Domain specific languages allow us to easily express problems in certain areas in the right way. Though they might make it harder to solve some problems, they allow us to optomize for a particular class of problems at hand. Using tools such as Bison, Antlr, or parser combinators, it is pretty easy to spin up your own language. So, we are left with the question, what prevents domain specific langauges from catching on? This is the question that I want answer and engineers solutions to in graduate school. Graduate school is the right place for me to work on this problem because such a program allows for longer incentive cycles: instead of having to get a product out by the end of the quarter, I can spend a while on a project, and I can develop and try many answers to it. Additionally, for this particular research focus, it will greatly aid me to listen to the kinds of problems that my colleagues in a wide variety of focuses are working on. In this personal statement I will outline four research lines that either try to find out why DSLs have trouble catching on, or provide solutions to known problems with DSLs. They all have a general theme: we have these really powerful compilers that generate code, can we use them to generate other parts of the development cycle.

We have really powerful compilers that generate code, can we use them to generate code profilers and debuggers? When we right code we should care about (among other things) readability, preformance, and robustness. Domain specific languages already provide us (hopefully) with very nice readability, But, often it can be hard to drag developers away from their shiny tools for C++ and Java. So, to solve th problem, how about we automatically create code profilers and debuggers from the parser description. To a first approximation, we can use our DSL compiler to attach tags to our generated code language (perhaps C), and then use the standard C debug flag to generate tags into machine code, do our debugging and then backthrough the C tags and then form those back through the DSL tags. But, this leaves out a couple of problems, such as when the DSL generates part of the program as some data (perhaps a full AST) for the C code to then run over. We could easily imagine this if we're running a Lisp interpreter, or some sort of database of Prolog clauses. Perhaps we can do some tracking of loads from the data section as tagging? I'm not sure, but this leaves open an interesting questions that I think would be useful to pursue further.

We have really powerful compilers that generate code, can we use them to generate static analyzers? Static analyzeres are powerful, and getting more still; this is especially the case we can use them to start synthesizing parts of our program: perhaps trying to super-optimize that kernel at the core of your program, or to generate that tree operation that you don't want to mess up. It would be really grerat if we could simply import some static analysis into our language. Take importing a type checker as an example, we can think of system where we just have a database of unification statements as a C library, and as we parse code we add the proper information into that database. But it does not seem obvious what is the best way to map type unification failures back to source in a general manner, and what if we want to do type-based-synthesis: can we use the parser definition to power that synthesis? There are a lot of

interesting questions about how to best tie a type checker to the source code, and even more interesting questions about other types of static analysis (can we do plug and play race-condition analysis?). Surely, this will provide interesting paths for research.

We have really powerful compilers that generate code, can we use them to figure what problems users have with our languages? An important part to any user facing design (which languages are obviously), is testing with real live users. What if we had a system that could automatically log where users had problems with our languages. If a user has trouble understanding some feature of a type system, how the import system works, or even just the syntax of our language, it would be great to be able to record that information in a way useful for later analysis. A simple first step would be to just record where users pass in syntactically incorrect programs: recording the rule in our compiler that could not find a parse, perhaps indicating a friction in documentation or just in the language design itself. Languages like Rust have pretty useful helpers for understand that language's linear type system, perhaps with the previous step we could provide helpers for type systems for arbitrary languages as well. This will help provide further areas of research, as it will help language designers understand why users are having trouble with their language.

We have really powerful compilers that generate code, can we use them to generate direct-manipulation systems for our languages? Direct-manipulation allows us to combine the percision of text source code, with programmer intuitions of what they think the result should be. Examples of these system include automatically generating SVG drawing code from a users' drawing, and changing parts of a users code to meet their desired values for particular variables. The trick for these systems is to be able to convert semantics back into syntax, which would mean that to this generally, we would need a way to automatically reverse any given parser. This is probably the most difficult challenge, but one could imagine using this to help fix tricky synchronization mechanisms by changing what the state should be, or changing a query in a custom database DSL such that it gives a result that the user is expecting.

My goal is to make domain specific languages more widely available. There are a lot of success stories of DSLs allowing users to write readable, performant, and robust code. In short they help reduce the turn around of programmer idea to prograemmer solution. But, they can often get meyered in generating the necessary acoutramont that allows developers to use such languages on a daily basis. I am not sure what is to best way to make DSLs widely used, but that is what I in part hope to answer. Through my research at UNIVERSITY, I would determine whether it is better to continue in academia or prostelize DSLs in industry. Either way, general purpose languages must be slightly substituted by DSLs in particular contexts.