A Haskell Implementation of the CYK Algorithm

Aaron Gorenstein

February 15, 2014

This is a brief "literate program" written in Haskell implementing the CYK algorithm. I assume the reader is already familiar with the CYK algorithm. My implementation is very simplistic—this is unavoidable as I am a Haskell novice. This has not deterred me, because my target audience is other novices! In particular, I'm eager to share what I found to be the "natural" way of expressing the CYK algorithm in Haskell, especially in comparison to my C++ implementation. In many ways it looks quite similar, but it seems to more faithfully reflect the recursive structure our dynamic algorithm counts on.

There are two main sections in this document. The first details the "foundation". There we define how to express a CFG that is in CNF, and some reverse-lookup functions. That is, for a string of symbols α in the gammar G, is there a production in G of the form $A \to \alpha$? If so, return all such A. The second section focuses entirely on the main CYK algorithm, and focuses on explaining how our Haskell code reflects the "imperative" definition as found in the C++ version. A final epilogue simply shows a very simple instatiation of the algorithm on a grammar and input string.

Without further ado:

1 Prologue: Building the Foundation

Our code begins in a very staid manner.

```
1  import Data.Array -- for Array
2  import Data.List  -- for nub
```

The function `nub` takes a list and removes redundant elements. For instance, the list $[1, 2, 3, 2, 3]$ is transformed to $[1, 2, 3]$.

This code assumes that the grammar is in Chomsky Normal form, the definition of which I will not address here—it's very standard. Regardless, we're able to define our grammar productions as having exactly two forms (tuples, really), as we know that each production involves exactly 2 or 3 symbols.

```
1  data Production = NTprod String String String | Tprod String String
2  type CFG = [Production]
```

The datatype `Production` is either a *nonterminal* production (`NTprod`) of the form $A \to BC$ (observe that the right-hand-side is made of *non*terminals), or a *terminal* (`Tprod`) production, of the form $A \to a$, where a is a *terminal*. We specify these options very concretely with Tuples of the associated string representations.

A context-free grammar is simply a collection of these productions, hence the naïve definition in list form in line 2 of the above code snippet. Observe that many things about the grammar are not checked (that the start symbol is not used recursively, for instance).

With our grammar object defined, the only remaining feature we want to define is the "reverse-lookup": given a string of symbols from the gammar α, return any nonterminal A such that the production $A \to \alpha$ exists. As we know our grammar is in CNF, there are exactly two cases: either α is two nonterminals, or α is a single terminal. To facilitate implementation, first we want to be able to, given a `Production` of either sort, extract its left-hand-side.

```
1  prodLHS (NTprod a _ _) = a
2  prodLHS (Tprod a _) = a
```

This is an obvious utility.

Now we can do the reverse lookup for the two cases. They're obviously very similar. Open question: is there a nicer way of implementing this, perhaps as a single function? Regardless, here is the $A \to BC$ lookup (given BC, find all A such that $A \to BC$):

```
1  ntGens :: CFG → (String, String) → [String]
2  ntGens cfg (a, b) = map prodLHS (filter filterFunc cfg)
3      where filterFunc (NTprod _ x y) = x == a && y == b
4            filterFunc _ = False
```

Here is the $A \to a$ lookup (given a, find all A such that $A \to a$).

```
1  termGens :: CFG → String → [String]
2  termGens cfg b = map prodLHS (filter filterFunc cfg)
3      where filterFunc (Tprod _ y) = y == b
4            filterFunc _ = False
```

Those are the only data types and functions we need! We are now ready to define our main algorithm.

2 The Main Algorithm

Let's jump right into the main algorithm, and we shall explain it line-by-line:

```
1  cykMatrix :: CFG → String → Array (Int, Int) [String]
2  cykMatrix cfg s =
3      let termGens' = termGens cfg
4          ntGens' = ntGens cfg
5          n = length s
6          m = array ((0,0), (n-1,n-1))
7                  ([((i,i), termGens' [s!!i])      | i ← [0..n-1]] ++
8                   [((r, r+l), generators r l) | l ← [1..n-1], r ← [0..n-l-1]])
9                     where generators :: Int → Int → [String]
10                          generators r l =
11                            nub $ concat [ntGens' (a,b) | t ← [0..l-1],
12                                                          a ← m!(r,r+t),
13                                                          b ← m!(r+t+1,r+l)]
14     in m
```

Lines 1-6 are just setting up the function—the real magic starts afterwards. Recall from the CYK algorithm design that $m!(i,j)$ is the set of all nonterminals which generates the substring of s starting at index i and ending at j. The base case, as shown on line 7, follows naturally: at $m!(i,i)$, the nonterminals are exactly those which generate the single terminal found at location i in s. (The curious $[s!!i]$ notation is simply because $s!!i$ is a character, and for type-safety we have to "promote" back to a String type, which we do with the [] notation.) On line 8, we say more-or-less the same thing, conceptually, but in the general case.

Thus, it is entirely the list-comprehension beginning on line 11 where the "magic" happens. It states: The nonterminals which can generate the substring from index r to $r+\ell$ are exactly those C involved in the production $C \to AB$, where AB are represented by (a,b) in the `ntGens'` call. Moreover, for any a, b, it must be that a generates some substring-prefix $s[r, r+t]$ and b generates the corresponding suffix $s[r+t+1, r+\ell]$. The values t can exist between 0 and $\ell - 1$. Quite a lot in packed into that single line!

I hope that was somewhat comprehensible. Note that compared with the C++ version, that single list comprehension replaces 3 `for` loops! Most importantly, I think the list comprehension better expresses the relationships between those three integers r, t, l. Hooray!

3 Epilogue: A Small Execution

That was the major part of the work. At this point we'll simply present a hard-coded example. In the future this code may be extended to actually take things from input files and so forth, but for now this is just a quick test.

First, we use the matrix to actually compute our decision algorithm:

```
1  cyk cfg s = let n = length s in
2      "S" `elem` cykMatrix cfg s !(0,n-1)
```

Now let's define a very simple grammar:

```
1  exampleCFG :: CFG
2  exampleCFG = [NTprod "S" "A" "B",
3                NTprod "A" "A" "A",
4                Tprod "A" "a",
5                NTprod "B" "B" "B",
6  --             (NTprod "B" "A" "A"),
7                Tprod "B" "b"]
```

And here is the easy way of using our CYK algorithm!

```
1  main = print (cyk exampleCFG "aaabb")
```

Thank you for reading, I hope this has helped you understand something about Haskell or the CYK algorithm. I also hope I have been able to share my excitement and enthusiasm with how different programming languages can sometimes express things in intellectually pleasing contrasting ways.