Gen with Lazy Evaluation*

Michael Hammond
U. of Arizona

November 1, 2009

1 Introduction

Optimality Theory (OT) now exists in multiple flavors, e.g. orthodox (McCarthy and Prince, 1993; Prince and Smolensky, 1993), stochastic (Boersma, 1997; Boersma and Hayes, 2001), harmonic (Smolensky and Legendre, 2006), etc. In the orthodox version, the derivation proceeds as follows. There is an input candidate and an infinite set of possible output candidates. There is a finite set of constraints that assign violations to the output candidates, and the candidate that violates the least number of constraints is selected as the surface form.[1] Orthodox OT calculates the winner in terms of strict ranking: constraints are strictly ordered and a single violation of a higher-ranked constraint overpowers any number of violations of a lower-ranked constraint.

In the schematic example below, there is a finite set of constraints ranked left to right. There is an infinite set of candidates given on the left side of each row. Violations are marked with asterisks, and the winning candidate is marked with the pointing hand.

*Thanks to Amy Fountain and Diane Ohala for useful discussion. All errors are my own.

[1]Nothing formally requires that there be a single winning candidate (Idsardi, 1992; Hammond, 1994, 2000), but we can set this issue aside.

(1)

/kæt/	A	B	...	C
☞ [k^{h}æt]	*			*****
[hit]	**!			
[hæk]	**!*			
[čer]	**!			
[vznork]	*	*!		
...	*	*!*		

Notice first that all the candidates violate constraint A; hence only candidates that violate A as little as possible remain in the candidate set: [k^{h}æt], [vznork], and the set denoted by "...". All but the first violate the lower-ranked constraint B, so the first emerges as the winning pronunciation. Notice too that the winning candidate violates constraint C five times, but these violations are irrelevant as higher-ranked constraints have already selected [k^{h}æt] as the winning pronunciation.

The main problem for any implementation of OT is the infinite candidate set (Ellison, 1994; Hammond, 1995; Tesar, 1995). First, if we view the derivation as analogous to the construction of a constraint tableau as above, then surely one step in that process would be to list out the candidates. On the most obvious interpretation of what this would entail, this step would never terminate and we would never be able to proceed to determining violations and selecting a winning candidate.

A second problem is that even if we find a mechanism to get all the candidates listed, we must still assign violations. Again, the simplest interpretation of what this entails is writing in the asterisks in the tableau. Since there are an infinite number of rows, any of which could violate any constraint, the job of determining violations would never end.

In fact, the problem continues on. Even if we could list all the candidates and get all the violations marked, we would still have to calculate which candidate violates the fewest constraints, *modulo* strict ranking or some alternative. This too poses an infinity problem.

There have been several attempts at dealing with these problems. Hammond (1995) and Tesar (1995) propose different finite versions of Gen. Ellison (1994) proposes to represent the candidate set with a finite automaton, thus allowing the infinite set to be represented with finite means. Finally, Karttunen (1998) elaborates the finite-state approach, developing an idea from the first paper.

In this paper, we take another approach. Specifically, we show how the notion of *lazy evaluation* in a functional programming context can be used to treat GEN. The basic idea behind lazy evaluation is that we construct infinite sets which are not immediately evaluated. The programming context is one where we only evaluate such sets when we need to and we only evaluate as much of them as we need in the context at hand. These properties, we will show, allow us an elegant treatment of GEN.

The organization of this paper is as follows. First, we outline how lazy evaluation works, using the functional programming language Haskell (Jones, 2003) as our framework.[2] Next, we provide an implementation of GEN using lazy evaluation to avoid the infinity problems listed above. We then show how the system works for the case of containment-based syllabification. Finally, we consider the issues raised by our implementation for phonological theory.

2 What is Lazy Evaluation?

Lazy evaluation is a concept from functional programming. To understand the former, we must first understand the latter. The basic idea behind functional programming is that a program is a set of functions and constants. A number like `6` or a string like `"phoneme"` are instances of constants. We can also define terms that refer to constants; for example, we might define π as:

(2)
```
myPi = 3.14159265358979
```

or `myName` as:

(3)
```
myName = "Ishmael"
```

The other component of a functional programming language is the notion of a function, something that pairs a set of values with one particular value. Addition is a built-in function: any two numbers are paired with a specific number. For example, the numbers 4 and 7 are uniquely paired with 11. We can also write our own functions as follows.

(4)
```
addTwo   :: Int → Int
addTwo x =  x + 2
```

[2] We choose Haskell for two reasons. First, it is a pure functional language. Second, there are free interpreter and compiler implementations for all platforms (`ghc` and `hugs`). Hutton (2007) is a nice pedagogical introduction to Haskell.

The first line is a specification of the type of elements that this function pairs, in this case, two integers. The second line above defines `addTwo` as a function that, when applied to some number—represented as `x` here—returns that value plus two. For example, we could invoke this function by writing the function name before some constant, e.g. `addTwo 6`, which would produce the result `8` when interpreted.

This is all there is in a strict functional language. Notice, in particular, that there are no separate variables, as one would find in familiar languages like Perl, Java, or C. Constants like `myPi` or `myName` are immutable once defined.[3]

To understand lazy evaluation in a functional programming context, we also need to understand how *lists* work.[4] A list is a sequence of elements terminated by the empty list. When a list is finite, we can represent it in one of two ways, either as a sequence of elements enclosed in square brackets, e.g. `[3,5,7,7]`, or more explicitly as a sequence of elements concatenated with the list construction operator `:`, e.g. `3:(5:(7:(7:[])))`. Both lists terminate with the empty list `[]`, but this is only overt in the latter notation. Notice too that the list construction operator takes a list as its right operand and a single element as its left.

We can manipulate lists in functions. Here is a function that returns the first element of a list:

(5)
```
myHead        :: [a] → a
myHead (x:xs) =  x
```

The function `myHead` returns the first element of a list.[5] It does this by pattern-matching on its argument, requiring that its argument be a list with two parts—a first element `x` and the remaining elements `xs`. The same sort of move can be used to write a function that returns the remaining elements of a list:

[3]One frequent misunderstanding of functions is that they *change* some set of things into another and this would seem to be at odds with the notion of immutability. It is better, therefore, to think of functions as described in the text.

[4]There are a host of other functional data structures with the properties we require, but lists are one of the very simplest, familiar from languages like Lisp, and very frequently used.

[5]Here and following, it is useful to write explicit functions for some functions that are typically already available in the standard Haskell prelude (library). This allows us to be maximally explicit about what our functions do. When we do this, the function we write will begin with the string `my`, e.g. `myHead`, `myTail`, `myConcat`, etc.

(6)
```
myTail        :: [a] → [a]
myTail (x:xs) =  xs
```

Finally, we can write functions that manipulate lists recursively. Here is a more complex function that returns the first n elements of a list:

(7)
```
myTake          :: Int → [a] → [a]
myTake 0 _      =  []
myTake n (x:xs) =  x:myTake (n-1) xs
```

This function takes two arguments, a number and a list. If the number is 0, then the function returns the empty list; it doesn't matter what the second argument is. If the number is greater than 0, then the function returns the first element of the list concatenated with the result of applying the function to the next smaller number and the remainder of the list.

We can show this schematically in steps with the function application `myTake 2 [4,5,6,7]`.

(8)

a. `myTake 2 [4,5,6,7]`

b. `4:(myTake 1 [5,6,7])`

c. `4:(5:(myTake 0 [6,7]))`

d. `4:(5:[])`

e. `[4,5]`

We begin by applying the function with arguments `2` and `[4,5,6,7]`. Remembering that the latter is equivalent to `4:(5:(6:(7:[])))`, the result is the concatenation of `4` with `myTake 1 [5,6,7]`. We then evaluate the embedded `myTake` call, producing `5:myTake 0 [6,7]`. Finally, the last call produces `[]`, and we assemble all the bits into `[4,5]`.

This mode of interpretation is, in fact, the way Haskell proceeds: from outside down through embeddings. Lazy evaluation refers to the fact that evaluation *only occurs when required and only as much as is required*. In the example above, we go through the list argument only as far as necessary; it doesn't matter what follows the number 5, since `myTake` is satisfied at that point.

We can see this with an *infinite* list. Here's how we can define a recursive function that returns an infinite series of numbers:

(9)
```
infnum   :: Int → [Int]
infnum x =  x:infnum (x+1)
```

Consider how this would work when invoked as `infnum 1`.

(10)
a. `infnum 1`

b. `1:(infnum 2)`

c. `1:(2:(infnum 3))`

d. `1:(2:(3:(infnum 4)))`

e. ...

Like `myTake`, the definition of `infnum` is recursive. However, unlike `myTake`, there is no *exit* clause; there is no mechanism to stop the recursion. Thus, invoking this command directly will result in the system trying to produce an infinite series of numbers.

Consider, however, what happens when we invoke `infnum 1` *inside* a call to `myTake`:

(11)
a. `myTake 2 (infnum 1)`

b. `myTake 2 (1:infnum 2)`

c. `1:(myTake 1 (infnum 2))`

d. `1:(myTake 1 (2:infnum 3))`

e. `1:(2:(myTake 0 (infnum 3))`

f. `1:(2:[])`

g. `[1,2]`

We begin with a call to `infnum 1` embedded in a call to `myTake` with the argument `2`. If its first argument is greater than 0, `myTake` requires its second argument be interpretable as a list with a first element: `x:xs`. We must therefore interpret down one more level to see if `infnum 1` can be parsed in this way. Step b above shows that it can. Step c shows that we can then interpret `myTake` one step further, altering its first argument to `1`. We repeat the cycle, producing one more call to `myTake`, but this time with the argument

`0`. Now the definition of `myTake` can return `[]` without interpreting its second argument at all; this is indicated with the single underscore in the definition of `myTake`. The bits are all assembled and the final result is `[1,2]`.

Since this call to `myTake` requires no more than two calls to `infnum`, the process terminates. Even though we are manipulating an object that represents an infinite list, we can do so with impunity since that object is invoked in a context where it only needs to be finitely interpreted. This is lazy evaluation.

3 The Overall Logic

The overall logic for our proposal can now be laid out. First, can we represent the entire candidate set as an infinite list, analogous to `infnum` above? Second, can we represent OT-style constraints as functions that winnow through such a list, like `myTake` above? We consider each of these problems in turn.

The first problem is superficially straightforward. We can generate an infinite list of strings quite easily. First, here is code to generate a list of strings containing only a single symbol. The key new bit here is that strings are themselves lists; thus a string like `"aaa"` is equivalent to `'a':('a':('a':[]))`.

(12)
```
infa   :: String → [String]
infa x =  x:infa ('a':x)
```

The `infa` function is actually quite similar to that for `infnum`. The difference is that the new function adds longer and longer strings to the list, rather than adding larger and larger numbers. If we invoke this as `myTake 5 (infa "")`, then we get: `["","a","aa","aaa","aaaa"]`.

It requires a little more sophistication to get an infinite set of strings over some alphabet. First we need some utility functions:

(13)
```
myMap          :: (a → b) → [a] → [b]
myMap f []     =  []
myMap f (x:xs) =  f x:myMap f xs

myPlusplus           :: [a] → [a] → [a]
myPlusplus [] ys     =  ys
myPlusplus (x:xs) ys =  x:myPlusplus xs ys

myConcat        :: [[a]] → [a]
myConcat []     =  []
myConcat (x:xs) =  myPlusplus x (myConcat xs)
```

The `myMap` function takes a function and a list of elements and applies the function to each element in the list producing a new list.[6] For example, `myMap (+2) [1,6,4]` produces `[3,8,6]`. The `myPlusplus` generalizes the list construction operator and allows us to concatenate two lists. Thus `myPlusplus [1,2] [7,4]` produces `[1,2,7,4]`. Finally, the `myConcat` function generalizes `myPlusplus` to any number of lists. If we invoke `myConcat` like this: `myConcat [[1,2],[4,7],[9,2]]`, this produces `[1,2,4,7,9,2]`.

Let's now look at the code to generate the infinite set of all possible strings over the alphabet $\{a, b, c\}$. First, we define the set of letters:

(14)
```
letters :: String
letters =  "abc"
```

We then define a function that will take a string and return the list of strings formed by prefixing each letter of the alphabet to the string. Thus the invocation `pfx "x"` produces `["ax","bx","cx"]`.

(15)
```
pfx   :: String → [String]
pfx x =  myMap (:x) letters
```

We then generalize `pfx` so that it does the same to lists of strings:

(16)
```
pfxall   :: [String] → [String]
pfxall x =  myConcat (myMap pfx x)
```

Invoking this on `["ax","bx","cx"]` with `pfxall ["ax","bx","cx"]` produces:

[6] Functional programming languages typically allow functions to be used directly as arguments to other functions.

(17) `["aax","bax","cax","abx","bbx","cbx","acx","bcx","ccx"]`

Finally, we write a recursive function over `pfxall` that creates lists of strings, each one produced by applying `pfxall` to the previous list. The function then joins all the results together with `myPlusplus`.

(18)
```
infstrings   :: [String] → [String]
infstrings x =  myPlusplus x (infstrings (pfxall x))
```

Invoking this directly with `infstrings [""]` would produce an infinite list of strings. We can use lazy evaluation and force only the first 30 strings to be produced with a call like this: `myTake 30 (infstrings [""])`. This produces the following output:

(19)
```
["","a","b","c","aa","ba","ca","ab","bb","cb",
"ac","bc","cc","aaa","baa","caa","aba","bba",
"cba","aca","bca","cca","aab","bab","cab",
"abb","bbb","cbb","acb","bcb"]
```

Assuming we can encode a phonological representation as a string of symbols, this establishes that Gen can be formalized in terms of lazy evaluation. Every possible string over the basic segmental vocabulary will be generated by this set of functions.

The phonological representation is, of course, richer than a simple string of segments, but this is not a substantive problem. Haskell, like any other programming language, can accommodate whatever data structures we might wish to define. As long as we can commit ourselves to some coherent structural implementation of any bit of nonlinear phonology, we can implement a lazy Gen using that structure.

Let's now turn to the question of how to winnow through such a set. Basically, we need something to implement Eval over the results of Gen. There are two general strategies. One possibility is to posit a function that checks every element for some property. This check would be a function itself that returned the boolean values `True` and `False`. We can write this function as below:

(20)
```
myFilter          :: (a → Bool) → [a] → [a]
myFilter _ []     =  []
myFilter f (x:xs) =  if f x
                     then x:myFilter f xs
                     else myFilter f xs
```

The `myFilter` function applies the function `f` to every element `x` in a list, keeping that element if `f x` returns `True`. We can use the built-in function `even` to test this with lists of numbers. If we invoke `myFilter` with `even` as in: `myFilter even [4,1,2,7]`, we get `[4,2]`.

Of course, if we use `myFilter` and `even` with an infinite list of numbers, the operation will never terminate. Thus a call like `myFilter even (infnum 1)` goes on forever. We can avoid this, of course, by embedding this call in an invocation of `myTake`: `myTake 10 (myFilter even (infnum 1))`. The latter will return `[2,4,6,8,10,12,14,16,18,20]`.

Something like `myFilter` is fine for constraints that allow for an infinite number of well-formed candidates. For example, a constraint like Ons, which penalizes any syllable that has no onset, produces an infinite set of well-formed candidates. This obtains because the only mechanism for making the candidate set infinite is epenthesis and Ons will let pass all those multiply epenthesized forms where all syllables have onsets.

There are, however, other constraints that cannot be modeled in this way, constraints that reduce the infinite candidate set to a finite subset. For example, a constraint like Fill, which penalizes epenthesis, will rule out any candidate that has epenthetic elements. The remaining candidate set is finite.[7]

To accommodate constraints like Fill, we need something else. Specifically, we must assume that the candidates are sorted, such that as we progress through the set, the number of epenthetic elements increases. Second, we need a function that tests each element for a property, but that terminates as soon as that property is not met. Here is how that would look.

(21)
```
myTakeWhile          :: (a → Bool) → [a] → [a]
myTakeWhile _ []     =  []
myTakeWhile f (x:xs) =  if f x
                        then x:myTakeWhile f xs
                        else []
```

This function takes elements from the front of a list as long as some property is satisfied. Once the property does not hold, no additional elements are taken. The difference from `myFilter` is the `else` clause: in the case of `myFilter`, we invoke the function on the remainder of the list; in the case of

[7] This assumes, of course, that the set of nonlinear structures and possible segments is finite.

`myTakeWhile`, we stop processing and return the empty list. If we invoke the second function on the same list with `myTakeWhile even [4,1,2,7]`, we get `[4]` because the function fails when it reaches the second element of the list.

Consider the different behaviors of these constraints when invoked with a predicate like `(<5)`, which tests for whether its argument is less than 5. We invoke these as below:

(22)
```
myTakeWhile (<5) (infnum 1)
myFilter (<5) (infnum 1)
```

In the first case, the function returns `[1,2,3,4]`. In the second case, the function continues forever. In the case of `myTakeWhile`, the function succeeds with 1 through 4. When it reaches 5, it terminates because 5 fails the test. In the case of `myFilter`, the function succeeds with 1 through 4, fails with 5, but continues on looking for numbers less than 5... which, of course, it will never find.

As functions for winnowing through an infinite list, `myFilter` has the advantage that it will find every element in the string that matches the test. It has the disadvantage that it will look forever if the list is infinite. The `myTakeWhile` function will only work if all the cases to be returned are at the beginning of the list. On the other hand, it has the advantage that it will terminate definitively if the list is sorted appropriately. We need both sorts of functions.

4 A Test Case

To assess whether lazy evaluation along the lines we've sketched here will work, we will take a simple test case: containment-based syllabification (Prince and Smolensky, 1993).

The basic idea behind containment is that the input and output cannot differ in terms of the segments they contain, but only in terms of how the output is syllabified. This restriction on the input-output mapping limits us to adding syllable structure and manipulating how elements in the input are parsed or not parsed by that structure. Since there is no bound on the number of syllables that can be assigned to the output and no restriction against syllables with vacuous terminal nodes, there are an unbounded number of candidate output forms. There is no other mechanism deriving the infiniteness of the candidate set.

Consider, for example a hypothetical input /CV/. We represent syllable boundaries—where necessary—with period (full stop), epenthetic elements with $\mathbb{C}$ or $\mathbb{V}$, and unparsed segments with angled brackets, e.g. ⟨C⟩ or ⟨V⟩. If we exclude epenthetic elements, we have just these four possibilities: [CV], [⟨C⟩V], and [⟨C⟩⟨V⟩]. Following Prince and Smolensky, we only consider syllable parsings that are canonically well-formed, i.e. [.V.], [.CV.], [.VC.], and [.CVC.], ruling out [C⟨V⟩].

If we add in epenthetic elements, the set of possible pronunciations expands infinitely. Let's start with one epenthetic element. Here is an exhaustive list of all 11 syllabifications of /CV/ possible with only one epenthetic segment.

(23)

CV	⟨C⟩V	(C⟨V⟩)	⟨C⟩⟨V⟩
$\mathbb{V}$.CV	⟨C⟩$\mathbb{V}$.V	$\mathbb{V}$C⟨V⟩	$\mathbb{V}$⟨C⟩⟨V⟩
$\mathbb{V}$C.V	⟨C⟩$\mathbb{C}$V	C$\mathbb{V}$⟨V⟩	
CV$\mathbb{C}$	⟨C⟩V$\mathbb{C}$		
CV.$\mathbb{V}$	⟨C⟩V.$\mathbb{V}$		

The relative order of epenthetic elements and unparsed elements or syllable boundaries are not contrastive. Thus [⟨C⟩$\mathbb{V}$.V], [$\mathbb{V}$⟨C⟩.V], and [$\mathbb{V}$.⟨C⟩V] are identical. Notice too that even though [C⟨V⟩] is not itself a legal candidate, we can generate legal candidates from it with epenthesis.

We can continue on in like vein. Here is a table of all 38 candidates with two epenthetic elements.

(24)

CV		⟨C⟩V	(C⟨V⟩)	⟨C⟩⟨V⟩
CV.$\mathbb{V}$.$\mathbb{V}$	$\mathbb{C}$V.CV	$\mathbb{C}$V⟨C⟩.V	$\mathbb{C}$V$\mathbb{C}$.⟨V⟩	$\mathbb{C}$V⟨C⟩⟨V⟩
CV.V$\mathbb{C}$	$\mathbb{C}$VC.V	$\mathbb{V}$.$\mathbb{C}$⟨C⟩V	$\mathbb{V}$.V$\mathbb{C}$⟨V⟩	$\mathbb{V}$C⟨C⟩⟨V⟩
$\mathbb{V}$.CV.V	$\mathbb{V}$.V.CV	$\mathbb{V}$.$\mathbb{V}$⟨C⟩.V	$\mathbb{V}$C.$\mathbb{V}$⟨V⟩	$\mathbb{V}$.$\mathbb{V}$⟨C⟩⟨V⟩
$\mathbb{V}$C.$\mathbb{V}$.V	$\mathbb{V}$.$\mathbb{V}$C.V	$\mathbb{V}$⟨C⟩.V$\mathbb{C}$	$\mathbb{V}$.C$\mathbb{V}$⟨V⟩	
$\mathbb{V}$.CV.$\mathbb{V}$	$\mathbb{V}$C.CV	$\mathbb{C}$⟨C⟩V$\mathbb{C}$	C$\mathbb{V}$C⟨V⟩	
$\mathbb{V}$C.V.$\mathbb{V}$	CV.C$\mathbb{V}$	$\mathbb{V}$⟨C⟩.V.$\mathbb{V}$	C$\mathbb{V}$.$\mathbb{V}$⟨V⟩	
$\mathbb{V}$C.$\mathbb{C}$V	CVC.$\mathbb{V}$	$\mathbb{C}$⟨C⟩V.$\mathbb{V}$		
$\mathbb{V}$.CVC	$\mathbb{V}$C.VC	$\mathbb{V}$C.⟨C⟩V		
C$\mathbb{V}$.$\mathbb{C}$V	C$\mathbb{V}$C.V			
C$\mathbb{V}$.V$\mathbb{C}$	C$\mathbb{V}$.$\mathbb{V}$.V			
C$\mathbb{V}$.V.$\mathbb{V}$				

We will see that it is essential in our account of Gen that the effects

of epenthesis be orderable, though other orderings are possible. Moreover, we order candidates into bins, each of which is finite in size. Ordering by epenthesis as above satisfies both requirements.

Another possibility is building an ordering of candidates on the number of syllables in the candidate. If there are no syllables in the candidate, there is only one: [⟨C⟩⟨V⟩]. If there is a single syllable, then we get the following 14 candidates:

(25)

⟨C⟩⟨V⟩	CV	⟨C⟩V	C⟨V⟩
⟨C⟩⟨V⟩𝕍	CV	⟨C⟩V	C⟨V⟩𝕍
⟨C⟩⟨V⟩ℂ𝕍	CVℂ	⟨C⟩ℂV	C⟨V⟩𝕍ℂ
⟨C⟩⟨V⟩ℂ𝕍ℂ		⟨C⟩Vℂ	𝕍C⟨V⟩
⟨C⟩⟨V⟩𝕍ℂ		⟨C⟩ℂVℂ	ℂ𝕍C⟨V⟩

Since, as we noted above, the relative order of unparsed segments and epenthetic elements is not contrastive, we can exclude unparsed elements from our graphical representations. This is just a matter of presentation, however, as one can reconstruct the number of unparsed elements by comparing candidates with the input. With this assumption, we can convert the table above to the following:

(26)

	CV	V	C
𝕍	CV	V	C𝕍
ℂ𝕍	CVℂ	ℂV	C𝕍ℂ
ℂ𝕍ℂ		Vℂ	𝕍C
𝕍ℂ		ℂVℂ	ℂ𝕍C

With two syllables, we get 112 candidates. Again, we leave out unparsed elements for perspecuity.

(27)

𝕍.𝕍	𝕍.V	V.𝕍
𝕍.ℂ𝕍	𝕍.ℂV	𝕍.ℂ𝕍
𝕍.CV	V.ℂ𝕍	𝕍.𝕍ℂ
𝕍.𝕍C	𝕍.Vℂ	V.𝕍ℂ
𝕍.ℂ𝕍ℂ	𝕍.ℂ𝕍C	𝕍.ℂ𝕍ℂ
𝕍.C𝕍ℂ	𝕍.CVℂ	V.ℂ𝕍ℂ

(28)

$\mathbb{C}\mathbb{V}.\mathbb{V}$	$\mathbb{C}\mathbb{V}.\mathrm{V}$	$\mathbb{C}\mathrm{V}.\mathbb{V}$
$\mathrm{C}\mathbb{V}.\mathbb{V}$	$\mathrm{C}\mathbb{V}.\mathrm{V}$	$\mathrm{C}\mathrm{V}.\mathbb{V}$
$\mathbb{C}\mathbb{V}.\mathbb{C}\mathbb{V}$	$\mathbb{C}\mathbb{V}.\mathbb{C}\mathrm{V}$	$\mathbb{C}\mathbb{V}.\mathrm{C}\mathbb{V}$
$\mathbb{C}\mathbb{V}.\mathrm{C}\mathrm{V}$	$\mathbb{C}\mathrm{V}.\mathbb{C}\mathbb{V}$	$\mathrm{C}\mathbb{V}.\mathbb{C}\mathbb{V}$
$\mathrm{C}\mathbb{V}.\mathbb{C}\mathrm{V}$	$\mathrm{C}\mathrm{V}.\mathbb{C}\mathbb{V}$	$\mathbb{C}\mathbb{V}.\mathbb{V}\mathbb{C}$
$\mathbb{C}\mathbb{V}.\mathbb{V}\mathrm{C}$	$\mathbb{C}\mathbb{V}.\mathrm{V}\mathbb{C}$	$\mathbb{C}\mathrm{V}.\mathbb{V}\mathbb{C}$
$\mathrm{C}\mathbb{V}.\mathbb{V}\mathrm{C}$	$\mathrm{C}\mathbb{V}.\mathrm{V}\mathbb{C}$	$\mathrm{C}\mathrm{V}.\mathbb{V}\mathbb{C}$
$\mathbb{C}\mathbb{V}.\mathbb{C}\mathbb{V}\mathbb{C}$	$\mathbb{C}\mathbb{V}.\mathbb{C}\mathbb{V}\mathrm{C}$	$\mathbb{C}\mathbb{V}.\mathbb{C}\mathrm{V}\mathbb{C}$
$\mathbb{C}\mathbb{V}.\mathrm{C}\mathbb{V}\mathbb{C}$	$\mathbb{C}\mathbb{V}.\mathrm{C}\mathbb{V}\mathrm{C}$	$\mathbb{C}\mathrm{V}.\mathbb{C}\mathbb{V}\mathbb{C}$
$\mathrm{C}\mathbb{V}.\mathbb{C}\mathrm{V}\mathbb{C}$	$\mathrm{C}\mathbb{V}.\mathbb{C}\mathrm{V}\mathrm{C}$	$\mathrm{C}\mathrm{V}.\mathbb{C}\mathbb{V}\mathbb{C}$

(29)

$\mathbb{V}\mathbb{C}.\mathbb{V}$	$\mathbb{V}\mathbb{C}.\mathrm{V}$	$\mathbb{V}\mathrm{C}.\mathbb{V}$
$\mathbb{V}\mathrm{C}.\mathrm{V}$	$\mathbb{V}\mathrm{C}.\mathbb{V}$	$\mathbb{V}\mathbb{C}.\mathbb{C}\mathbb{V}$
$\mathbb{V}\mathbb{C}.\mathbb{C}\mathrm{V}$	$\mathbb{V}\mathbb{C}.\mathrm{C}\mathbb{V}$	$\mathbb{V}\mathbb{C}.\mathrm{C}\mathrm{V}$
$\mathbb{V}\mathrm{C}.\mathbb{C}\mathbb{V}$	$\mathbb{V}\mathrm{C}.\mathbb{C}\mathrm{V}$	$\mathrm{V}\mathbb{C}.\mathbb{C}\mathbb{V}$
$\mathbb{V}\mathbb{C}.\mathbb{V}\mathbb{C}$	$\mathbb{V}\mathbb{C}.\mathbb{V}\mathrm{C}$	$\mathbb{V}\mathbb{C}.\mathrm{V}\mathbb{C}$
$\mathbb{V}\mathrm{C}.\mathbb{V}\mathrm{C}$	$\mathbb{V}\mathrm{C}.\mathbb{V}\mathrm{C}$	$\mathrm{V}\mathbb{C}.\mathbb{V}\mathbb{C}$
$\mathbb{V}\mathbb{C}.\mathbb{C}\mathbb{V}\mathbb{C}$	$\mathbb{V}\mathbb{C}.\mathbb{C}\mathbb{V}\mathrm{C}$	$\mathbb{V}\mathbb{C}.\mathbb{C}\mathrm{V}\mathbb{C}$
$\mathbb{V}\mathbb{C}.\mathrm{C}\mathbb{V}\mathbb{C}$	$\mathbb{V}\mathbb{C}.\mathrm{C}\mathbb{V}\mathrm{C}$	$\mathbb{V}\mathbb{C}.\mathrm{C}\mathrm{V}\mathbb{C}$
$\mathbb{V}\mathrm{C}.\mathbb{C}\mathrm{V}\mathbb{C}$	$\mathbb{V}\mathrm{C}.\mathbb{C}\mathbb{V}\mathrm{C}$	

(30)

$\mathbb{C}\mathbb{V}\mathbb{C}.\mathbb{V}$	$\mathbb{C}\mathbb{V}\mathbb{C}.\mathrm{V}$	$\mathbb{C}\mathbb{V}\mathrm{C}.\mathbb{V}$
$\mathbb{C}\mathbb{V}\mathrm{C}.\mathrm{V}$	$\mathbb{C}\mathbb{V}\mathrm{C}.\mathbb{V}$	$\mathbb{C}\mathrm{V}\mathbb{C}.\mathbb{V}$
$\mathbb{C}\mathrm{V}\mathbb{C}.\mathrm{V}$	$\mathbb{C}\mathrm{V}\mathbb{C}.\mathbb{V}$	$\mathbb{C}\mathbb{V}\mathbb{C}.\mathbb{C}\mathbb{V}$
$\mathbb{C}\mathbb{V}\mathbb{C}.\mathbb{C}\mathrm{V}$	$\mathbb{C}\mathbb{V}\mathbb{C}.\mathrm{C}\mathbb{V}$	$\mathbb{C}\mathbb{V}\mathbb{C}.\mathrm{C}\mathrm{V}$
$\mathbb{C}\mathbb{V}\mathrm{C}.\mathbb{C}\mathbb{V}$	$\mathbb{C}\mathbb{V}\mathrm{C}.\mathbb{C}\mathrm{V}$	$\mathbb{C}\mathrm{V}\mathbb{C}.\mathbb{C}\mathbb{V}$
$\mathbb{C}\mathrm{V}\mathbb{C}.\mathbb{C}\mathrm{V}$	$\mathbb{C}\mathrm{V}\mathbb{C}.\mathrm{C}\mathbb{V}$	$\mathbb{C}\mathrm{V}\mathbb{C}.\mathrm{C}\mathrm{V}$
$\mathbb{C}\mathbb{V}\mathbb{C}.\mathbb{V}\mathbb{C}$	$\mathbb{C}\mathbb{V}\mathbb{C}.\mathbb{V}\mathrm{C}$	$\mathbb{C}\mathbb{V}\mathbb{C}.\mathrm{V}\mathbb{C}$
$\mathbb{C}\mathbb{V}\mathrm{C}.\mathbb{V}\mathbb{C}$	$\mathbb{C}\mathbb{V}\mathrm{C}.\mathrm{V}\mathbb{C}$	$\mathbb{C}\mathrm{V}\mathbb{C}.\mathbb{V}\mathbb{C}$
$\mathbb{C}\mathrm{V}\mathbb{C}.\mathbb{V}\mathbb{C}$	$\mathbb{C}\mathrm{V}\mathbb{C}.\mathrm{V}\mathbb{C}$	$\mathbb{C}\mathrm{V}\mathbb{C}.\mathbb{V}\mathbb{C}$
$\mathbb{C}\mathbb{V}\mathbb{C}.\mathbb{C}\mathbb{V}\mathbb{C}$	$\mathbb{C}\mathbb{V}\mathbb{C}.\mathbb{C}\mathbb{V}\mathrm{C}$	$\mathbb{C}\mathbb{V}\mathbb{C}.\mathbb{C}\mathrm{V}\mathbb{C}$
$\mathbb{C}\mathbb{V}\mathrm{C}.\mathbb{C}\mathbb{V}\mathrm{C}$	$\mathbb{C}\mathbb{V}\mathrm{C}.\mathrm{C}\mathbb{V}\mathrm{C}$	$\mathbb{C}\mathbb{V}\mathrm{C}.\mathrm{C}\mathrm{V}\mathbb{C}$
$\mathbb{C}\mathbb{V}\mathrm{C}.\mathbb{C}\mathrm{V}\mathbb{C}$	$\mathbb{C}\mathrm{V}\mathbb{C}.\mathbb{C}\mathbb{V}\mathrm{C}$	$\mathbb{C}\mathrm{V}\mathbb{C}.\mathbb{C}\mathrm{V}\mathbb{C}$
$\mathbb{C}\mathrm{V}\mathbb{C}.\mathbb{C}\mathrm{V}\mathbb{C}$	$\mathbb{C}\mathrm{V}\mathbb{C}.\mathbb{C}\mathrm{V}\mathrm{C}$	

It will turn out that the latter ordering on syllables is empirically superior to the former ordering based on epenthesis.

Let's now consider the constraints Prince and Smolensky use to manipulate these representations. There are four basic ones:

(31) a. PARSE: Underlying segments must be parsed into syllable structure.

b. FILL: Syllable positions must be filled with underlying segments.

c. ONS: A syllable must have an onset.

d. -COD: A syllable must **not** have a coda.

The first two constraints are *faithfulness* constraints. They serve to limit epenthesis and deletion. The last two constraints are *markedness* constraints and militate for the least marked syllabification.

Ranking is used to get different effects. If a markedness constraint is ranked above a faithfulness constraint, then the lowest-ranked faithfulness constraint will determine whether epenthesis or deletion is used to satisfy that markedness constraint. If, on the other hand, faithfulness constraints are ranked above markedness, then inputs are syllabified as best they can without epenthesis or deletion. Here is an example of the former, where markedness is ranked high and FILL is ranked lowest.

(32)

/V/	ONS	-COD	PARSE	FILL
V	*!			
⟨V⟩			*!	
☞ ℂV				*

Here, the markedness constraints are ranked at the top, meaning that the requirements for an onset and that there not be a coda must be met. The FILL constraint is ranked at the bottom of the hierarchy entailing that these requirements are met by epenthesis. If, instead, PARSE were ranked at the bottom, we'd get deletion instead:

(33)

/V/	ONS	-COD	FILL	PARSE
V	*!			
☞ ⟨V⟩				*
ℂV			*!	

The same logic works in the case of codas with a suitable input. The following two tableaux show how this works for something like /CVC/. First, we see that epenthesis results when -Cod is ranked above a faithfulness constraint and Fill is ranked bottommost.

(34)

	/CVC/	Ons	-Cod	Parse	Fill
	CVC		*!		
	CV⟨C⟩			*!	
☞	CVC𝕍				*

Then we see that deletion results when Parse is at the bottom.

(35)

	/CVC/	Ons	-Cod	Fill	Parse
	CVC		*!		
☞	CV⟨C⟩				*
	CVC𝕍			*!	

Finally, we see that when both faithfulness constraints are ranked above the markedness constraints, markedness violations in the input surface as is.

(36)

	/CVC/	Fill	Parse	Ons	-Cod
☞	CVC				*
	CV⟨C⟩		*!		
	CVC𝕍	*!			

This system provides an account of the fact that, while onsets can be required, codas can never be, and that while codas can be disallowed, onsets never are. In fact, Prince and Smolensky present this as a theorem.

(37) Universally Optimal Syllables
No language may prohibit the syllable .CV. Thus, no language prohibits onsets or requires codas.

The constraints presented have different effects in terms of the finiteness of the candidate set that they might permit. The constraints Parse, Ons, and -Cod permit an infinite candidate set. To see this, note that for each case, there are an infinite number of possible candidates for /CV/ that do not violate the constraint at all.

For Parse, we can generate an infinite number of candidates that are well-formed by suffixing any number of instances of $\mathbb{V}$, e.g. [CV], [CV.$\mathbb{V}$], [CV.$\mathbb{V}$.$\mathbb{V}$], [CV.$\mathbb{V}$.$\mathbb{V}$.$\mathbb{V}$]. etc. The same set suffices for -Cod. For Ons, we generate an infinite set of well-formed candidates by appending the sequence $\mathbb{C}\mathbb{V}$: [CV], [CV.$\mathbb{C}\mathbb{V}$], [CV.$\mathbb{C}\mathbb{V}$.$\mathbb{C}\mathbb{V}$], [CV.$\mathbb{C}\mathbb{V}$.$\mathbb{C}\mathbb{V}$.$\mathbb{C}\mathbb{V}$], etc.

This is impossible for Fill. The only way to generate an infinite candidate set is with epenthesis, but epenthesis gives rise to violations of Fill. Hence, the set of candidates that are well-formed with respect to Fill is finite.

We must therefore distinguish between Fill violations and the other cases in terms of technology analogous to the difference between `myTakeWhile` and `myFilter`. To do this, we generate candidates lazily, binning by the number of syllables: $\{B_0, B_1, B_2, \ldots\}$. Starting at the first bin, we evaluate candidates as usual, selecting a winner—or winners—for that bin, call this $w(B_0)$. We then go on to the next bin and evaluate the candidates there, determining the winner for that bin $w(B_1)$. In the general case, if the winner for some bin $w(B_n)$ is better than $w(B_{n-1})$, we continue on to B_{n+1}. If $w(B_n)$ is not better than $w(B_{n-1})$, then we are done and the winning candidate for the entire set is $w(B_{n-1})$.

We can express this algorithm in (procedural) pseudocode as follows:

(38)
a. Set global winner to null.
b. Go to first bin.
c. Assess violations for all candidates in current bin.
d. Choose winner from current bin.
e. If current winner is better than global winner:
    i. set global winner to current winner, and
    ii. go to next bin, and
    iii. go to (c)

   else end: global winner is winner.

Let's now look at an example. Consider the candidate /VC/ with the constraint ranking Ons $\gg$ -Cod $\gg$ Parse $\gg$ Fill.

The first bin B_0 has no syllables; hence all segments are unparsed and there is only one candidate: $[\langle \mathrm{V}\rangle\langle \mathrm{C}\rangle]$, and it is, of course, the winner, e.g. $w(B_0) = \langle \mathrm{V}\rangle\langle \mathrm{C}\rangle$. We now compare that winner to the winner of B_1. First, we determine $w(B_1)$ as in the following tableau. There are a finite number of candidates, but for convenience not all candidates are given.

(39)

	/VC/	Ons	-Cod	Parse	Fill
	$\mathrm{V}\langle \mathrm{C}\rangle$	*!		*	
	VC	*!	*		
	$\mathbb{C}\mathbb{V}\langle \mathrm{V}\rangle\langle \mathrm{C}\rangle$			**!	**
☞	$\langle \mathrm{V}\rangle \mathrm{C}\mathbb{V}$			*	*

We see that $w(B_1) = \langle \mathrm{V}\rangle \mathrm{C}\mathbb{V}$. We must now compare $w(B_0)$ with $w(B_1)$. This is shown in the following tableau.

(40)

	/VC/	Ons	-Cod	Parse	Fill
	$\langle \mathrm{V}\rangle\langle \mathrm{C}\rangle$			**!	
☞	$\langle \mathrm{V}\rangle \mathrm{C}\mathbb{V}$			*	*

Since $w(B_1)$ wins, we must go on to evaluate the candidates of B_2. Again, there are a finite number of candidates, but too many to display easily, so the following tableau just contains a few of them.

(41)

	/VC/	Ons	-Cod	Parse	Fill
	$\mathrm{VC}\mathbb{V}$	*!			*
	$\mathbb{C}\mathrm{VC}\mathbb{V}\mathbb{C}$		*!		***
	$\langle \mathrm{V}\rangle \mathrm{C}\mathbb{V}\mathbb{C}\mathbb{V}$			*!	***
☞	$\mathbb{C}\mathrm{VC}\mathbb{V}$				**

Again, the winner(s) here must be compared with the previous best candidate(s).

(42)

	/VC/	Ons	-Cod	Parse	Fill
	$\langle \mathrm{V}\rangle \mathrm{C}\mathbb{V}$			*!	*
☞	$\mathbb{C}\mathrm{VC}\mathbb{V}$				**

Since $w(B_2)$ wins, we must go on to consider B_3. Once again, only representative candidates are given (though the full number is, of course, finite).

(43)

	/VC/	Ons	-Cod	Parse	Fill
☞	ℂ𝕍ℂ𝕍ℂ𝕍				****
☞	ℂ𝕍ℂ𝕍ℂ𝕍				****
	ℂ𝕍ℂ𝕍ℂ		*!		***

Here two candidates tie for $w(B_3)$, so both must be compared with the previous winner.

(44)

	/VC/	Ons	-Cod	Parse	Fill
☞	ℂ𝕍ℂ𝕍				**
	ℂ𝕍ℂ𝕍ℂ𝕍				***!*
	ℂ𝕍ℂ𝕍ℂ𝕍				***!*

Here $w(B_2)$ wins out over the candidates from $w(B_3)$ and Eval terminates with [ℂ𝕍ℂ𝕍] as the overall winning candidate.

The general procedure involves considering the candidate set in increments determined by syllable structure. At each stage only a finite number of candidates need be considered and the procedure only goes on to the next stage if the last stage produces the best candidate up to that point.

Other binning logic would not fare as well. Consider again binning by the number of instances of epenthesis: B_0 would have no epenthesis, B_1 only one instance, and so on. The problem here is that sometimes a single instance of epenthesis will worsen a candidate and only a second instance of epenthesis will improve it.

Axininca Campa (Spring, 1990; McCarthy and Prince, 1993) provides a concrete example. Axininca stems must satisfy a prosodic minimum in certain contexts. This prosodic minimum must occasionally be achieved by multiple instances of epenthesis. For example, the root *na* 'carry on shoulder' is realized as [na𝕋𝔸] with the suffix string [piro𝕋aanc$^{\text{h}}$i].[8]

The precise constraints that force this are not the issue. Basically, words must be at least feet and feet must be at least binary. On such an analysis, a single instance of epenthesis provides no improvement. More concretely, let's assume an analysis with the following constraints:

(45) -Cod ≫ Ons ≫ FtBin ≫ Parse ≫ Fill

[8] In our discussion of Axininca, we will follow McCarthy and Prince in representing epenthetic elements as 𝕋 and 𝔸, rather than as ℂ and 𝕍.

The FtBin constraint forces feet to be binary.

Let's first consider how syllabic bins get the correct result. At B_0, we have only one candidate $w(B_0) = \langle \text{n} \rangle \langle \text{a} \rangle$. At B_1, we have:

(46)

/na/	-Cod	Ons	FtBin	Parse	Fill
☞ na			*		
naT	*!		*		*
⟨n⟩aT	*!	*	*	*	*

We then compare $w(B_0)$ with $w(B_1)$.

(47)

/na/	-Cod	Ons	FtBin	Parse	Fill
☞ na			*		
⟨n⟩⟨a⟩			*	*!*	

Since $w(B_1)$ is better than $w(B_0)$, we go on to B_2.

(48)

/na/	-Cod	Ons	FtBin	Parse	Fill
☞ naTA					**
AnaT	*!	*			**
naA		*!			*

We must now compare $w(B_1)$ and $w(B_2)$.

(49)

/na/	-Cod	Ons	FtBin	Parse	Fill
☞ naTA					**
na			*!		

Since $w(B_2)$ wins, we must go on to determine $w(B_3)$.

(50)

/na/	-Cod	Ons	FtBin	Parse	Fill
☞ naTATA					****
⟨n⟩aTATA		*!		*	****
⟨n⟩⟨a⟩TATATA				*!*	******

Finally, we compare $w(B_2)$ with $w(B_3)$:

(51)

	/na/	-Cod	Ons	FtBin	Parse	Fill
☞	na𝕋𝔸					**
	na𝕋𝔸𝕋𝔸					***!*

Since $w(B_2)$ wins, we are done and have gotten the desired result.

If we were to bin by number of instances of epenthesis, we would not get the correct result. Let's go through the same derivation with bins by epenthesis to see this. First, we consider B_0. Notice that the number of syllables is not controlled in this bin, only the number of instances of epenthesis.

(52)

	/na/	-Cod	Ons	FtBin	Parse	Fill
☞	na			*		
	⟨n⟩a		*!	*	*	
	⟨n⟩⟨a⟩			*	*!*	

We now go on to B_1, where every candidate has a single instance of epenthesis.

(53)

	/na/	-Cod	Ons	FtBin	Parse	Fill
	na𝕋	*!		*		*
	𝔸na		*!	*	*	*
☞	𝕋⟨n⟩a			*	*	*

We now compare $w(B_0)$ with $w(B_1)$:

(54)

	/na/	-Cod	Ons	FtBin	Parse	Fill
☞	na			*		
	𝕋⟨n⟩a			*	*!	*

Here, $w(B_0)$ wins, so the algorithm terminates with the incorrect result. The problem is that we have found a local minimum in optimality before we reached B_2. Hence, if we have captured the essential properties of the Axininca analysis correctly, binning by the number of instances of epenthesis is empirically inadequate.

On the other hand, we have seen that for containment-based OT, syllable-based binning works with lazy Gen.

5 Remaining issues

There are several remaining issues.

One issue is whether the model generalizes to correspondence-based Optimality Theory (McCarthy and Prince, 1995). In this version of OT, anything can change from input to output and we are not bound to containment.

Correspondence-based OT is not a problem for lazy Gen. We've already seen in Section 3 that we can generate an infinite candidate set over any finite alphabet. If we syllabify those candidates, we can just as easily bin them by the number of syllables each candidate contains. If, on the other hand, the alphabet is not finite, then there would indeed be a problem.

A second general question concerns the nature of the bins. Syllable-based bins will not generalize to other domains, e.g. morphology or syntax. We are not committed to syllable-based bins for every possible domain of grammar. It may very well be that bins in other domains are empirically determined. We are committed to the position that some binning will work in all other domains, because lazy Gen requires bins.

A third remaining issue is whether syllable-based bins are adequate for phonological theory. The prediction of syllable-based bins is that the best candidate will never be more than a bin further along than the previous best candidate. In more formal terms, we cannot have a situation whether $w(B_n)$ is the true winner, but $w(B_{n-1})$ loses to $w(B_{n-2})$.

What would such a case look like? Imagine a language like Axininca, but where the optimal candidate must be at least *three* syllables long, i.e. [na𝕋𝔸𝕋𝔸]. Syllable-based bins with lazy Gen would not find this candidate.

References

Boersma, Paul. 1997. How we learn variation, optionality, and probability. ROA #221.

Boersma, Paul, and Bruce Hayes. 2001. Empirical tests of the Gradual Learning Algorithm. *Linguistic Inquiry* 32:45–86.

Ellison, T. Mark. 1994. Phonological derivation in Optimality Theory. *COLING 94* 1007–1013.

Hammond, Michael. 1994. An OT account of variability in Walmatjari stress. ROA #20.

Hammond, Michael. 1995. Syllable parsing in English and French. ROA #58.

Hammond, Michael. 2000. The logic of OT. *WECOL* 28:146–162.

Hutton, Graham. 2007. *Programming in Haskell*. Cambridge: Cambridge University Press.

Idsardi, William J. 1992. The computation of prosody. Doctoral Dissertation, MIT.

Jones, S.L.P. 2003. *Haskell 98 language and libraries: The revised report*. Cambridge: Cambridge University Press.

Karttunen, Lauri. 1998. The proper treatment of optimality in computational phonology. In *The proceedings of FSMNLP '98: International workshop on finite-state methods in natural language processing*, 1–12. Ankara, Turkey: Bilkent University.

McCarthy, John, and Alan Prince. 1993. Prosodic morphology. U. Mass.

McCarthy, John, and Alan Prince. 1995. Faithfulness and reduplicative identity. In *Papers in Optimality Theory*, ed. J. Beckman, L. Dickey, and S. Urbanczyk, volume 18 of *U. Mass. Occasional Papers in Linguistics*, 249–384. [ROA].

Prince, Alan, and Paul Smolensky. 1993. Optimality Theory. U. Mass and U. of Colorado.

Smolensky, P., and G. Legendre. 2006. *The harmonic mind: From neural computation to optimality-theoretic grammar*. Cambridge: MIT Press.

Spring, Cari. 1990. Implications of Axininca Campa for prosodic morphology and reduplication. Doctoral Dissertation, University of Arizona.

Tesar, Bruce. 1995. Computational optimality theory. Doctoral Dissertation, University of Colorado at Boulder.

A Notes on the Implementation

The proposal outlined in the text is implemented in Haskell here as a demonstration proof that the system works.

This paper is written in *literate Haskell* style, which means that the code and the paper derive from the same source code. This file thus constitutes the working code and the paper that describes it.

For convenience, unparsed elements are not indicated in output forms. Thus one candidate output for input /hat/ is [ha$\mathbb{C}$], with an unparsed [t] and an epenthetic [$\mathbb{C}$]. This would be equivalent to [ha$\langle$t$\rangle\mathbb{C}$] or [ha$\mathbb{C}\langle$t$\rangle$]. It is possible to reconstruct the number of unparsed elements in an output by comparing it with the input and this is how the implementation of Parse works below.

The program can be invoked from within the `ghci` or `hugs` interpreters by calling the function `eval` with two arguments. The first argument is the input form in double quotes. The second argument is an ordered list of constraints in square brackets, separated by commas. For example:

(55) `eval "hat" [onset,nocoda,fill,parse]`

Alternatively, the program can be compiled with the `ghc` compiler and run on the command-line like this:

(56) `./lazy hat onset nocoda fill parse`

Lastly, the program can be run in a one-off mode with `runhaskell` like this:

(57) `runhaskell lazy.lhs hat onset nocoda fill parse`

B Implementation

```
import List (isPrefixOf)
import System.Environment (getArgs)

--constraints make a number from an input and and output
type Constraint = String → String → Int

--a candidate is a string and a vector of violations
```

```
type Candidate  = (String,[Int])

--to run the program on its own
main = do as ← getArgs
          if (length as) < 2
            then error "usage: lazy input c1 c2 c3..."
            else do let i = head as
                    let cs = map convert $ tail as
                    putStr $ unlines $ map fst $ eval i cs

--converts strings to constraints
convert          :: String → Constraint
convert "onset"  =  onset
convert "nocoda" =  nocoda
convert "fill"   =  fill
convert "parse"  =  parse
convert "ftbin"  =  ftbin
convert x        =  error (x ++ " is not a constraint name")

--entry function for eval
eval      :: String → [Constraint] → [Candidate]
eval i cs =  evl [] $ map (makeCanVecs i cs) (gen i)

--evaluates bin by bin, called by eval
evl               :: [(String, [Int])] → [[(String, [Int])]]
                     → [(String, [Int])]
evl [] (y:ys)     =  evl (getBest [] y) ys
evl (x:xs) (y:ys) =  if rank (>) (head best) x
                       then (x:xs)
                       else if rank (==) (head best) x
                              then evl ((x:xs) ++ best) ys
                              else evl best ys
                       where best = getBest [] y

--gets the highest-ranked candidates from a set
getBest            :: [Candidate] → [Candidate] → [Candidate]
getBest xs []      =  xs
getBest [] (y:ys)  =  getBest [y] ys
getBest (x:xs) (y:ys)
   | rank (==) x y =  getBest (y:x:xs) ys
```

```
   | rank (<) x y  =  getBest (x:xs) ys
   | otherwise     =  getBest [y] ys

--compares the ranking of two candidate,vector pairs
rank       :: ([Int] → [Int] → Bool) → Candidate →
              Candidate → Bool
rank c a b =  c (snd a) (snd b)

--makes a set of candidate,vector pairs for a bin
makeCanVecs             :: String → [Constraint] → [String] →
                           [Candidate]
makeCanVecs _ _ []      =  []
makeCanVecs i xs (c:cs) =  (c,makeVec i xs c):makeCanVecs i xs cs

--makes a vector of violations for a ranked set of constraints
makeVec            :: String → [Constraint] → String → [Int]
makeVec _ [] _     =  []
makeVec i (x:xs) c =  x i c:makeVec i xs c

--NOCODA constraint
nocoda      :: Constraint
nocoda _ "" =  0
nocoda i c  =  if length c > 1 ∧ c!!1 == '.' ∧ isConsonant (c!!0)
               then 1 + (nocoda i (tail c))
               else nocoda i (tail c)

--ONSET constraint
onset      :: Constraint
onset _ "" =  0
onset i c  =  if length c > 1 ∧ c!!0 == '.' ∧ isVowel (c!!1)
              then 1 + (onset i (tail c))
              else onset i (tail c)

--PARSE constraint
parse     :: Constraint
parse i c =  (length i) - (((length c) - (fill i c)) -
             (count "." c))

--FILL constraint
fill     :: Constraint
```

```
fill _ c =  (count "V" c) + (count "C" c)

--FTBIN constraint
ftbin     :: Constraint
ftbin _ c =  if (count "." c) > 2 then 0 else 1

--counts how many times something occurs in a string
count      :: String → String → Int
count _ "" =  0
count p s  =  if isPrefixOf p s then 1 + (count p (tail s))
                                else count p (tail s)

--creates the infinite candidate set
gen   :: String → [[String]]
gen w =  gn w 0 where gn w n = makeBin w n:gn w (n+1)

--makes a single syllable bin
makeBin     :: String → Int → [String]
makeBin w n =  concat $ map (makeSyl w) (polysyllables n)

--makes all parses of an input for a single template
makeSyl      :: String → String → [String]
makeSyl _ "" =  [""]
makeSyl w s  =  map fst $ doAllSubs ((length s)-1) [(s,w)]

--does n substitutions in a list of templates+inputs
doAllSubs      :: Int → [(String,String)] → [(String,String)]
doAllSubs 0 ps =  concat $ map (doSubs 0) ps
doAllSubs n ps =  concat $ map (doSubs n) (doAllSubs (n-1) ps)

--makes all substitutions for a particular position in template
doSubs          :: Int → (String,String) → [(String,String)]
doSubs n (ps,w) =  (ps,w):map makePairs fixedBits
                   where
                   makePairs x = (makeSub (fst x) ps n,snd x)
                   fixedBits = filter
                                  (segType (ps!!n) ∘ fst)
                                  theBits
                   theBits = map (bits w) [0..(length w)-1]
```

```
--substitutes an indexed character in a string
makeSub       :: Char → String → Int → String
makeSub c s n =  (take n s) ++ [c] ++ (drop (n+1) s)

--gets segment type by C,V
segType       :: Char → Char → Bool
segType 'C' x =  isConsonant x
segType 'V' x =  isVowel x
segType '.' _ =  False

--returns the nth character plus remainder of the string
bits        :: String → Int → (Char,String)
bits w n =  (w!!n,drop (n+1) w)

--tests for consonanthood
isConsonant :: Char → Bool
isConsonant =  not ∘ isVowel

--tests for vowelhood
isVowel   :: Char → Bool
isVowel v =  elem v vowels

--set of recognized vowels
vowels :: String
vowels =  "Vaeiou"

--set of possible syllables
syllables :: [String]
syllables =  words "V CV VC CVC"

--generates the set of polysyllabic shapes
polysyllables   :: Int → [String]
polysyllables 0 =  [""]
polysyllables n =  map (++".") (polysyls n)

--called by polysyllables to make the shapes
polysyls   :: Int → [String]
polysyls 0 =  [""]
polysyls n =  concat (map sylPfx (polysyls (n-1)))
```

```
--prefixes all syllable types to a shape
sylPfx   :: String → [String]
sylPfx x =  map ((x++".")++) syllables
```