Independent Study Final Report

Joshua Sellers
Rochester Institute of Technology
Department of Computer Science
## Contents

1 Introduction

2 Papers

2.1 Creating Languages in Racket

2.2 Building Domain-Specific Embedded Languages

2.3 POP-PL: A Patient-Oriented Prescription Programming Language

2.4 DSL Engineering 1

2.5 DSL Engineering 2

2.6 DSL Engineering 3

2.7 The Spofax Language Workbench

2.8 The RASH JavaScript Editor (RAJE)

2.9 Modeling HTML in Haskell

3 Projects

3.1 CardGameDSL

3.2 AccountingDSL

3.3 Final Project

4 Conclusion

5 Appendix

5.1 Work Completed

5.2 Explanation of Deviations
1 Introduction

For this course I explored the creation of computer science languages. This was a broad topic, so I narrowed it down to a more specific area. Based on my discussion with Professor Fluet, I chose to explore domain-specific languages (DSLs). This technique, also referred to as Language-Oriented Programming, is where a programmer eschews a general-purpose language for a problem and instead creates a domain-specific language (usually using some existing resources, like existing general-purpose languages, for tools).

In order to gain a better understanding of DSLs, I read through numerous research papers on the topic. The papers covered DSL creation in Spoofax, Racket and Haskell (three languages that are well suited, whether intentionally or coincidentally, for being used as the back end of a DSL). After reading the papers, I then completed two shorter projects using the knowledge gained from reading. I used Haskell to create a DSL that would allow users to specify their own rules for card games and I used Racket to create a DSL that simulated the first few steps of the accounting life-cycle. Finally, I completed a, larger, final project (in Haskell). My final project was a DSL that allows coders to simulate low-level hardware (like AND gates and DECODERS).

Enclosed in this report is my write-ups for each of the papers that I read, the summaries for my three projects and an appendix of my schedule (that also covers any deviations from the original one in my independent study proposal).
2 Papers

Here are copies of the write-ups written for each research paper read for this course (each one includes the date that it was submitted).

2.1 Creating Languages in Racket

21 January 2019

Creating Languages in Racket

Matthew Flatt

For my first paper, I chose an introductory one to ease myself into the topic of Domain Specific Languages. I picked this paper both for its simplified nature and the language it covers (I plan on using Racket for one of my projects in this class).

As Flatt stated in the paper, “A Racket program can contain definitions that extend the syntax of the language for use later in the same program, and language extensions can be packaged as modules for use in multiple programs.” Additionally, the extension can even go as far as creating a new language entirely. For the paper, Flatt used a text-based adventure game to introduce how Racket can be utilized (he points out that this is not a trivial example, since a Racket-based language was used to implement content for the video-game series Uncharted). The game was an effective demonstration of the uses of Racket because its construction poses issues for general-purpose coding languages (e.g. how the code would handle user input, which comes in text for these types of games). While overloading and laziness in some languages could be used to circumvent the issues brought up by Flatt, Racket is a better and more direct option. It has a full scope that supports both small extensions to itself and transformations into new languages. This scope was covered by Flatt through the use of his text-based adventure game.

Before I go over what he showed, it is necessary to describe the game he created as an example. Text-adventure games involve the user providing text inputs to interact with a game-world. Flatt broke up the components of this type of game into verbs, things and places (for instance, a room is a place containing things and is south of another room, which you could travel to using a verb). The textual input and variety of interactions a user can have with the environment are what makes Racket a better option than a general-purpose language.
Flatt used the game example to first show a slight abstraction of Racket and later went over how a more-complete extension could be accomplished.

Racket is based on Lisp; specifically, Lisp’s descendent Scheme. This means that Racket uses parenthetical structures as its basis. Some other important facts about it are that semicolons start newline-terminated comments and #:mutable after a field makes that field mutable. Overall, the basic structure of Racket seemed fairly similar to that of Scheme. For the game itself, Flatt extended Racket by declaring structs for verbs, things and places. Then, using these structs, he used define to create the various aspects of the game. Finally, he used Racket-specific procedures around hash tables to define the storing and recording of names. As you can see, Racket uses a symbol preceded by a single quote as an alias for what the user sees. Additionally, Flatt used the cons function to pair a verb “south” with a place “desert” in order to show that the user can go south from the meadow to reach the desert (using the !record-element function for mapping between names and places).

The next thing shown by Flatt was how to use syntactic abstraction to further simplify the use for the new additions to the language. Specifically, he described how patter-based macros could be used. The one he covered was define-syntax-rule, which allows the coder to give a pattern and a template (the ellipses allow for more than just one thing, verb or expression to be matched to the place). This would allow a future coder to
simply define a new item in the game world using the template instead of rewriting all of the lines of Racket code. Finally, Flatt went into how to apply this concept to verbs. This was more complicated than places or things, since verbs have any number of aliases and have to be collected into a list for the game’s parser.

For the sake of space, I will simply include the code he used to make verbs work. These concepts encompassed what a coder would do if they just wanted to abstract Racket for a single game. However, if they wanted to make their work reusable, they would have to go into syntactic extension; exporting their procedures and hiding the Racket-specific code that they contained.

In the section on syntactic extension, and those that followed, Flatt went into how to extend some Racket code into a world-defining module where macros like define-place had the same weight as built-in procedures like define. The way you do this is through the use of (require “code.rkt”), which is basically Racket’s version of import. Additionally, if you want to indicate that some file is in the Racket language, start your file with #lang racket. Additionally, Flatt discussed how one could treat the extension-code as a language by replacing #lang racket with #lang s-exp “code.rkt” (this indicates the notation of this file’s code will be S-expression). These were the main points, but he also covered how to export everything from racket from “code.rkt” and add in type-checking to the created procedures. The final idea introduced was how to create new syntax by using the line #lang reader “code.rkt.” The reader is something that parses the new syntax into an S-expression.

Overall, this was an interesting, if broad, introduction to Racket. It did an excellent job of introducing Racket to a novice reader like me (though I had to reread it a couple times to really understand everything). I plan on using it as a starting point when I begin my Racket project.
2.2 Building Domain-Specific Embedded Languages

24 January 2019

Building Domain-Specific Embedded Languages

Paul Hudak

This paper covers the topic of embedding a new programming language in an existing infrastructure (creating a domain-specific embedded language, or DSEL). It is a position paper, so there are some examples, but it is mainly Paul Hudak’s proposal on this topic (with references to other works).

First and foremost, before I go into Hudak’s idea for Domain Specific Languages, I should note my thoughts about this paper. Because the paper was more of a proposal than a concrete example of a tested idea, I found it hard to grasp some of the concepts; I understood his overall point, just not some specific details. As such, I may go back and reread this later in the course to see if it makes more sense to me then. Additionally, the pertinent information in this paper is not extensive, due to the nature of the paper, so this writeup will be a shorter one.

In Paul Hudak’s words, “A domain-specific language is the ‘ultimate abstraction.’” What he means is that a domain specific language (DSL) is the ultimate way to tackle an application or problem. He goes on to counter the notion that a DSL, while “ultimate,” is impractical. He proposes embedding a DSL in an existing language’s infrastructure, which is basically how languages like Racket are used. The language he uses for the embedding is Haskell (he gives several existing examples of how Haskell has already been used for this).

This big benefit that Hudak puts forward for using Haskell to create a DSEL is that it can be modular i.e. isolated into its essential parts. This is because, as Hudak explains, a DSEL in Haskell is like a higher-order algebraic structure. It would allow new additions to the language to be added, without changing existing code, after construction of the language was already complete. He accomplishes this through the use of Monads, which he shows can be configured to users’ specifications. This modular framework is shown in his diagram.
for creating a computation-solver on the previous page.

In the final section of the paper, Hudak delves into partial optimization on the interpreters for the DSELs. This was the section I had the most trouble following. It seems like he is proposing a user-input driven approach to how people make these modular interpreters, whether with respect to the task at hand or with respect to the idea of creating these task optimizers (like Racket). However, I would feel more comfortable with my comprehension of this section after running it by my advisor, Professor Fluet.

The premise of this paper did not shift my perception of DSLs that much, but it did give me a better idea of how Haskell could be used to make them more efficiently. I think that reading a paper that covers Haskell’s use in this area in a more granular fashion would be helpful.

### 2.3 POP-PL: A Patient-Oriented Prescription Programming Language

31 January 2019

**POP-PL: A Patient-Oriented Prescription Programming Language**

Florence et. al.

In POP-PL, the authors look into how to improve workflow in hospitals. Specifically, they attempt to streamline prescription writing by creating a Domain Specific Language (DSL) just for that task. Their language is an extension of Racket, which I looked at in one of last week’s papers. They do not go into the coding as much, but this paper gives another good look into using Racket. One final thing to note before going into the contents of the paper is that I actual looked at two version of this work: one was a workshop paper and the other was the full journal version. While I focused on the workshop version, I also read the journal paper and they were fairly similar (though the journal one, obviously, was a bit more in-depth).

The authors do a good job of explaining why their DSL is important for medical professionals. The current system for writing a prescription, at least at the time this paper was written, involves flowcharts, tables or other manual methods. These approaches raise concerns because medical errors account for one sixth of all deaths in America each year. Therefore, turning the process for writing out a patient’s plan for recovery, which can be quite complicated, into a well-defined language is a good area to reduce human-error. There are four areas of concern with prescriptions that the authors want their language to either solve or reduce in their impact. The first is “forgotten
monitoring;” forgetting or not being told to monitor the patient. Next, there is “alarm fatigue,” where the number of alarms that a patient causes makes hospital workers less attentive to important ones. Then there is “delayed reaction,” which is self-explanatory (the paper gives the example of a delay that occurs when a nurse is not told what antidote to give a patient and has to go ask the doctor). The final issue is “task overload,” the paper says that 98% of nurses had to drop at least one task in the span of a week. The authors mitigated these issues through their language design.

This is the authors’ description of their language:

“To formally model how POP-PL behaves, we define an evaluator that handles one new message at a time. This evaluator takes the current history of the hospital (including the new message), and an actor, which is nothing more than a list of handlers. These handlers can send messages to the outside world, add new handlers to the actor, or remove existing ones. This produces a set of outgoing messages and a new set of handlers for the actor.”

This structure allows users to encode messages for the system, which are then evaluated. These messages can also be queried in order to compare them to previous inputs to the system (like if a doctor wants to check if a patient has had multiple test results of similar values). An example of the language in action can be seen to the right.

Beyond example code, the authors also delve into the model syntax and evaluation of their language (the model syntax and an incomplete version of the evaluation are shown to the right). This shows how the messages, actors and handlers interact with their model. This structure is
useful for me, because it has given me direction on how I might go about making my own language in Racket for my project. Before, I had good ideas for what problems I wanted to solve with my language, but no real idea of how the language would actually be structured.

The final section of the paper delves into how the authors tested the effectiveness of their work. They gave an encoded prescription to a group of 51 medical professionals and a set of questions on the code to test their comprehension of the new language (the code was shown in the journal version of the paper, but not the workshop paper). Their conclusion is that anyone with coding knowledge is able to easily use and understand their DSL, but that people without that background will have trouble with more computer-science related things like querying (something that can even be a tough tool for computer scientists to use properly in various languages). Beyond that, there were also some issues with understanding specific syntactical areas of the language. Overall though, the authors seemed optimistic about the future use for their language.

Both versions of the paper were interesting reads. I need to read more papers that give example DSLs in order to gain ideas for my own DSL projects that I will be undertaking later in this class. This paper showed me some of the behind-the-scenes work that goes into a DSL, before code is even written. I plan on refreshing my knowledge on computer science language theory for my own languages. It is better to have a structure planned before you start extending a language into a DSL.

2.4 DSL Engineering 1

31 January 2019

DSL Engineering 1

Markus Voelter

After last week, I realized I had bitten off more than I could chew with regards to learning about Domain Specific Languages (DSLs). What I read made sense to me, but I was having trouble giving the information context; this was probably reflected in my questions to Professor Fluet on the papers I read. I decided I needed to take a step back with my reading material and look at one of the introductory texts he had recommended I read. Obviously, I could not commit to reading an entire textbook on DSLs in one week, but I did make my way through most of the section on an introduction to DSLs.
The introduction to DSLs section helped answer some of my questions. The main thing that was helpful was how it gave me definitions for terminology I had been reading in other papers. In this section, Markus Voelter explained the differences between terms like: target platform, execution engine, interpreter and generator. He also went into the differences between concrete syntax and abstract syntax. However, the definition I enjoyed learning about the most, though it is not actually that useful to DSL creation, was technical, used by programmers, versus application domain, used by non-programmers, DSLs (I like that there is a divide between the two types of DSLs and I have actually seen the difference between these types in the DSLs I have observed). Beyond just definitions, Voelter also discussed something that I briefly asked professor Fluet about: the distinction between General-Purpose Languages and DSLs. Specifically, the grey area around things like SQL or Mathematica (basically, it is up to interpretation).

The other helpful thing Voelter did was discuss modeling and model-driven development. My guess is that he will go into how to use models for devolvement in a later section, but it seems like they are useful for creating DSLs. What Voelter defines as good DSLs are modular languages: a basic language with modules that can be added to provide scope for programs. This was what was discussed in the Haskell paper I read last week, though that paper was a bit vaguer. After that, Voelter went into the benefits and issues of creating DSLs. I already knew most of these, but it was still a good refresher to read them again.

The section I read from Markus Voelter’s book was useful. It was a solid introduction into the topic of DSLs. I did not get to the end of it, which had some subsections on the applications of DSLs, and I will try to read that next week. In future weeks, I also plan on reading other sections in order to gain further knowledge in other DSL topics. Specifically, I would like to read the section after the introduction DSLs that goes into DSL design.

2.5 DSL Engineering 2

9 February 2019

DSL Engineering 2
This paper is an extension of my first one on the book by Markus Voelter and, as such, is briefer and more direct. I finished the first paper before the final few subsections on the introduction section of the book. In this installment, I finished the introduction section and started the next section on Domain Specific Language (DSL) design.

The final subsections of the introduction section of the book go into the field of DSLs as a whole. They cover these topics: the benefits of DSLs, the challenges of DSLs, applications of DSLs, and differentiation from other works and approaches. The stated benefits of DSLs are pretty straightforward: improved productivity, domain expert involvement, and easy validation were the points that stuck out to me. For the challenges of DSLs, the effort involved, the need for language engineering skills, evolution and maintenance and DSL Hell, getting too stuck into creating new DSLs for every new problem, were the points I felt made most sense. With the Applications of DSLs subsection, Voelter talks about the various types of DSLs. This felt abstract and categorical, since the end result is fairly similar for all of them—some form of DSL. Finally, the author discussed the differentiation from other works and approaches. This involved going into the difference between internal, or embedded, and external DSLs, how UML relates to DSLs (peripherally) and graphical versus textual DSLs (Voelter prefers textual for programming languages).

I only got a little way into the second part of Voelter’s book, DSL Design, but it seems much more interesting than the introduction. In the first subsection, he breaks down programs, languages, and domains. Essentially, a language covers a subset of all conceivable programs. The area it covers is called its domain (which can be broken down into even smaller domains within itself). There are two approaches to defining a domain: bottom-up or top-down. The bottom-up approach involves defining a domain in terms of existing software used to address a particular class of problems or products. For top-down, a domain is considered a body of knowledge about the real world; this is much harder since you have to be an expert on the body of knowledge to define the domain.

That is as far as I have gotten through the book. Due to its well-written nature, I will probably read this throughout the course (even if I try to read other papers next week in preparation for my first project).
This is my final writeup on the excellent book on Domain Specific Languages (DSLs) by Markus Voelter. It has proven to be a good resource for learning about the broader concepts of DSL creation and I intend to read the rest of it (without the writeups however). In this installment, I cover the sections on DSL after section 3.1.

In the book, Voelter uses the terms program and model interchangeably. It is important to know what you want from a model and a DSL. Voelter explains how the purpose of a model can be automatic derivation of a parent-domain program, formal analysis and model checking, platform-independent specification of functionality or generation of documentation. Beyond explaining the thought process for DSL creation, Voelter also covers concrete and abstract syntax. His definitions are as follows: concrete syntax is the notation a user uses to interact and edit a program, while abstract syntax is the data structure that represents the data expressed by a program. Furthermore, he defines parser-based systems as those that map the concrete to the abstract syntax and projectional editors as those that have the user directly change the abstract syntax (even though they perform their changes through the concrete syntax); Spoofax is an example of a projectional system (he does not say in the book, but I think Racket is an example of a parser-based system).

He then gives definitions for fragments, languages, functions, relationships, independence, and homogeneity. The definitions are fairly straightforward. How they interact with respect to DSLs is used to show the different variations and connections between various domains and their corresponding languages.

In the next section, which I have not yet completed, he explains the balance of expressivity. A DSL creator want their language to have a lot of expressivity in each method, function, and whatever
else they use to make the language. However, even though it is possible to make a language run off of just one symbol, this is trivial and useless; you would have to create a new language for each new problem you encountered. Voelter’s point is that one needs balance when creating a language.

This book is very useful. I intend to keep reading it throughout this course. It helps to clarify concepts and gives me more useful questions to ask Professor Fluet.

2.7 The Spoofax Language Workbench

9 February 2019

The Spoofax Language Workbench

Kats & Visser

So far, I have read papers that cover Domain Specific Languages (DSLs) in Racket and Haskell. I chose this paper since one of my projects will have to be written in Spoofax. In this paper, Lennart Kats and Eelco Visser cover everything you need to know about Spoofax. Much of the paper’s sections only made sense to me on a conceptual level, but this will be a good resource once I actually start working in the language and have a better idea of the more basic details of Spoofax. For this writeup, I will cover the broader ideas put forward by Kats and Visser.

According to Kats and Visser,

“Spoofax is a language workbench for efficient, agile development of textual domain-specific languages with state-of-the-art IDE support.”

This is its main selling point. From what I can tell, Racket and Haskell do not require too much knowledge of things beyond just them. Spoofax is not as simple, but it does have the benefit of being usable on an IDE. In the paper, the authors talk about how it can be used on the Eclipse IDE (I checked, and it looks like Spoofax may also be usable on IntelliJ through an independently-created plugin).

In order to define a language in Spoofax, you follow the Eclipse plugin project model. Beyond the files that all plugins have, an Eclipse Spoofax plugin has three parts: a syntax that is defined using Syntax Definition Formalism (SDF), which is used to describe formal languages and their syntax; editor services, which are defined in Editor Service (ESV) files; and semantic definitions, which are defined using the language Stratego. From what I read, this setup allows the language creator to incrementally and selectively create parts of the desired language (the paper really stressed the agile aspects of development in Spoofax).
Beyond this general idea, the paper also covers the language-creation process in great detail. It has a section that covers how to use SDF. The details do not need to be covered here, but the main takeaway from that section was how Spoofax uses the SDF syntax that you define to create a parser for the language. The editor services section was fairly brief, it did not provide enough information for me to truly understand how that part of Spoofax works; I will have to more research there too. For the section on Stratego, Kats and Visser go over how Stratego can be used to desugar and transform a DSL into something that can be run on the IDE. It also provides nice type- and error-checking capabilities. Due to my unfamiliarity with these three tools, I plan on researching and learning more about them once I am working on my Spoofax project.

Beyond going into the tools used in Spoofax, Kat and Visser also went over their general observations for this tool. I believe this will be the most complicated tool I use in this course – based on just how many different things I will need to know. I definitely plan on doing more reading before I start my Spoofax project.

2.8 The RASH JavaScript Editor (RAJE)

14 February 2019

The RASH JavaScript Editor (RAJE)

Spinaci et. al.

For my final paper, I wanted to read up on another example of a Domain Specific Language (DSL) that has been fully implemented. Based on Professor Fluet’s suggestion, I looked into RASH. RASH stands for Research Articles in Simplifies HTML. Research publications have begun allowing non-PDF submissions. Specifically, they now allow articles to be written using HTML5. RASH is a markup language, much like a more user-friendly version of LaTeX, that restricts the user to 32 key elements in HTML and enables the use of Research Description Framework (RDF) statements.
The idea behind the language is interesting in and of itself, but I found a newer development that was even more intriguing than its initial implementation; a word processor for HTML via the RASH framework.

In the paper put forward by Spinaci, Peroni and Di Iorio, an improvement is proposed for the use of RASH. The authors have created what they refer to as a What You See Is What You Get (WYSIWYG) word processor for writing in HTML called the RASH JavaScript Editor (RAJE). This editor is user-friendly; it utilizes the GitHub API to allow users to store their articles in a repository and be able to see their version history. I find this development for RASH interesting because I have been reading about graphical and textual implementations of DSLs but have only seen textual examples.

The specifications of RAJE are fairly simple. It is a rigid system but seems easy to use. It allows users to include mathematical notation through the Asciimath tool. Beyond that, it is a clickable interface where the writer enters metadata, like title and author name, mathematical notation, and text by clicking on various regions in the document (all of this is encoded to correspond with the specifications of RASH). Additionally, the author pushes the encoded article to GitHub, using the company’s API, which allows for collaborative editing. All of this is run the RAJE core scripts: init.js, which initializes all variables; caret.js, which defines utility methods for caret, a graphical text editing tool; const.js, which defines constant values that are used by other scripts; raje.js, which implements a set of actions to add elements into the body; shortcuts.js, which defines the editor’s shortcuts; toolbar.js, which defines the entire graphical set of elements; derash.js, which converts the code to a usable version of HTML; and rendered.js, which has everything needed to communicate with other services like GitHub.

All of this work has led to an excellent alternative to how research papers can be written. PDFs are fine, but they lack the interactivity of an HTML document. The authors tested their system out on six people, which did seem like a small grouping to me, and the people gave it a reasonable high usability score (interestingly enough, it seems the more background knowledge one has in writing these types of articles, the harder the system is to learn). This is a useful tool and one that should see solid usage in academia.
2.9 Modeling HTML in Haskell

5 April 2019

Modeling HTML in Haskell

Peter Thiemann

For my final paper, I read “Modeling HTML in Haskell” by Peter Thiemann. I read it because I felt that I could use even more information on Domain-Specific Languages (DSLs) in Haskell. This paper provides a nice overview of how to create an embedded DSL in Haskell as well as the downsides of converting a completely different set of rules, in this case HTML’s, to Haskell’s structure.

The goal of the language is to provide a better interface for web-page creation that will reside in the middle ground between tools with WYSIWYG, what you see is what you get, front ends and tools that are purely textual representations of HTML. Thiemann uses Haskell’s combinator library to build an HTML page as one large data structure (with robust constraints provided through Haskell’s type classes). This tool can be used to create and manipulate entire web sites.

Thiemann provides some excellent examples of his embedded DSL in action. All HTML definitions start with doc, which has a return of type HTML. Shown below is a simple “Hello World” example of using doc and other functions to make the most basic version of an HTML document. The function make_html build up an HTML object (with the help of the combining add function, which combines all of the other functions’ HTML objects). The type checking for things like HTML, HEAD and BODY ensure that the HTML output is properly aligned with existing rules for the language. The output is then displayed with the show_html function, which displays the actual HTML code.

Thiemann adds even more versatility to his language with the inclusion of parameterized documents. With the parameterizing functions like add_contents users can specify how documents
should look (in this case, this would force all documents to look like the “Hello World” HTML document). He has many “transformer” functions that can be used to mutate the BODY, and other parts of an HTML document, into a more desirable form.

Finally, the embedded DSL has linked nodes, which can be linked to the next, previous and higher order nodes. Using the function node2html, a node becomes an HTML data structure and the links become hyperlinks. All of this, and the previously mentioned functionality, is done through the use of extensive data types and encompassing classes.

Overall, Thiemann seemed to be happy with his creation. His only issue was how it was impossible to have everything defined through subclasses; he had to use the format shown below for certain things. The issues for this DSL seem to stem from how HTML’s rules and structure are different from Haskell’s. These seemed to me to be minor issues and the language seems to be an effective use of Haskell.
3 Projects

Here are the three projects done for the class: two smaller projects and one larger one. These write-ups are copied from the projects' READMEs on GitHub (https://github.com/joshuasellers/personal/).

3.1 CardGameDSL

Goal
I want to be able to create the logic and syntax that can be used to create the rules for card games in the form of a Domain Specific Language.

1. Domain is any card game involving a regular playing card deck
2. Initial work done over the course of two weeks
3. Room still left for future improvements
4. Overall Usability
5. I envision this being used to create any kind of game that requires one or more regular playing card decks (so not something like Uno). Essentially all the rules and scoring must involve the cards from the deck (this excludes games like Spoons where you have actual spoons in play).

The first iteration has enough easy usability for games like Texas Hold’Em and Five Card Draw. It can do more, but that involves more practice and refinement with the language.

Description
This DSL was written in Haskell. I chose the language due to my familiarity, I used it in a prior course, and its flexibility (you can do quite a bit with it ... if you are able to find and understand the information; the Wikis are a bit dense). I especially used Haskell’s data and class tools in Haskell to great effect.

The DSL is comprised of seven data types. They are housed in Types.hs.
<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rank</td>
<td>e.g. King or Queen</td>
</tr>
<tr>
<td>Suit</td>
<td>e.g. Spade or Clubs</td>
</tr>
<tr>
<td>Card</td>
<td>Each Card is a combination of a Rank and a Suit</td>
</tr>
<tr>
<td>Player</td>
<td>Has things like Hand and Score associated with it</td>
</tr>
<tr>
<td>Dealer</td>
<td>Essentially a Player with access to the deck-functions</td>
</tr>
<tr>
<td>Table</td>
<td>For games like Texas Hold’Em</td>
</tr>
<tr>
<td>Game</td>
<td>Has all of the programmer-defined rules</td>
</tr>
</tbody>
</table>

These datatypes are used throughout the DSL and are the building blocks that go into almost every function. Here is the code for Card:

```haskell
data Card = Card {_rank :: Rank, _suit :: Suit}
deriving (Eq)
instance Show Card where
    show (Card r s) = show r ++ show s
```

There are two other important files: Game.hs and Functions.hs (I also made example files, but they are not relavent to the DSL itself). Function.hs contains all fo the general functions for the DSL. Examples include draw and deal:

```haskell
{- Drawing Functions -}

class Drawing c where
draw :: Dealer -> c -> ([Card], Dealer)
instance Drawing Int where
draw dealer 0 = ([],dealer)
draw dealer n = ((take n deck), d)
    where deck = _deck dealer
d        d = dealer {_deck = (drop n deck)}
instance Drawing Card where
draw dealer c = (card, d)
    where out = removeCard (_deck dealer) c
d        d = dealer {_deck = (snd out)}
card = fst out
instance Drawing [Card] where
draw dealer cards = (cs, d)
```
where out = drawSpecificCards (_deck dealer) cards
cs = fst out
d = dealer {_deck = (snd out)}

{- Deal Functions -}

class (Drawing n) => Dealing p n where
deal :: Dealer -> p -> n -> Dealer
instance (Drawing n) => Dealing Dealer n where
deal d1 _ x = dealer
  where (cards, dlr) = draw d1 x
dealer = dlr {handD = cards ++ (handD dlr)}
instance (Drawing n) => Dealing Player n where
deal dealer player x = d
  where (cards, dlr) = draw dealer x
dl = dlr
  p = player {hand = cards ++ (hand player)}
d = dl {players = replace (players dl) player p}

dealToDealer :: (Drawing n) => Dealer -> n -> Dealer
dealToDealer dealer n = deal dealer dealer n

deals :: (Drawing n) => Dealer -> n -> Dealer
deals dealer n = foldl (\ b x -> deal b x n) dealer (_players dealer)

The class allowed for polymorphism when it comes to commonly used functions like draw and deal. I had to include these tags to make them work:

{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE MultiParamTypeClasses #-}

For Game.hs, I put all of the rule-specific functions into there. The most important functions, that are also included in the Game datatype, are involved with comparing hands and scoring. I made the decision to have the user actually define those areas, to allow for more flexibility, but I did write numerous helper functions for the user to utilize like these:

-- combine suit and rank into card ranking
fullCardRanking :: Game -> [(Card, Integer)]
fullCardRanking game = c
    where s = sortBy sortGT (_orderSuit game)
    r = sortBy sortGT (_orderRank game)
    c = combine r s
    sortGT (_, b1) (_, b2)
        | b1 < b2 = GT
        | b1 > b2 = LT
        | b1 == b2 = EQ

combine :: [(Rank, Integer)] -> [(Suit, Integer)] -> [(Card, Integer)]
combine [] _ = []
combine _ [] = []
combine (r:ranks) s = (foldl \ b x -> if (((snd $ (last s)) - (snd $ (head s))) == 0)
    then b ++ [(Card {_rank = fst r, _suit = fst x}),
    else b ++ [(Card {_rank = fst r, _suit = fst x},
    [] s) ++ combine ranks s

Example Code
I made an example to show some basic usage of the DSL. I created a simplified version of Five Card Draw Poker. I reused my compare method from a previous project that specifically focussed on Poker and Cribbage. The files for this example are: ExampleMain.hs and PokerExample.hs. I made a simple main to run a game.

Next Steps
1. Update discard and fold to not return a tuple (I already fixed that for deal)
2. Add betting functions and fields for players and dealers
3. Add in odds-calculating functions so that programmers can encode a computer oponent
4. Test out some non-traditional cards games (like War or Go Fish)
5. Clean up some functions that are outdated after I improved other, similar, ones
6. Add in more helper functions to Game.hs - PokerExample.hs still wound up being a fairly large file
3.2 AccountingDSL

Description
For this project, I created a Domain-Specific Language (DSL) for the first two steps in the Accounting Cycle. I used the language Racket for the project. The domain was specified to be anything contained within those first two steps in the cycle:

1. Journals Entries
2. Ledger accounts
3. All of the information needed to make the structures

The goal of the project was to make a basic language that accountants could use to make their initial journal of entries (all of the information that is used in the later steps of the accounting process). Then, commands could be utilized to turn that journal into things like the ledger and other data structures used further on in the cycle.

Installing
To install the project, make sure you have Racket downloaded on your device. Additionally, make sure you have the modules of Racket known as: Beautiful Racket (bf), while-loop, and Gregor (gregor). These will be needed to run the DSL. To install them, run this command:

```raco pkg install --auto while-loop```

Overall Usability
There are six files for this project: the parser, the lexer, the tokenizer, the example code, the reader and the expander. The example file contains sample code and can be edited for your convenience as you wish to test additional cases. The parser and lexer contain the rules for the language:

1. Every file is an ac-line
2. Each ac-line can either be a journal-entry, command, conditional, value or loop
3. The commands are show, clear and ledger
4. The values are len, bool, int and date
5. The journal entry is formatted as so: entry-datedebitscredits
6. An entry date is of the form: yyyy-mm-dd
7. Debits and credits are lists of the debit/credit accounts and the amounts going into them
Description
The reader uses the parser and lexer to tokenize and parse files. I don’t have any specific error catching mechanisms in place, so incorrect code should return some type of parsing error or reading error.

Parsing:

```
ac-program : [@ac-line] (/NEWLINE [@ac-line])*
ac-line : journal-entry | @command | conditional | @value | loop
```

```
conditional : "["((bool-func @value @value) | bool) "]" /"?" @command ":" @command ...
```

Lexing:

```
(define basic-lexer
  (lexer-srcloc
   ["\n" (token 'NEWLINE lexeme)]
   [whitespace (token lexeme #:skip? #t)]
   [date (token 'DATE lexeme)]
   [digits (token 'INTEGER (string->number lexeme))]  
   [account (token 'ACCOUNT lexeme)]
   [bool (token 'BOOL lexeme)]
   [(:or "clear" "ledger" "show" "len" "date" ">">" "<" "=" "<=" ">=") (token lexeme lexeme)]
   [(char-set "/=%<>{}#\[",?:") lexeme]))
```

The tokenized, lexed and parsed code is then sent to the expander. This is where the accountingDSL code is expanding into usable racket s-expressions. In order to more accurately mimic the format of racket, I have the the expander work in a functional manner (e.g. no stored global values or data structures). Essentially, I fold over the ac-lines. Each time the fold function encounters a journal-entry, it adds it to the fold output value using cons. For the commands and other non-entry options, the action is performed. Everything returns a list containing the current journal and date.

Sample expander code:

```
;;;;;;;;;;;;;;;;;;;;;
;; ac-program funcs ;
;;;;;;;;;;;;;;;;;;;;;
```
(define (fold-funcs apl ac-funcs)
  (for/fold ([current-apl apl])
    ([ac-func (in-list ac-funcs)])
    (define ret (apply ac-func current-apl))
    (if (equal? (length ret) 2) ret (cdr ret))))

(define-macro (ac-program ENTRIES ...)
  #'(begin
      (define ledger (list empty null))
      (void (fold-funcs ledger (list ENTRIES ...))))
  (provide ac-program))

Example Code

I made an example to show some basic usage of the DSL. The file for this example is: accounting-test.rkt.

Next Steps

Overall, this is a solid DSL. This was my first time working in Racket, so in the future I'd like to move away from beautiful racket and stick to the actual code (beautiful racket hides some aspects of racket to make things easier for beginners). Also, there are a lot of rules in accounting. I only covered a couple in this DSL, but I would need to factor in all of them for a full version of this DSL. A full version for this DSL would be the next step (the other steps in the Accounting Cycle would mainly involved updates to my ledger code, which is very basic right now).

3.3 Final Project

Description:
For this project, I created a Domain-Specific Language (DSL) the describes low-level hardware. I see it having use in classrooms as a learning tool. I got the inspiration from my own computer systems class; I remembered having trouble understanding the concepts (the seemed a lot more abstract than coding). I though by having a physical tool that simulated the systems, students could better understand hardware.

This project is essentially only limited by the hardware in existence. I created the functions for the
most common features, but you could keep adding more things in order to cover all possible facets of hardware creation.

**Installing:**
To install the project, make sure you have Haskell downloaded on your device. I used GHCI on my machine. Then, just download the Haskell files.

**Overall Usability:**
This code can theoretically be used at multiples levels. Right now, I have the most basic hardware encoded, and these functions could be used to build the more complicated pieces, but you can create most aspects of hardware with it. If I have time, I will make more pieces since there is still a bit of Haskell knowledge needed to make new items (which goes against the idea of a DSL).

**The Code:**
There are four files:

1. Data: the most basic parts like Bits, Bytes and Registers
2. Hardware: items like gates, switches and other more complicated structures
3. Test: I created a test file for the extremely complicated parts (like a register file)
4. ExampleHardware: used to show how a user could create their own hardware

This is the code for a decoder:

```haskell
n_to_2n_decoder :: [Bit] -> [Bit]
n_to_2n_decoder [] = error "n_to_2n_decoder invalid input"
n_to_2n_decoder bs = o
  where o = decoder_helper bs negs
        xs = unfoldr (\ b -> if (b == (length bs)) then Nothing else Just (b,b+1)) 0
        negs = powersort xs

n_to_2n_decoder_enable :: Bit -> [Bit] -> [Bit]
n_to_2n_decoder_enable _ [] = error "n_to_2n_decoder invalid input"
n_to_2n_decoder_enable e bs = o
  where o = map (\ x -> if ((bitVal e)==0) then e else x) (decoder_helper bs negs)
        xs = unfoldr (\ b -> if (b == (length bs)) then Nothing else Just (b,b+1)) 0
        negs = powersort xs
```
decoder_helper :: [Bit] -> [[Int]] -> [Bit]
decoder_helper [] _ = error "decoder_helper invalid input"
decoder_helper _ [] = []
decoder_helper bs (n:negs) = neg bs n : decoder_helper bs negs
  where neg [] _ = error "neg invalid input"
    neg bits [] = and_gate bits
    neg bits (i:ns) = neg (replaceNth i (not_gate (bits!!i)) bits) ns

Those functions, as well as others, are used to create functional representations of the hardware.

Example Code: For the example file, I made a Arithmetic Logic Unit (ALU). This seemed like a good example for a couple reasons. First, while I could have made it one of the functions, I contains most of the items I had already hardcoded so it would be a prime example of what a user could do with the DSL. Second, the structure itself, unlike something like a register file, is fairly simple; it would not require a lot of extra Haskell code to complete and would be a good example of the DSL.

Next Steps:
As I said before, my only limitation is when I get tired of making hardware. A class of students could use any level of my code, but I could make all hardware possible as single functions. There are still some basic items I would like to make. I still need to add in delays and time-related items like Flip-Flops. That would probably be simple to code, but would require a great deal of thought to come up with an elegant solution. Overall, this has helped me understand hardware better and I hope it can be used to help other students as well.

4 Conclusion

I appreciated the versatility of this class. Not only was I learning about DSLs, but I gained research abilities, improved my project management skills and learned more about Haskell and Racket. In creating a DSL, you need to have a complete understanding of the problem; I wound up gaining a lot of knowledge in card games, accounting and hardware. Once I had the requisite knowledge, I had to create the DSL. It turns out, and this is applicable to many areas of development, that the more modular you make your work the better your output. In keeping your tasks small, you actually wind up being more productive; DSLs are a perfect example of this since they grow modularly. Finally, I had coded in Haskell before, and this class helped to keep my skills sharp, but I had never even heard of Racket. Adding another functional language to my repertoire was an
enjoyable experience.

Learning about DSLs is helpful for computer science students. They are an interesting part of the language ecosystem and can be much easier to learn and develop than general-purpose languages. This class has improved my coding skills and development abilities; I look forward to using my newfound abilities in future projects.
5 Appendix

In this appendix, the work done is broken down on a week-by-week basis (any deviation from my original plan are explained).

5.1 Work Completed

Week 1: Papers:

(a) Creating Languages in Racket
(b) Building Domain-Specific Embedded Languages

Week 2: Papers:

(a) POP-PL: A Patient-Oriented Prescription Programming Language
(b) DSL Engineering 1

Week 3: Papers:

(a) DSL Engineering 2
(b) The Spoofax Language Workbench

Week 4: Papers:

(a) DSL Engineering 3
(b) The RASH JavaScript Editor (RAJE)

Week 5: CardGameDSL:

(a) Worked on DSL
(b) Researched and planned Haskell functions
(c) Created initial code

Week 6: CardGameDSL:

(a) Finished DSL
(b) Wrote README

Week 7: AccountingDSL:

(a) Worked on DSL - ran through related tutorials
(b) Researched and planned Racket functions
(c) Created initial code

**Week 8:** AccountingDSL:

(a) Finished DSL

(b) Wrote README

**Week 9:** Spoofax:

(a) Researched Spoofax

(b) Created initial code

**Week 10:** Spoofax:

(a) Continued researching Spoofax

(b) Tried different websites’ coding techniques

**Week 11:** Accounting DSL and Paper:

(a) Added more functionality to DSL

(b) Cleaned up existing code

(c) Modeling HTML in Haskell

**Week 12:** Final Project and Class Write-up

(a) Researched low-level hardware

(b) Wrote intro and set up outline for paper

**Week 13:** Final Project and Class Write-up

(a) Started making hardware

(b) Added in research paper write-ups

**Week 14:** Final Project and Class Write-up

(a) Added in majority of hardware

(b) Added in project write-ups

**Week 15:** Final Project and Class Write-up

(a) Tested functions and added example code

(b) Finished paper
5.2 Explanation of Deviations

The only real deviation from my planned work was that I did two weeks of work on a, planned, Spoofax DSL, but was unable to come up with anything worth turning in. This was due to the lack of useful resources, like tutorials, for Spoofax online. It is a fairly complicated language to work with and I would have needed more information to make anything useful. Hence, I removed the planning phase for the final project and did more work on my AccountingDSL and read another paper. This allowed me to still get in meaningful work.