In-Depth Research Seminar on Foundations for Computational Biology

Syllabus for a Research Seminar Course

by Laurence Loewe
Assistant Professor, Laboratory of Genetics and Wisconsin Institute for Discovery, UW-Madison

2 Credits, Spring 2016, Monday 3:30pm – 5:25pm
Room: 410B, WisCEL, 4th floor, Wendt Library
Genetics 677-Section22 | Class 46549 (Genetics), 46566 (Medical Genetics)

http://evolution.ws/loewe-lab/teaching

General Introduction

In a modern world dependent on computing, it is increasingly important to make programming languages more accessible and easier to use. Developing an easy-to-learn, easy-to-use language requires a deep consideration of word choices for developing a syntax that is easily understood by many diverse collaborators. This course aims to bring a varied group of people and perspectives together to help construct a new programming language that is ideally fun to use by people in the sciences, social sciences, humanities, and beyond.

While we often see programming languages as tools that tell computers what to do, it is easy to forget that these languages tell us what we have communicated to the computer. Seemingly trivial at first glance, the link between human minds and code is far more complex than the link between the code we write and what computers do. Getting computers to interpret code in the same way is much easier than getting humans to agree on the meaning of the code, especially if they do not read the manual. Thus, a formidable challenge in the design of a general language for biology is to get human researchers from many disciplines to agree on a syntax that is interpreted intuitively across fields in the same precise way. The usability of a language and its error messages depends on the quality of the underlying semantics and the regularity of its syntax, which must anticipate the diversity and unpredictability of how humans will interpret its code.

The discussions of this course will work towards simplifying an existing design for Evolvix, the (possibly) first general programming language designed by biologists for biologists. While a simple prototype of Evolvix exists already, the general programming extensions discussed in this course have not yet been implemented. We can nevertheless review what we want the language to do when implemented, and which language features would make it easier to write code for biological analyses. Pooling the expertise of students with very different skillsets in the sciences, social sciences, humanities, programming, mathematics, and writing will enable sharing our knowledge and perspectives and thus simultaneously much improve and simplify the development of Evolvix over the long-term. Along the way, you will pick up points of view that will help you identify where many programs go wrong and what might be done about it.

Who is this course for?

Everyone interested, because biology touches everything.

Participants can be at any level from freshmen to graduate, as long as they are happy to work with a broad range of experts from other fields. This course will accommodate and draw on diverse student interests and expertise; in fact, successful discussions will hinge on this diversity, which is represented in part by the instructor's trans-disciplinary research interests. For example, while **biology** students are vital participants (since Evolvix first aims to create models related to biology), students of **natural sciences** and **physical sciences** often easily find uses for a general programming language that mirror some uses in biology and could therefore help the group hone its questions and target key concerns for scientific modeling. Those from social science and humanities are similarly essential. **Social scientists** help us understand how modern societies work; their perspectives are important for engineering a system that people will want to use and for finding ways to deal with the social idiosyncrasies that prevent people from trying at all. They also have important expertise in designing questionnaires, which might hold the key towards unlocking secrets on what users of programming languages really think. **Humanities** students are trained to view the world from different angles, much like modelers must. By combining their mental flexibility with their substantial training in effective writing, they can help programmers develop code that is easier to read. **Linguists** and **philosophers** with a deeper understanding of logic, probability, syntax and the semantics of human languages are also likely to help very much. **Computer scientists**, with their substantial understanding of how programming languages, databases, operating systems, algorithms, and hardware systems work, can bring rigor and insightful experiences to the group discussions, and so can **mathematicians** and **statisticians** with an interest in abstract algebra, model theory, category theory, logic, numerics, geometry, probability theory, calculus, analysis, … as long as they have the patience to answer all the questions, which 'the rest of us' need to ask to get their insights.

No student *with a keen interest* should be excluded from contributing to this course on the grounds of belonging to the "wrong" discipline or having the "wrong" skills and experiences. Why? If a programming language is truly generic, then it has to be generic for everybody, despite discipline, level of experience, etc. If such a language aims to make programming easy for non-programmers, then it should be very simple to understand, otherwise it is bound to fail.

> ***Simplicity is to be highly valued in a standard.***
> ***Simplicity cannot be added later; you can only remove complexity.***
> ***Simplicity is the ultimate sophistication.***
> ***Simplicity is incredibly hard.***

Thus, from freshmen to professor, in this course *everybody* has a shot at contributing flashes of simplifying insights. *Everybody*'s input is required to help hold on to the clearest insights. *Everybody* will need patience to get through yet-to-be-simplified semantics.

So, to clarify who should and should not enroll in this course:

Prerequisites

Nothing formal – enrollment is open; you do not need instructor's consent[1].
This course is for you if you are thorough, reliable, and want to participate in the unique experience of helping to develop and review the semantics and syntax of a new general programming language alongside peers in distant fields.

To participate, you will be expected to sign a contributor's license agreement so that all your contributions remain usable for any type of open-source software that implements your ideas.

This course is ***not*** for students who

- want to learn how to use the latest computational tools for bioinformatics;
- want to learn any *particular* programming language to the level being 'able to program';
- feel uncomfortable discussing questions to which the answers are unknown;
- find it boring to think about the details of specific semantics and syntax;
- are troubled by looking for too long at too many negative outcomes, such as bugs, crashes, errors, warnings, contradictions, and all the rest that can go wrong in code.

What we will do

The form of the course is a series of in-depth research seminar discussions led by the instructor or by students on the topics listed at the end of this syllabus. These topics may spark other related questions that are worth pursuing; if so, participants of this course and the instructor will decide together whether such a question is better pursued or postponed. Specifically, we will focus on:

1. **Semantics.** Introduce, explain, discuss, review, and extend some of the formal foundations for a semantics capable of representing problems in computational biology, as currently proposed for implementation in Evolvix. Dealing efficiently and accurately with the enormous diversity and complexity of biology can often benefit from using lesser known abstractions developed in math and computer science.
2. **Syntax.** Introduce, explain, discuss, review, and simplify words and syntactic structures that constitute the syntax of Evolvix and which represent its semantics in the most straight-forward way (ideally readable even for K-12 students).

In other words: we will review, review, review. Repeatedly revisiting syntax and semantics from different angles, different disciplines, different levels of expertise from freshmen to professor, is the best way of following Einstein's interpretation of Occam's razor:

Everything should be as simple as possible, but not simpler.

[1] The official course catalogue for Spring 2016 states that you need instructors consent. This is *not* true; several people tried to change that, but the course catalogue was implemented using a programing language that apparently made this change prohibitively difficult – as if there were not already enough examples for such types of problems that this course is about ...

In addition to learning how to simplify and how to use the abstract topics of fundamental importance to computational biology that are discussed in the course, students can participate in several overlapping ways:

A. Act as an '**expert in your field'**, help our multi-disciplinary team understand your perspective, and produce corresponding essays investigating detailed aspects of the state-of-the-art in your field. These may inform the semantics of Evolvix.

B. Act as a '**usability reviewer'** and assess how easy or difficult it is to understand the explanations given by experts and the syntax they propose. You will honestly review and communicate what you did or did not understand while participating in discussions with the goal of unambiguously describing key concepts.

C. Act as a '**common-sense debugger'**, ideally, by detecting all ambiguities in any syntax variants proposed for implementation. You do this by reading examples differently from everybody else (but justifiably so, based on English grammar). Even if we cannot easily fix an ambiguity you found, you will have made a valuable contribution by highlighting a problem with the group's perceived 'common sense', which obviously did not include you. This probably indicates that many others share your view, proving that the semantics of such syntax is no longer '*common* sense'. Such language proposals will require special attention, lest they cause many programming headaches.

D. Act as a '**language designer'** by proposing alternative word choices that are better, shorter, convey a specific semantics faster, clearer, with less intuitive ambiguity, understandable at a younger age, and require less learning. This process is necessarily iterative and may require revisiting previous results once it becomes clear that the 'nice abbreviation' in one field is in conflict with the much 'nicer abbreviation' in another field. Even if you don't find "the final syntax", any new iteration or pros and cons you can contribute gets us closer to a language that makes it easy to reliably reproduce a given understanding accurately in the minds of others.

Reproducibility: the reason why we do what we do

Computational support for accurate, efficient, and user-friendly methods for automated analysis in computational biology requires a programming language that:

- is based on a formally complete and contradiction-free semantics;
- provides maximal control over the applicability of underpinning assumptions;
- sets user-friendly defaults for the most common-use cases to reduce code length;
- chooses commonly understood words that convey the correct meaning to most users.

Implementing programming language has become a routine job for those who know *how* to do it, as demonstrated by the thousands of languages that exist. However, the question of *what* precisely should be implemented is far from trivial and often a difficult art that depends on the precision with which the designer(s) of a language observe the world. That is where your unique perspective could contribute essential insights if a language designer listens to you.

Many programming languages are made inadequate for many problems at very early stages because their designers are often forced to 'fly blind' and make decisions that they consider to be simple without truly understanding their implications. At this stage, language designers often do not yet have a working version of their language that allows them to test their ideas. They also might lack the expertise or necessary awareness, and impending deadlines might demand a quick release of their product. These and similar reasons often push them to make foundational implementation decisions that can cut entire problem domains from efficient computation in a language. Many of these decisions are so foundational that they cannot realistically be changed later without completely changing the language. Mistakes at this stage can be extremely costly and may inflict huge pain on the users of a language – as the 'Billion Dollar Mistake' by Tony Hoare has shown (Google it!). In computational biology, such mistakes hamper the reproducibility of computational analyses worth about $7 Billion in pre-clinical biomedical research in the US each year [1]. They can also make it impossible to explore entire fields of investigation.

This is a huge problem and standard approaches for language development have not yet resulted in satisfactory solutions that are expressive enough to describe what biologists encounter on a daily basis: diversity, uncertainty, gaps of knowledge, complex exceptions, …

These difficulties motivate the use of an innovatively different approach for developing Evolvix: since *how* to implement is much less a problem than *what* to implement, there is little point in jumping the gun and implementing quickly just to have a working language[2] while wasting the valuable opportunity to revisit its fundamental assumptions, strengthen its foundations, and ensure excellent support for biological problems. After developing a rough draft, the leading language designer (LL) decided that it would be worth reviewing what to implement from many different perspectives before many decisions are locked in by writing actual code. This strategy prevents much unnecessary complexity that emerges when promising versions break after being confronted with a different perspective that was readily available before implementation. The central contribution of this course is to draw on as many perspectives and skillsets as possible to review and refine the language's semantics and syntax to simplify coding, simplify extending, simplify learning, and simplify modeling in computational biology.

[2] An early prototype of Evolvix has been implemented already; it is very good and efficient at simulating the deterministic and stochastic mass-action systems biology that it allows users to describe (freely available at http://evolvix.org). However, it is far from enabling efficient *general* descriptions of biological diversity and its evolution. In-depth analyses have shown that the most efficient way to add these capabilities to Evolvix is to combine it with *general* programming functionality. Such implementations exist too, as many different *general* programming languages have been implemented in the last decades, some more widely used than others; hence, no need exists for "yet another *general* language". Evolvix could be easily extended by adding such general languages to the system, an idea that was tested while conducting a simulation study (Ehlert & Loewe 2014)[2]. However, the lack of integration and the many idiosyncrasies of typical programming languages combined with the lack of support for various specific recurring needs in biology make this an unnecessarily complex proposition. While there is no shortage of domain-specific *specialist* programming languages designed by biologists (e.g. see http://sbml.org), these typically only work for biologists interested in problems very similar to those targeted by the language designer (similar to the current Evolvix prototype). LL has learned enough about *how* to implement programming languages to make it possible; LL has worked for 15+ years in computational biology research, learning *what* a language needs to do, and he spent about 2 years integrating and distilling everything into a draft syntax and semantics to be reviewed in this course for simplicity, accuracy and appropriate expressiveness.

Why Take This Course?

Have you ever wished a programming language would serve your needs better?

This course is your chance to make yourself heard!

It offers a unique opportunity to impact the long-term syntax of a programming language that will eventually be released as an open-source project for use in research, teaching, and beyond – wherever computing has to be reliable. In this course, you can interact directly with a language designer who invests the time to find out what is worth implementing by asking potential users from beginners to experts.

It is unlikely that you will find a similar opportunity elsewhere at UW-Madison and your chances of ever encountering a language-development project that brings together such a diverse group of disciplines is even slimmer. If you want to improve something about programming languages, from ease-of-use to functionality, then this may be your chance.

While developing the current design for Evolvix, the instructor encountered so many diverse and surprisingly useful non-standard concepts that you might be interested in taking this course for no other reason than to review, expand, and strengthen your understanding of sets, types, probabilities, concurrency, different types of logic, calculus, and more.

You will learn by investigating questions important for modeling, concurrency, logic, numeric quantification, probability, uncertainty, semantics, documentation, data organization, with a view to computational biology, genetics, big data, and other fields in computational science and engineering. We will explore new topics or new perspectives on old ones. Some of this work is very much at the cutting-edge of programming language development and research that none of us will have all of the answers (including the professor). We will all benefit from learning how group members apply different disciplinary approaches to solve new challenges.

This course also strongly contributes towards developing general skills such as critical thinking, creative problem solving, and the trans-disciplinary skill of learning how to navigate previously untraversed fields while developing a consciousness for the depth of one's own ignorance.

Specifically, this course will help students

- see much more clearly where computational analyses often fail, paying particularly attention to the most difficult cases (those that go wrong without producing errors);
- develop a common language for discussing said problems, which are often ignored precisely because no such language exists;
- gain exposure to a broader diversity of formalisms than taught on standard curricula, which helps when choosing the right tool or formal approach for a given analysis;
- understand some of the epistemological foundations of science, which draw principled, impenetrable boundaries around what is knowable by any scientific method;
- practice some aspects of developing a computational semantics;
- translate challenging concepts into every-day language without falling prey to communicating the wrong notions.

Progress in this course is not measured in topics covered, but rather in problems found and solved; to increase support for reproducibility, it is more important in this course to do a good job covering a fundamental topic properly than to rush towards applications with poorly defined semantics. The goal of this course is to detect and avoid errors that can be eliminated automatically or by definition, similar to the "Billion Dollar Error" of Tony Hoare. The instructor has extensive experience in leading such open discussions with beginner and expert participants from very diverse disciplines; these have demonstrated that this format is highly efficient in detecting open semantic gaps, inconsistences, and other problems, as well as solutions and simpler forms of syntax. These precise activities also sharpen your analytical skills.

Coursework

All texts you write in this course have to be submitted under a contributor's license agreement that gives the instructor the permission to re-submit or use your text in any public, open-source release of Evolvix if it passes all corresponding reviews. You still keep your copyright; you merely provide an irrevocable, transferrable, world-wide, royalty-free, … license for the instructor to include your contributions to the course in Evolvix if they pass the Evolvix review process. Those contributions that do not pass Evolvix review will not be published unless discussed explicitly in the course.

In Class: After the first two introductory seminars led by the instructor, a number of conceptual starting points for discussion will be provided in the diverse areas of the course (see potential list of topics at the end). These discussion starters can be provided by the

- **Instructor,** either giving a broad overview of the syntax and semantics of some concepts in Evolvix, or discussing details of the language design deemed worth considering, maybe in response to previous discussions;
- **Presenting Students,** who have signed up either for covering an area or a problem proposed by the instructor or by students. Students then either present their own background research (e.g. find the best introductions and reviews of x) or build on literature given by the instructor.

During Week: Students will use the topics of the class meeting as starting points for their own investigations during the week. These include

- **Write** 1 page of reflections on the topic of the class or some other topic deemed relevant for foundational aspects of computational genetics or biology;
- **Improve or use** a feedback form aiming to capture ideas on how to improve modeling ease and accuracy by improving some proposed syntax and/or semantics either for beginners or for experts or by comparing it to the possibilities and limitations of (an)other modeling system(s).
- **Read** the next paper to be discussed (if reading was recommended).

During Semester: Students will select topics of the class as starting points for:

- **Write one longer text** that covers a more substantial aspect of semantics and syntax in an area of your choosing (or as assigned by the instructor).
- **Present at least once** a mini-overview or similar presentation that you prepared before the class (in addition to constant participation in all discussions).
- **Conduct at least one mini survey** on the usability of some aspect of Evolvix by using the latest adaptation of the interview form developed in the course; talk to at least 5 people of your choosing.

Constant focus questions:
All contributions shall aim to improve the:

1. Accuracy of mapping to the real world;
2. Ease of constructing accurate models;
3. Readability of code, documentation, models, and error messages;
4. Brevity while maintaining clarity, regularity and consistency of syntax;
5. Expressivity of overall semantics;
6. Matching defined semantics with real-world observations;
7. Opportunities for improving ease and quality feedback.

Grading in general

In-class participation (interactive questions are *always* welcome!) **40%**
Always ask questions, especially if something is not absolutely clear to you. All presenters shall aim to simplify enough so everybody can follow. Extra kudos for trying to ask super hard or super simple questions that might lead to improvements in syntax, semantics, or possibly even the overall design of Evolvix.

Weekly Reflection Papers (1 page) **20%**
Any relevant topic related to programming or modeling in biology, genetics or otherwise. In the absence of other instructions or relevant ideas, reflect on the constant focus questions above.

Feedback Collection (organize 1 survey, participate weekly): **10%**
The precise forms are open to discussion and can be modified as needed.

In-Class Presentation (at least once): **10%**
Present some substantial paper review or critique in class; can be related to other contributions or your project. The form is up to you.

Project Paper(s) (x pages): **20%**
Accumulate at least x pages worth of more organized and edited quality contributions, which can treat a larger topic in a single paper or several small ones. Graduate students are expected to write twice the amount of pages.

Differences in Grading Undergraduate and Graduate Students

Generally, more mature contributions will be expected from graduate students, especially in their area of expertise. Complex and well-presented figures and tables that need substantial effort to compile count as a 1000 words.

Undergraduate Project Paper(s) (>8 pages, > 4000 words):
Accumulate at least 8 pages worth of more organized and edited quality contributions (over 4000 words total); can be in a single paper or in up to 4 small ones.

Graduate Project Paper(s) (>16 pages, > 8000 words):
Accumulate at least 16 pages worth of more organized and edited quality contributions (over 8000 words total); can be in a single paper or in up to 8 small ones.

Trans-disciplinarity and Course Topics

It should be clear by now that this course is likely to be more productive for a diverse set of students from the sciences, social sciences, and humanities. However, the instructor understands that the course's aims in combination with the topics listed below may be intimidating for students who bravely consider stepping outside the comfort of their home discipline. *You should not let these topics or terminology deter you from taking this course.* This course does not require any prior experience in maths, biological or other sciences, programming, or languages of any kind development. Using his extensive experience in communication across disciplines, the instructor will mediate the dialogue based on the background of the participants, clarify the meaning of technical terms, and foster "common-speech" translations of technical jargon to enable computational and non-computational students alike to follow along and participate regularly.

Don't worry if you don't understand all words in course topics right now: we will work through this together and all basic "beginner's questions" will be welcome
(*in fact, the point of the course is to hear which questions you will come up with*!).

If you are ahead of all others in the course on a particular topic (because it's your field), then you can write down what you have learned and enlighten the rest of us too. It turns out that such teaching enables very effective learning. Alternately, if you feel like you are doing more learning than teaching, then that is fine too – despite substantial insights into the course topics below, the instructor also expects to learn much from the diverse pool of expertise represented in this course. We will all be teachers and students at the same time in this course.

Since students in the course may be asked to vote on whether important new topics are addressed on the spot when discovered or later in a separate discussion. The following topics list is not exhaustive and probably provides more discussion starters than there is time in the semester.

Incomplete Topics list (unordered except session 1-2; Evolvix *Concept Words* capitalized):

1. **Overview**. Why yet another programming language?
   What do we get out of a course that discusses a language that does not yet exist?
   Why review a language in an interdisciplinary group before implementation?
   How can you contribute while learning about foundations for computational biology?
   Language design goals. Contributors License Agreement.

2. **High-level motivating example:** simulating biology from biochemistry to ecology by defining the parts and actions in a *Mass Action Algebra* that represents an underlying *Continuous Time Markov Chain* (CTMC) *Model*. [3-13]
3. *Sets* and *Types* are complementary notions that depend on *Context*
4. Mass *Action* and concurrency from biochemistry to ecology (CTMCs)
5. *Logical Operators* for diverse *Types*
6. *Quality Assessment*, *Stability* and editing formalisms (used in the course)
7. *Oftenness* a new concept integrating all *Types* of *Probabilities* and *Frequencies*
8. *Probability* interpreted as rational belief (Logical theory)
9. *Probability* interpreted as the degree of belief of an individual, as humans may not share the evidence or have different opinions on how to interpret it (Subjective theory)
10. *Probability* interpreted as limiting frequencies of infinite test series (Frequency theory)
11. *Probability* interpreted as inherent propensity (Propensity theory)
12. *Frequencies, Dice* and *Distributions; NoDice, OneDice, MultiDice, InfiniDice*
13. Foundations for *Closeness*; *Usual*, *Other*, *Boundary*, *Rest*; different *Types* of *Boundaries*
14. *Chains* for *Comparisons* across different *Types* of *Dimensions*
15. *Comparison Operators*
16. *Numbers* of different *Types*
17. Semantics of *Zero* (Nothing)
18. Semantics of *Infinity* (Everything)
19. *Intervals*, *Ranges, Domains;*
20. Abstract *Algebra* of *Operators; Arithmetic Expressions;*
21. Classical *Arithmetic* and *Linear Calculus, Change over, Integration*
22. *Exponential Calculus* and biological growth
23. Generalized *NonLinear Calculus* and the computation of Averages
24. *Computational Concurrency* Compared to *Mass Action Concurrency*
25. *Assignments, Sets, Memory*, *Borders*, *Walls*, *Bridges*, and *OKO* Semantics
26. A Turing machine with *Assignments*, *Conditions* and *Loops*.
27. Nesting *Constructs*
28. Multi-dimensional *Arrays*, Spaces, and Landscapes.
29. Machines overview (Context Free Grammars, Regular Expressions, and more)

Goal: Exploring the big questions, we may find an answer, or not, or maybe the answer is 42.

References

Here are some starting points for relevant literature; most of the relevant papers will be shared in class.

1 Freedman, L. P., Cockburn, I. M. & Simcoe, T. S. The Economics of Reproducibility in Preclinical Research. *PLoS Biol* **13**, e1002165, doi:10.1371/journal.pbio.1002165 (2015).

2 Ehlert, K. & Loewe, L. Lazy Updating of hubs can enable more realistic models by speeding up stochastic simulations. *J Chem Phys* **141**, 204109, doi:10.1063/1.4901114 (2014).

3 Gillespie, D. T. Exact Stochastic Simulation of Coupled Chemical-Reactions. *Journal of Physical Chemistry* **81**, 2340-2361, doi:Doi 10.1021/J100540a008 (1977).

4 Gillespie, D. T. *Markov processes : an introduction for physical scientists*. (Academic Press, 1992).

5 Gillespie, D. T. Stochastic simulation of chemical kinetics. *Annu Rev Phys Chem* **58**, 35-55 (2007).

6 Daigle, B. J., Roh, M. K., Gillespie, D. T. & Petzold, L. R. Automated estimation of rare event probabilities in biochemical systems. *Journal of Chemical Physics* **134**, 044110, Doi 10.1063/1.3522769 (2011).

7 Gillespie, D. T., Hellander, A. & Petzold, L. R. Perspective: Stochastic algorithms for chemical kinetics. *Journal of Chemical Physics* **138**, 170901, doi:10.1063/1.4801941 (2013).

8 Klipp, E. *et al. Systems Biology: A Textbook*. (Wiley-Blackwell-VCH, 2009).

9 Faeder, J. R., Blinov, M. L. & Hlavacek, W. S. Rule-based modeling of biochemical systems with BioNetGen. *Methods Mol Biol* **500**, 113-167, doi:10.1007/978-1-59745-525-1_5 (2009).

10 Sneddon, M. W., Faeder, J. R. & Emonet, T. Efficient modeling, simulation and coarse-graining of biological complexity with NFsim. *Nat Methods* **8**, 177-183, doi:10.1038/nmeth.1546 (2011).

11 Anderson, D. F. A modified next reaction method for simulating chemical systems with time dependent propensities and delays. *J Chem Phys* **127**, 214107, doi:10.1063/1.2799998 (2007).

12 Anderson, D. F. Incorporating postleap checks in tau-leaping. *J Chem Phys* **128**, 054103, doi:10.1063/1.2819665 (2008).

13 Anderson, D. F., Enciso, G. A. & Johnston, M. D. Stochastic analysis of biochemical reaction networks with absolute concentration robustness. *J R Soc Interface* **11**, 20130943, doi:10.1098/rsif.2013.0943 (2014).