1 Project Overview

In addition to reading about languages other people have designed, in this class you will design and build your own Embedded Domain-Specific Language (EDSL) in Haskell using Haskell’s quasi-quotiation library. This project has a number of parts:

- In the first part (described in more detail below), you will write a design document describing and motivating your language. **Due October 15.**

- In the second part, you will design a syntax for the language documented as a grammar, define an abstract syntax tree data structure to represent your program texts, and write a parser to map from program texts to abstract syntax trees. **Tentatively due October 29.**

- In the third part, you will write a translator that maps abstract syntax trees for your language into data structures representing Haskell code. You will then connect these pieces together using Haskell’s quasi-quotiation library. The quasi-quotiation library will allow end users to write programs in your language and have the Haskell compiler invoke your parser and translator to generate, compile, and execute the Haskell code providing the implementation for your language. You will be able to write any necessary “run-time” system code as a Haskell library that links with the generated code. Note that the Hackage web site ([http://hackage.haskell.org](http://hackage.haskell.org)) has a large number of Haskell libraries that you can use in your implementation. The quasi-quotiation framework is quite flexible in that you can use it to generate multiple interpretations for programs in your language (like the parser and pretty printing semantics for PADS). **Tentatively due November 19.**

- In the last week of class, you will present your language design to the class, explaining your design decisions and evaluating your design using the same criteria we’ve been applying to the DSL papers we’ve been reading for class. **Tentatively due December 1 and 3.**

- Finally, in lieu of a final exam, you will turn in a paper explaining your design and evaluating it. **Tentatively due December 8.**

You may choose to work in groups on this project. The scope of the project should be commensurate with the size of the team. Project assignments should be submitted and will be evaluated as a team. (That is, the entire team should submit only one response to each of the project parts and the entire team will receive the same grade.) Students in COMP 150PP may chose to do a joint project as long as it satisfies the project requirements of both courses. If you have any questions, please feel free to ask for clarification or input.
2 Design Document

For this first part of the project, you need to write a document explaining and motivating your proposed design. Careful thought in this phase will make the later parts of the project dramatically easier, so don’t leave this assignment to the last minute. Your design document should address at least the following points:

1. What is the domain of your language?

2. Who are the intended users? What skills do they have and what kind of programming tasks do they need to accomplish?

3. What goals are you trying to accomplish with your design? (For example, the designers of ESP wanted to make it easier to develop correct, modular code for firmware while not paying too much of a performance cost.)

4. Why are existing languages ill-suited to the task?

5. Describe what features your language will provide and give some rationale for why this set of features is appropriate.

6. Explain what if any services you will need to provide in a “run-time system.”

7. What existing Haskell libraries, if any, are you planning to use to support the domain-specific aspects of your implementation? You don’t need to mention quasi-quotation, template Haskell, or parsing libraries.

8. Explain what benefits domain experts will get from using your language. For example, do they get multiple artifacts from a single program? Can they write substantially less code? Is the code more likely to be correct or efficient? Can specialized analyses be applied?

9. Develop three to four use cases, each of which specifies
   (a) The specific task the user is trying to accomplish
   (b) The information the user would have to supply, perhaps in the form of preliminary syntax for your language
   (c) A high-level description of the code you intend to generate from the supplied information. If you plan to generate multiple artifacts from a single description, you should describe each of them.
   (d) A brief discussion of why you will have sufficient information to generate the desired implementation from the user-supplied information.

10. Sketch a plan for evaluating your design. This plan should include measurable goals that if achieved would lead you to conclude your design was a success. (Example goals might be code-size reductions or measured performance overheads, but many other goals are possible).

A Final Note. It is easy in projects such as these to try to do too much. While having grand visions is a good thing, trying to materialize the entire vision at once is often impossible. I encourage you to think about designing your language so you can build it in levels. For example, a good design might allow you to implement the features for your first use case relatively quickly before moving on to the additional features necessary for the second use case, etc.