

# EECS 510 Final Project

12.13.2024

---

Sam Grimsley

## Pt. I - Formal Language Design

This language is focused on the world of rock climbing; specifically, how do climbing holds and moves translate to a specific grade? To those who aren't climbers, it can be very difficult to see how routes differ in difficulty. After all, at a certain point "hard" is just "hard" and the grades can seem arbitrary. As such, this is an attempt at translating a sequence of holds and moves (or in rock climbing terms, "beta") to a more definitive grade.

In order to translate climbing beta into grades in a language, we must first define what makes up a beta. This is divided into two categories, the elements of which form the basis for our alphabet: holds and moves.

Holds in climbing all have very specific names, some of them intuitive like handle or sloper (literally a sloped hold), while others less-so such as crimps (think the edge of a credit card). To avoid some of these complexities (as well as others that will be mentioned in a moment), we will restrict our alphabet to the following holds: crimp, sloper, jug (a big, great hold), handle, side cling, under cling, pinch, pocket.


Likewise, climbing moves can also have fairly confusing names. While fans of Beauty and the Beast might understand what a gaston is, figuring out what barn doors, chimneys, and bicycles have to do with climbing can take a moment. However, as this language is intended more for climbers than those with no experience in the matter, we will still include these in our alphabet: mantle, barn door, chimney, dyno, static, stem, drop knee, gaston, bump, flag, rock over, walk through, match, layback, heel hook, toe hook, bicycle, deadpoint, knee bar.

To make this language work, we also add a third, much smaller category to the alphabet called style. In climbing, routes are typically divided into boulders (routes on short walls typically up to around 15 feet) and big wall or tall wall climbing (for which you are typically on a rope and climbing far higher than a boulder). These route styles are graded with separate systems, and to account for this we include boulder and tall wall as "letters" in our alphabet to mark these differences.

With this alphabet fleshed out, we now move to the format of strings in the language. Here, strings take the form of a complete beta, allowing the language to decode and grade a given route. We restrict strings to be of the following format, using the categories defined above:

Style Hold Move Hold Move Hold ... Move Hold Style

That is, we start and end with marking the style of route (The end is marked only for the grammar, and is not marked for the machine or code testing as in these cases it becomes



redundant), then list the holds and moves in an alternating pattern ending with the final hold. This is done to mimic climbing routes, with moves done between different holds.

Next we define the output, or grades to be used. In the US, the predominant grading systems used are the Vermin or V-Scale for bouldering and Yosemite Decimal System (YDS) for tall wall. The language will focus on these, although it could just as easily use any of the other systems in wide use throughout the world. The V-Scale uses  $V_0$  as the easiest grade, then  $V_1$ ,  $V_2$ , and so forth. For our purposes, we will stop with  $V_{10+}$  as the hardest, as while many grades do exist beyond this they are less common and the efficacy of this language will struggle to capture the differences. YDS defines grades in the manner of 5.x, with 5. meaning it is a climb and the x number being the difficulty. We set 5.6 as the easiest, and use 5.14+ (read as five fourteen) as the cutoff for the hardest.

It now becomes clear that we can transform the input strings of our language to the grades using some form of point system, and this is where the automaton for the language is focused. We assign points to each move and hold, then tally them up to transform them to a grade. Once we have a grade, we can then accept the string as part of the language, meaning it is both a valid route and we now have a grade for it. Notably, if a string does not get accepted then that means it is not a valid route (for instance, if it ends with a move then clearly there is no way to finish the climb as you can't grab on to nothing).

Now that we have an outline for how the language looks and functions, it is worth noting a few of the caveats of the language. Firstly, we are not using every possible hold and move type; there are simply far too many to cover in this project in a way that does them justice. Secondly, and perhaps more importantly, in reducing holds and moves to a given point total we are necessarily losing some information and making, in some cases rather large, assumptions about the route. Not all moves of a given type are equal, and in real life the start and end holds of a move can make a big difference, but accounting for this in the language proves rather complicated. As such, it is impossible to accurately represent all routes with this language; the goal is rather that it gives a rough idea and could serve as a starting point, both for climbers to determine grades and for future languages to build off of. Additionally, there is a tendency for this language to grade longer routes as necessarily harder than shorter routes, since points are totalled up. There is a modification to try to circumvent this as much as possible, but it is worth pointing out that with extra long routes this circumvention is not enough. However, the language should still work well for comparing routes of similar lengths. Additionally, experienced climbers can modify their use of the language to focus on grading the harder part of the route only, again improving results.

## Pt. II - Grammar - Double check against machine

The grammar for this language is divided into three sections: First, we translate the input string into points, represented in unary. Then, based on the style of the route, we transform this point total by dividing it (to account for the length of the route). Finally, with the updated point total, we assign this to a grade. For clarity, the grammar is likewise broken up into three sections.

### Input to Points

Here we transform the input string into the unary points, with holds and moves both turning into 1s. This is divided among the following table for ease of reading.

Each hold -> Some 1s	Each move -> Some 1s
Crimp -> 11	Mantle -> 11
Sloper -> 11	Barn door -> 11
Jug -> $\lambda$	Chimney -> 1
Handle -> $\lambda$	Dyno -> 111
Side cling -> 1	Static -> $\lambda$
Under cling -> 11	Stem -> 1
Pinch -> 1	Drop knee -> 1
Pocket -> 1	Gaston -> 11
	Bump -> $\lambda$
	Flag -> $\lambda$
	Rock over -> 1
	Walk through -> $\lambda$
	Match -> $\lambda$
	Layback -> $\lambda$
	Heel hook -> 11
	Toe hook -> 1

	Bicycle -> 111
	Deadpoint -> 11
	Knee bar -> 11

## Point Modification

With the string moved to unary in the last section, we next modify it based on if it is a boulder or tall wall route. For boulders, we divide by two. Likewise, for tall wall we divide by four. Note that Boulder and Tall wall are shortened to B and T respectively.

B1 -> C	T111 -> E
C1 -> 1B	E1 -> 1T
CB -> D	TT -> F
BB -> D	T1T -> F
	T11T -> F
	ET -> F

To ease the final conversion, we move the markers for tall wall/boulder back to the beginning.

1D -> D1	1F -> F1
----------	----------

## Point Total to Grade

We finally convert the string to the final grade. This conversion is done as simply as possible, with each unary converting to a grade value. This follows the pattern  $V_n 1 \rightarrow V_{n+1}$  and  $5.n 1 \rightarrow 5.(n+1)$ .

D -> $V_0$	$V_6 1 \rightarrow V_7$
$V_0 1 \rightarrow V_1$	$V_7 1 \rightarrow V_8$
$V_1 1 \rightarrow V_2$	$V_8 1 \rightarrow V_9$
$V_2 1 \rightarrow V_3$	$V_9 1 \rightarrow V_{10+}$
$V_3 1 \rightarrow V_4$	$V_{10+} 1 \rightarrow V_{10+}$
$V_4 1 \rightarrow V_5$	
$V_5 1 \rightarrow V_6$	F -> 5.6

5.6 1 -> 5.7	5.11 1 -> 5.12
5.7 1 -> 5.8	5.12 1 -> 5.13
5.8 1 -> 5.9	5.13 1 -> 5.14+
5.9 1 -> 5.10	5.14+ 1 -> 5.14+
5.10 1 -> 5.11	

With the grammar written out, we can now do a quick check with two test strings. First:

Beta = T Handle Static Handle Static Handle Static Side cling Gaston Jug Static Jug Static  
Handle T

This tall wall route should be an easy 5.6, being made up of easy holds and simple moves. From the grammar:

Beta => T 111 T => ET => F => 5.6

We thus reach 5.6 as our endpoint, giving us our grade estimation. This matches up well with the eye test for the route, meaning our language should work well for easy routes like this.

Now we test a route of higher difficulty:

Beta = B Crimp Match Crimp Bump Sloper Mantle Pocket Rock over Side cling B

This boulder route should be of intermediate difficulty, having some difficult holds and challenging moves but nothing too intense. Working through the grammar:

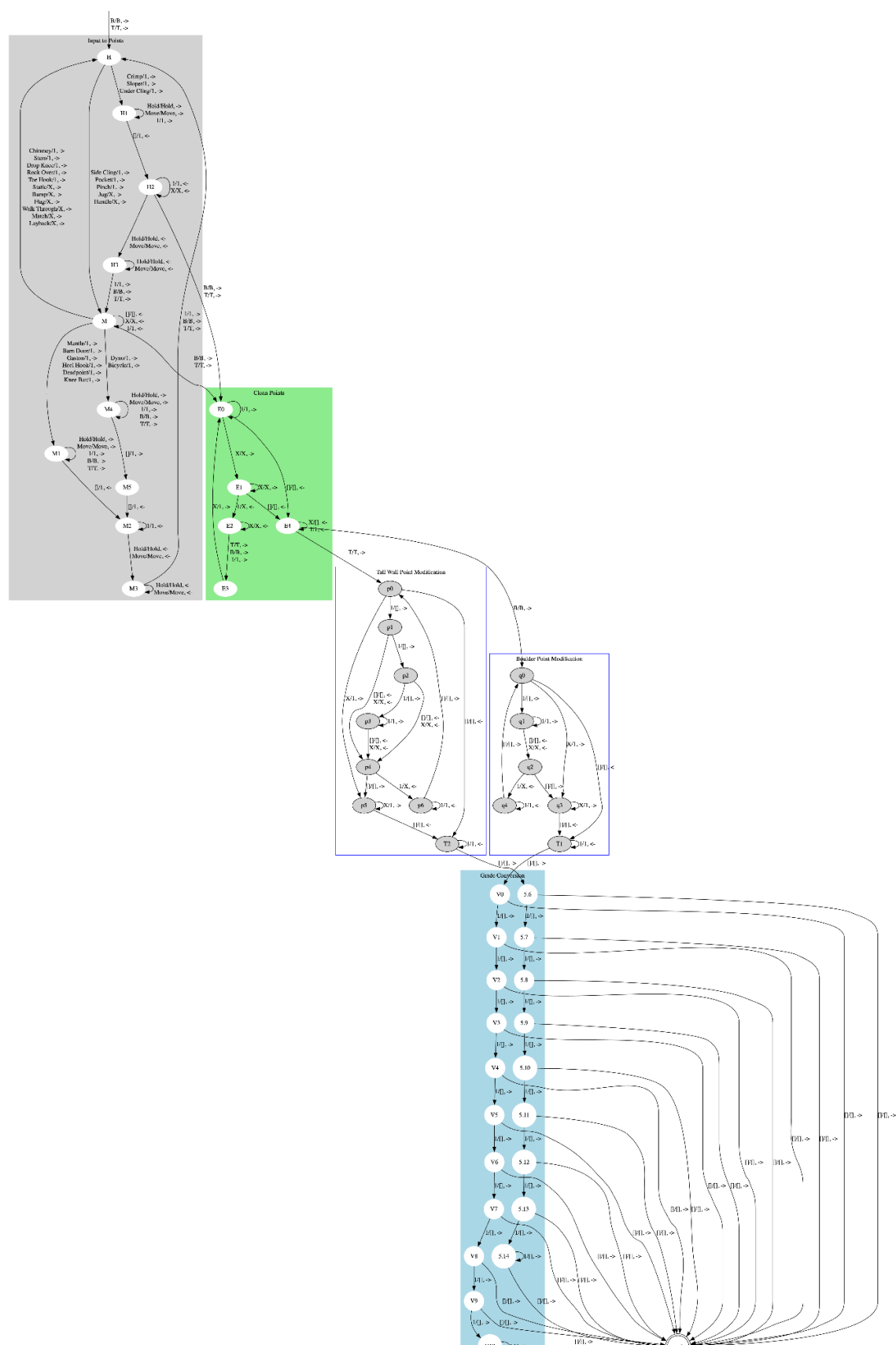
Beta => B111111111111B => 11111D => D11111 => V<sub>0</sub>11111 => V<sub>5</sub>

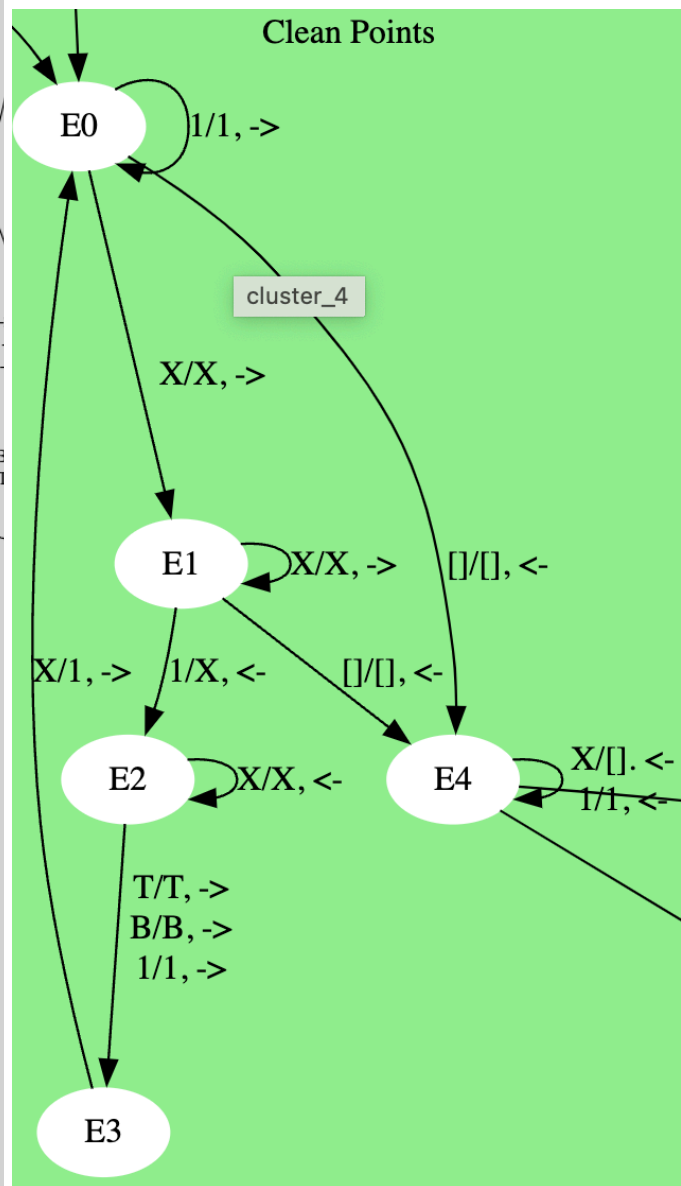
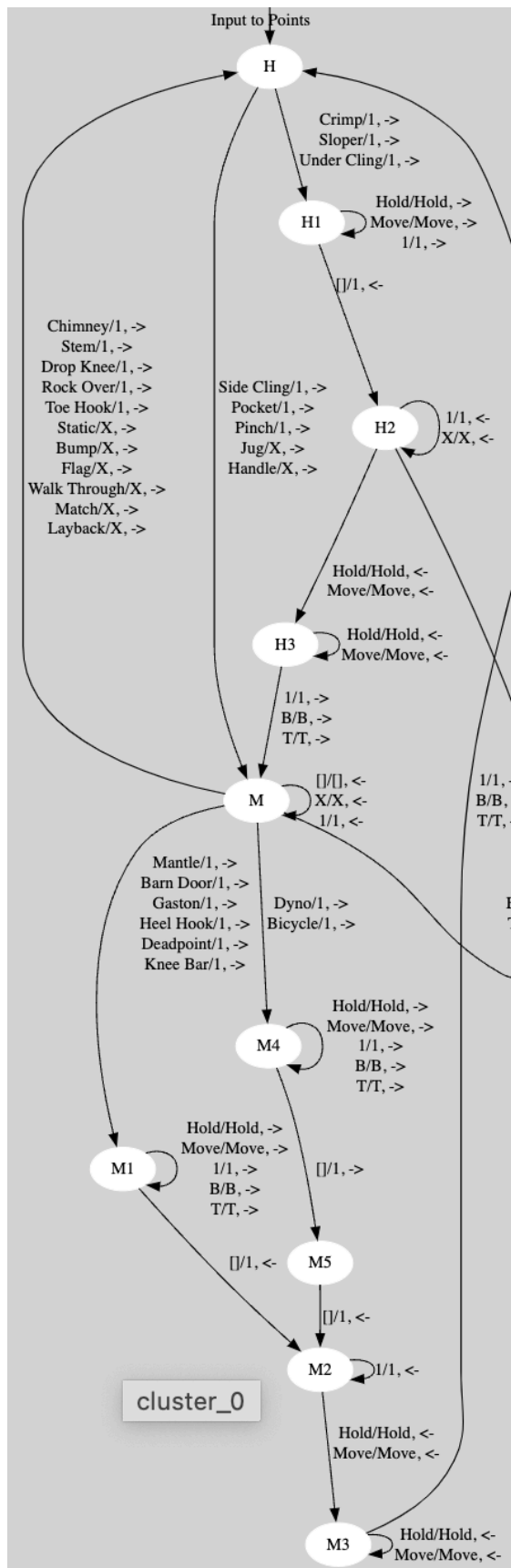
This gives us a V<sub>5</sub>, squarely in the middle of our scale and certainly a solid intermediate difficulty as we hoped.

Both our test cases passed the eye test for our grammar. Further testing shows it holds up for most strings, though as previously mentioned there can be some difficulties on the higher end of the scale and a tendency to overweight longer routes, both of which can be overcome by modifying how the language is used or perhaps with additional fine tuning and modification to the rules of the language.

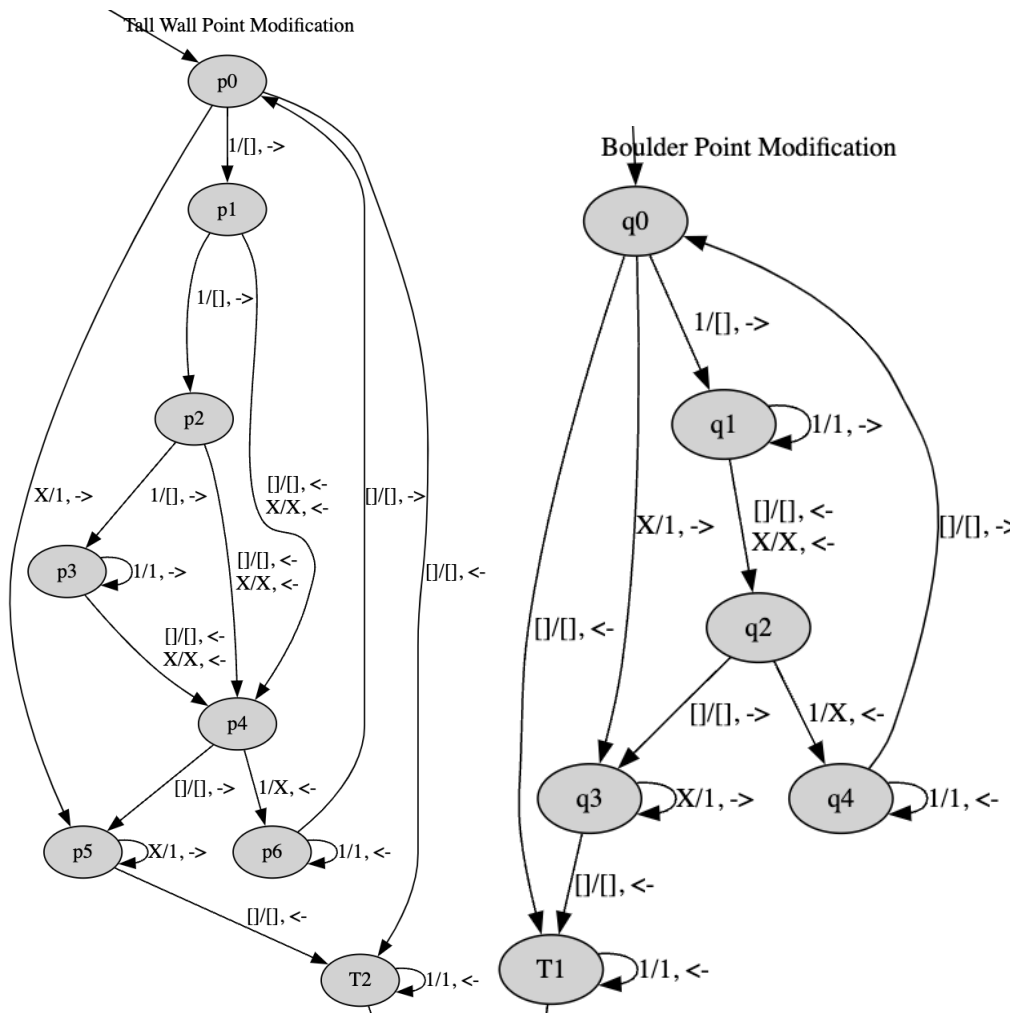
## Pt. III - Automaton

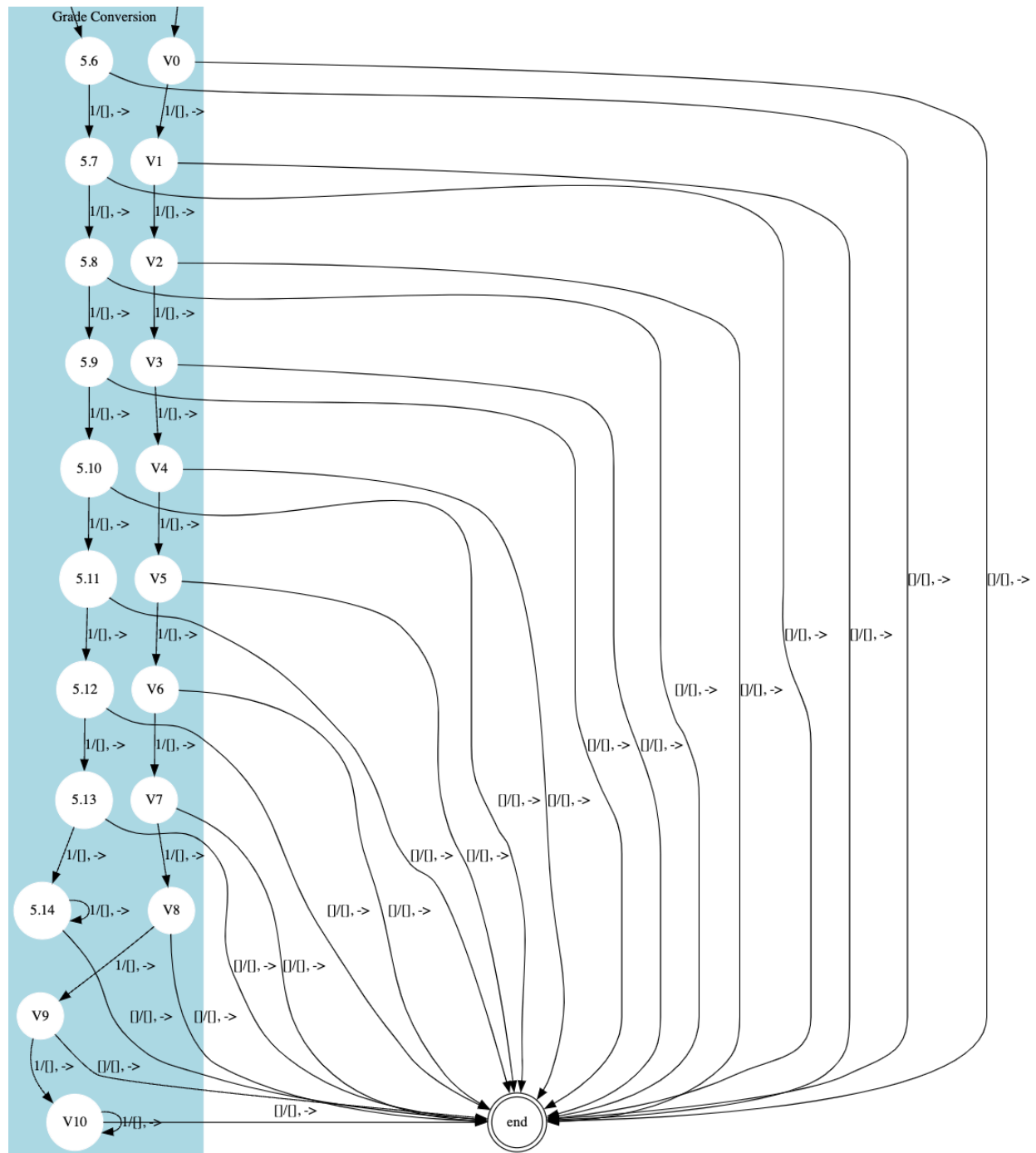
To make this more legible due to the size of the automaton, I am including the entire diagram as well as individual images of each section.











## Pt. IV - Data Structure

To represent this language we can use a tuple, dividing the information into the initial state, final state, alphabet, and transition functions. We have the initial state, Start; final state,

End; alphabet, defined above; and then the transition functions. Due to the size of this machine there are too many transition functions to list here. However, they are provided in lines 71-430 in the python file `language.py`, provided in the Github repository listed below.

As far as using this data structure, we can very easily transfer it into Python. The initial and final states are used to define the start and end conditions for a Turing Machine simulator (another defined class in `language.py`). The alphabet determines how the input string is created by the user, but is not explicitly necessary for the Python implementation. The transition functions define the transitions between each node. This is the bulk of the structure and the most important part; without this, the language cannot be implemented. The transition functions take the form of a Python dictionary, with the keys representing the original states and what to read and the values representing the new state, what to write, and the direction to move on the tape. That is:

{(State 1, Read): (State 2, Write, Direction), ...}

This format closely mirrors the delta notation typically used to write out states, helping with readability. To get a better idea of the data structure, a small snippet is included here:

```
alphabet = ("T", "B", "Mantle", "Barn door", ...)

initial_state = "Start"

final_states = {"End"}

accepting_states = ["End"]

transition_function = {("Start", "B"): ("H", "B", "R"),
                        ("Start", "T"): ("H", "T", "R"),

                        ("H", "Crimp"): ("H1", "1", "R"),
                        ("H", "Sloper"): ("H1", "1", "R"),
                        ("H", "Under cling"): ("H1", "1", "R"),
                        ("H", "Side cling"): ("M", "1", "R"),
                        ("H", "Pocket"): ("M", "1", "R"),
                        ("H", "Pinch"): ("M", "1", "R"),
                        ("H", "Jug"): ("M", "X", "R"),
                        ("H", "Handle"): ("M", "X", "R"),
                        ...}
```

## Pt. V - Testing and Code

The code can be found at the following Github repository. Download and run the Python file to see a few test cases, and either change the input strings or manually use the accept function to test additional cases. It is important to note that for the Python implementation, the alphabet must be separated with an underscore to distinguish between “characters”.

<https://github.com/SGrimsley02/EECS510>

Download language.py from the repository, then run in Python3. Alternatively, the Python file is included in the zipped submission.