in Haskell is merely recommended
- purely applied projects in which you take a logical system in the formal semantics literature (e.g., Parsons' sub-atomic event semantics) and implement it in Haskell in a satisfactory way, i.e., you show that your code works and that it derives the expected analyses for a significant range of examples that the logical system was designed to account for
- projects between these extremes, e.g., that propose a new analysis and implement (part of) it in Haskell, are also welcome and probably the most desirable.

4 Course structure (subject to change)

The following is a general outline of the progress of the course. Details are subject to change.

1: Getting started with Haskell (Weeks 1-3)

We will spend most of our first 5-6 classes introducing the basics of Haskell and getting comfortable interacting with the `ghci` interpreter. The relevant lecture notes and associated Haskell code are most probably already available on the LaLoCo lab web page and / or in our Dropbox folder.

The Haskell Platform and editors. The best way to get Haskell up and running on your computer is to install the Haskell Platform (`http://www.haskell.org/platform/`), available for all major operating systems.

You can use whatever text editor you want and simply copy-paste code into `ghci`. But certain professional editors provide Haskell support, i.e., syntax highlighting and possibly a convenient way to run pieces of code in `ghci`. Here are some resources – Geany and Gedit are probably the easiest editors to get started with:

- Geany (`http://www.geany.org/`) is a lightweight editor; go here `http://www.geany.org/Download/Releases` to install it on Windows (consider choosing the full installer with GTK included) and here `http://wiki.geany.org/howtos/osx/running` to install it on Mac OSX

- Gedit is another good editor that is straightforward to use and has support for all major operating systems; go here https://projects.gnome.org/gedit/ and follow the relevant link in the 'Downloads' sidebar on the right-hand side of the page
- the Haskell plugin for the Eclipse IDE (Integrated Desktop Environment) is available here http://eclipsefp.github.com/
- Emacs packages and modules for Haskell are available here http://www.haskell.org/haskellwiki/Haskell_mode_for_Emacs
- Vim plugins are available here http://www.haskell.org/haskellwiki/Vim
- more info about other editors and IDEs is available here http://www.haskell.org/haskellwiki/Editors and here http://www.haskell.org/haskellwiki/IDEs

Speaking of editors. Emacs (http://www.gnu.org/software/emacs/) and Vim (http://www.vim.org/) require some time to learn. Coming from a 'Word'-like world, that might seem like a lot of work for a task that should require very little work if any: just start the editor and start typing. Not dealing with this extra complication is a perfectly fine choice for this course. But if you expect to spend a lot of time in your future career generating and editing technical text, be it research papers / reports in LaTeX, scripting web pages, online surveys or apps and / or writing any other kind of software, learning to use one of these editors will definitely be worth your while.

Both Emacs and Vim have built-in tutorials that can get you started, and there are plenty of resources on the web. Read some reviews online to make a choice or choose whatever your friends use. The choice is less important, what's more important is to stick with it for 2 weeks or so. Learn these editors like you learn something new in a medium-difficulty linguistics course, don't expect to just 'get it' and start typing. Alternatively, think of these editors as a new kind of fairly complicated computer game and play with them for a while (say, about 1 hour a day for 2 weeks). Don't try to do any serious work for the first few days (like writing a paper). Things will become much easier after several days and you will be productive after 1 week or so of learning / playing around with the editor. This might seem like a long time to spend learning a text editor, but chances are that later on you'll be very glad you did it.

2: Formal Semantics (Weeks 4-5)

Quick recap of chapters 2 and 3 of Dowty, Wall & Peters (1981), working through chapter 4 in detail and then introducing some extensions of the English fragment in chapter 4.

3: Implementing Formal Semantic Theories in Haskell (Weeks 6-9)

Implementing in Haskell: propositional logic, first-order predicate logic, the syntax and semantics of the English fragment in chapter 4 of Dowty, Wall & Peters (1981) plus various extensions (depending on how much time we have).

Here are some of the extensions: parsing a more substantial fragment of English with a more realistic grammar, providing a compositional semantics for the resulting syntactic structures, implementing a dynamic semantics system, accounting for quantifier scope ambiguities in several ways etc.

4: Presentation and discussion of the final projects (Week 10)

A couple of project ideas that involve mostly reading, understanding and (re)implementing previous research. The readings will be / are already available on Dropbox:

- implement part of Parsons (1990), *Events in the Semantics of English*
- understand monads and discuss Shan (2002) 'Monads for natural language semantics' and / or Unger (2012) 'Dynamic semantics as monadic computation'
- understand continuations and discuss Barker (2001) 'Introducing Continuations' and the relevant chapter in the *Computational Semantics* textbook by van Eijck & Unger
- discuss aspects of Unger's (2010) PhD dissertation *A computational approach to the syntax of displacement and the semantics of scope*

But you should feel free to pursue an original research project in formal semantics as long as you will provide a formally explicit account and start thinking about how to implement it. In this case, a correct formal account is required while a possibly partial implementation in Haskell is merely recommended.