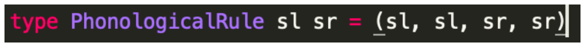
Creating a Simple Finite State Transducer

Introduction

I have a lot of interest in phonology and was curious how to implement phonological rules in a more convenient way. So for this project I decided to add onto the final problem in Assignment 5 and created a Haskell program that transforms a phonological rule into a finite state transducer. I use the same structure for the transducer as was used in the Assignment, and used the type *PhonologicalRule* as a way to define the rules:



```haskell
type PhonologicalRule sl sr = (sl, sl, sr, sr)
```

I am using the format of a normal phonological rule of the form:

$$\alpha \rightarrow \beta \,/\, \lambda \_ \gamma$$

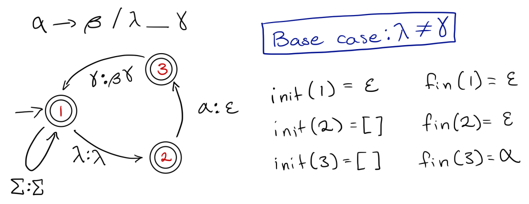
Where alpha changes to beta when it is between lambda and gamma. *PhonologicalRule* takes each part of the format and orders it in the code as

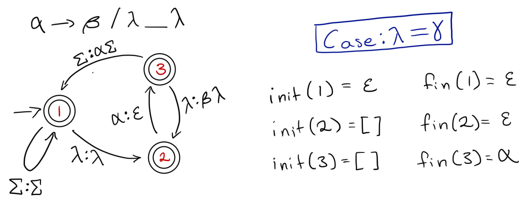
$$(\alpha,\ \beta,\ \lambda,\ \gamma)$$

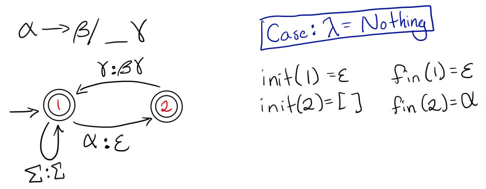
I based the parameter types off of the final problem, so for *sl* I used the type *Char* and for *sr* I used *String*. However, since it is not always necessary that lambda and gamma have values, I created data *PhonRight* for *sr* that has types *Maybe String* and *Nothing*. This way, if a phonological rule does not have a right or left hand side, the input can be *Nothing* instead.

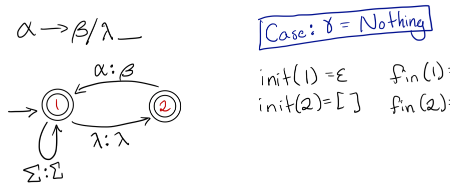
Some Helper Functions

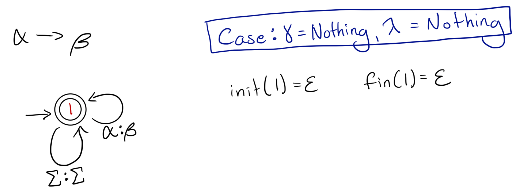
I found possible cases and diagrammed some basic outlines for creating the transducers shown below:











(Note: I forgot to input an arrow going back to state 1 for the second diagram, i.e. where gamma equals lambda, so it is not indicative of the functionality of said function)

I created a different function for each case and then combined it all into one function called *phonRule_to_Transducer*. An issue with these base cases I've diagrammed is that they only really work if lambda and gamma are composed of length 1. So in order to generalize the function and allow it to work for strings other than the given ones, I created some helper functions that break down strings into transition functions instead. This allows me to easily connect the integral states from the diagram to the varying string parameters.

Test Cases for the Main Functions

Referring back to the diagram above, I created 5 functions according to the templates for the transducers above. The function names (presented in the same order that they appeared on the diagram) are:

both_sides
both_sides_equal
no_lambda
no_gamma
no_neither

I then combined them all into one function called *phonRule_to_Transducer* which takes in a phonological rule and creates a transducer based on its lambda and gamma values. If the function has both lambda and gamma values, then if the values are equal it will call *both_sides_equal*, otherwise it will call *both_sides*. If lambda is *Nothing* but not gamma, it calls *no_lambda*. Vice versa, if gamma is *Nothing* but lambda is something, then it calls *no_gamma*. And if both are *Nothing,* then it calls *no_neither.* Below I have given a few test cases for each function. Further detail on the type of phonological rule is within the code written.

Tests for both_sides:

```
*Project> val (both_sides test1) "uten"
["uten","uTen"]
*Project> val (both_sides test2) "upten"
["upten","upTen"]
```

Tests for both_sides_equal:

```
*Project> set (val (both_sides_equal test3)
"ututuen")
["uTuTuen","uTutuen","utuTuen","ututuen"]
```

Tests for no_lambda:

```
*Project> val (no_lambda test4) "jutun"
["jutun","juTun"]
*Project> val (no_lambda test4) "jtun"
["jtun","jTun"]
*Project> val (no_lambda test4) "jtutun"
["jtutun","jtuTun","jTutun","jTuTun"]
```

Tests for no_gamma:

```
*Project> val (no_gamma test5) "utpen"
["utpen","uTpen"]
*Project> val (no_gamma test6) "uppertn"
["uppertn","upperTn"]
*Project> val (no_gamma test6)
"uppertvsuppert"
["uppertvsuppert","uppertvsupperT","upperTvsu
ppert","upperTvsupperT"]
```

Tests for no_neither:

```
*Project> val (no_neither test7) "utb"
["utb","uTb"]
*Project> val (no_neither test7) "t"
["t","T"]
*Project> val (no_neither test7) "tt"
["tt","tT","Tt","TT"]
*Project> val (no_neither test7) "atbtx"
["atbtx","atbTx","aTbtx","aTbTx"]
```

Tests for phonRule_to_Transducer:

```
*Project> val (phonRule_to_Transducer test7)
"atbtx"
["atbtx","atbTx","aTbtx","aTbTx"]
*Project> val (phonRule_to_Transducer test4)
"jtun"
["jtun","jTun"]
```

Issues:

This is by no means a fully working transducer creator. This program only accounts for the basic cases. For cases such as when a rule occurs at a word boundary the program tends to run into issues (specifically for *no_lambda*). Another kink I couldn't wholly figure out how to fix is an issue of over duplication of entries found within the list. This issue specifically arises in the function *both_sides_equal*. I believe this is due to how I implemented the creation of transitions, since *both_sides_equal* has the special cases where the gamma of phonological rule can overlap with the lambda of it. Unfortunately any attempts to fix this only caused the function to fail.

An example of the repetition issue with both_sides_equal. Calling *set* to remove all duplicates removes this issue and gives the correct value, however it does not do so on its own.

```
*Project> val (both_sides_equal test3) "ututuen"
["ututuen","ututuen","utuTuen","ututuen","uTutuen","uTutuen","uTuTuen"]
```

An example of the word boundary issue in no_lambda. It appears that the program gets confused when the first letter is the same as the gamma value, and ends up appending the beta value to it.

```
*Project> val (no_lambda test4) "utun"
["utun","uTun","Tutun","TuTun"]
```

Conclusion

This project was a good learning experience. It really made me think of how to approach a linguistics problem and solve it from the ground up. I found the idea of creating a function that can convert a myriad of phonological rules into transducers a daunting but fun task. If I had more time and were able to continue working on this project, I would like to add more functionality to the program. First I would start with fixing the issues I had with word boundaries and repetition For one thing I used the latin alphabet as my alphabet set, so changing it to IPA would be something that would me more practical in real world settings. I would also like to implement a more general way to write rules. As it is now, the program only works on individual sounds and not on groups of them (i.e. labials, dentals, fricatives, etc.). Since in a real world use setting phonological rules are typically for a group of sounds and not individual ones, that would also be something I'd like to add.