Ling 185A: RE conversion project

Dylan Bumford

due: Fri, Mar 22

1 Converting regular expressions to deterministic machines

In class, and in HW4, we wrote an algorithm to convert a regular expression to a finite state machine. That machine, however, used "epsilon-transitions" to move freely from some states to others without consuming any of the input. This prevents us from using the normal FSA parsing functions like `walk` and `accepts`, which only take steps when they consume characters. So to get a usable machine from the regular expression, we had to subsequently convert the `EpsFSA` to a real `FSA`.

This project invites you to consider an alternative method for converting `Regexp` directly to an `FSA` without wasting time with epsilon-transitions. In fact, we will convert the `Regexp` directly to a **deterministic** `FSA`: one in which there is always exactly one choice of which state to transition to.

2 Semiring equations

Recall that a type is a monoid if there is an operation `(<>)` on its values that is associative and has an identity element.

```
class Monoid a where
  idty :: a
  (<>) :: a -> a -> a
```

- `(x <> y) <> z == x <> (y <> z)`
- `idty <> x == x`
- `x <> idty == x`

And a monoid is a semiring if there is an operation `summarize` that its monoid operation distributes over. That is, for any value `k` and list `vals`, the equations on the right hold.

```
class Semiring a where
  summarize :: [a] -> a
```

- `summarize [k <> x | x <- vals] == k <> summarize [x | x <- vals]`
- `summarize [x <> k | x <- vals] == summarize [x | x <- vals] <> k`

Think of adding up a bunch of numbers each multiplied by `k`; you could simplify your life by just adding the numbers straight up and then multiplying the total by `k`.

The distributivity law tells you more than meets than eye. Let's call the result of summarizing an empty list `empty`, so `empty = summarize []`. What happens when you apply the monoid operation to `(<>)` to a value `k` by `empty`: `k <> empty == ...` Well, a little reasoning shows that *for any semiring whatsoever*:

```
   k <> empty
== k <> summarize []            -- by definition of empty
== summarize [k <> x | x <- []] -- by distributivity
== summarize []                 -- by definition of list comprehension
== empty
```

For instance, in the boolean semiring, `(<>) = (&&)` and `summarize = or`. So `empty == or [] == False`. And indeed, `k && False == False` no matter what `k` is.

Similarly,

```
   empty <> k
== summarize [] <> k            -- by definition of empty
== summarize [x <> k | x <- []] -- by distributivity
== summarize []                 -- by definition of list comprehension
== empty
```

3 Regular semirings

We saw that lists form a monoid under concatenation:

```
instance Monoid [a] where
  idty = []
  xs <> ys = xs ++ ys
```

For any list `xs`, concatenating `xs` with `[]` just returns the same list `xs`. And for any lists `xs`, `ys`, and `zs`, it doesn't matter if you glue `xs` to `ys` and then the result to `zs`, or glue `xs` to the result of gluing `ys` to `zs`. You get the same big list either way.

Regular expressions also form a monoid under concatenation:

```
instance Monoid Regexp where
  idty = undefined
  r <> s = r <.> s
```

That is, any string that matches `(r <.> s) <.> t` also matches `r <.> (s <.> t)`, and vice versa. Which regular expression is the identity element for `(<.>)`? This expression `i` should have the property that `i <.> r` matches all the same strings as `r` does (and likewise for `r <.> i`).

It turns out, regular expressions also form a semiring! Find a definition for `summarize` that makes the semiring laws true:

```
instance Semiring Regexp where
  summarize []       = undefined
  summarize (re:res) = undefined
```

You should check your definition by constructing some simple `Regexp`s and testing that `<.>` distributes over your definition of `summarize`. For instance:

```
ghci> (r, s) = (str "a", str "b" <|> str "c")
ghci> k = rep (str "d")
ghci> dist u = match (k <> summarize [r, s]) u == match (summarize [k <> r, k <> s]) u
ghci> all dist ["a", "b", "ddd", "ddc", "addb", "ddda"]
```

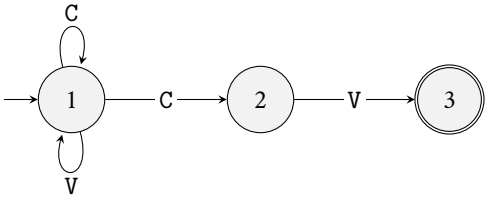
Notice that the `(<.>)` defined in `Regexp.hs` takes some shortcuts when building complex expressions. Those shortcuts are just exactly the algebraic facts discussed here. Which lines take advantage of the fact that `(<.>)` is the monoid action for `Regexp`, and which take advantage of the fact that `Regexp` is a semiring?

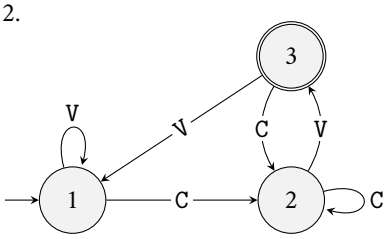
```
{- Provide your answer in this comment here

-}
```

4 Warm up

Our encoding of FSAs is **nondeterministic**: a character `c` may lead to several (or no) next states. For instance, in the simple FSA on the left, from state 1, a C can transition to state 1 or state 2.





In a **deterministic** automaton, things are simpler. For any character `c` and state `q`, there is exactly one transition leaving `q` and consuming `c`, so you always know exactly where to go next. For instance, the automaton above on the right is equivalent to the one on the left, but at every state there's exactly one transition per character. The transitions of such a machine may be encoded as a *function* rather than a *table*. And there can only be one start state. Compare the data types of `FSA` and `DFA` below.

```
type State = Int
type TransL = [(State, Char, State)]
data FSA =
 FSAWith [State] [Char] [State] [State] TransL
 --        Q       Σ       I       F       Δ
```

```
type State = Int
type TransF = (State, Char) -> State
data DFA =
 DFAWith [State] [Char] State [State] TransF
 --        Q       Σ       I       F       Δ
```

For instance, the deterministic machine above might be encoded as:

```
myDFA = DFAWith qs ss q0 fs tf where
  qs = [1,2,3]
  ss = "CV"
  q0 = 1
  fs = [3]
  tf = \(q,c) -> case (q,c) of
    (1, 'C') -> 2; (1, 'V') -> 1
    (2, 'C') -> 2; (2, 'V') -> 3
    (3, 'C') -> 2; (3, 'V') -> 1
```

To write a parser for a deterministic machine, we replace all of the *nondeterinistic* state transitions of the form `State -> [State]` with *ordinary functions* of the form `State -> State`. Test your definitions by parsing a few strings with `myDFA` (or any other DFA you define).

```
-- FSA parsing functions designed to transit
-- from one state to a list of next states

return :: a -> [a]
return = \a -> [a]

(>=>) :: (a -> [b]) -> (b -> [c]) -> (a -> [c])
f >=> g = \a -> [c | b <- f a, c <- g b]

step :: TransL -> Char -> State -> [State]
step trs c =
  \q -> [r | (s,x,r) <- trs, s == q, x == c]

walk :: TransL -> String -> State -> [State]
walk trs ""     = return
walk trs (c:cs) = step trs c >=> walk trf cs

accepts :: FSA -> String -> Bool
accepts (FSAWith _ _ is fs ds) u =
 or [elem qf fs | qi <- is, qf <- walk ds u qi]
```

```
-- DFA parsing functions designed to transit
-- from one state to exactly one next state

quit :: a -> a
quit = undefined

(>->) :: (a -> b) -> (b -> c) -> (a -> c)
f >-> g = undefined

dstep :: TransF -> Char -> State -> State
dstep tf c =
  undefined

dwalk :: TransF -> String -> State -> State
dwalk tf ""     = undefined
dwalk tf (c:cs) = undefined

daccepts :: DFA -> String -> Bool
daccepts (DFAWith _ _ q0 fs tf) u =
  undefined
```

Eventually, you'll need to be able to add new transitions to a partially built-up `TransF` function. Might as well write that auxiliary function now. Given a function `f`, a key `k` and a value `v`, return a new function that is just like `f` except that when its argument is equal to `k` then it returns `v` (otherwise it just returns whatever it would have returned before).

```
extend :: Eq k => (k -> v) -> k -> v -> (k -> v)
extend f k v = \k' -> undefined
```

5 The Idea

Here is the main idea for the conversion procedure we'll follow. We are going to build an DFA whose states are regular expressions. The initial state will be the regular expression we are aiming to convert, and each transition will take us from a one regular expression to another. What should the transitions be? Well, imagine we're at a regular expression `re`, and we then consume a character `c`. What regular expression should we move to? Answer: one that is just like `re`, except that it just matches the stuff that can come after `c`.

For example, say we know that

```
ghci> mset re
["cat", "catt", "cattt", "dog", "dogg", "doggg"]
```

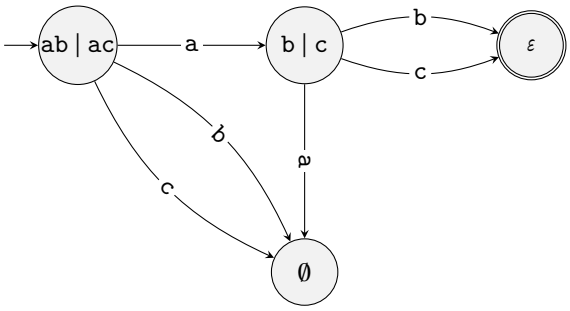
Then after consuming `c`, we would want to transition to an expression `re'` such that

```
ghci> mset re'
["at", "att", "attt"]
```

That is, it matches all of the `c`-suffixes of strings that `re` matched. Or put another way, if `re` matches a string that starts with `c`, then `re'` should match the tail of that string: if `match re (c:u) == True`, then `match re' u == True`. In an equation:

$$[\![\texttt{re}']\!] = \{x_1 \cdots x_n \mid \texttt{c}x_1 \cdots x_n \in [\![\texttt{re}]\!]\}$$

To build out the machine, we'll just need to iterate this process, consecutively reducing regular expressions one character at a time, so to speak. Here's a picture:



The original RE `(ab|ac)` matches `"ab"` and `"ac"`. After reading `'a'`, it transitions to the RE `(b|c)`, which matches `"b"` and `"c"`, since these are the strings that can come after `'a'`. However, after consuming either `'b'` or `'c'`, it transitions to `∅`, which doesn't match anything because there are no matches to `(ab|ac)` that begin with either `'b'` or `'c'`. Then the process repeats at `(b|c)`. This RE matches `"b"` and `"c"`. So after reading either `'b'` or `'c'`, it transitions to `ε`, which matches the empty string. This makes sense, since there are no more characters left to consume (the empty string is what follows `'b'` and `'c'` in `"b"` and `"c"`).

More generally, we should be able to stop at any regular expression that accepts the empty string. So write a function `emptyOK` that returns `True` if its argument matches the empty string, and `False` if it does not.

```
emptyOK :: Regexp -> Bool
emptyOK re = case re of
  Zero    -> undefined
  One     -> undefined
  Lit _   -> undefined
  Alt r s -> undefined
  Cat r s -> undefined
  Star r  -> undefined
```

```
ghci> emptyOK (One <|> str "ab")
True
ghci> emptyOK (str "ab" <|> str "ac")
False
ghci> emptyOK (str "a" <|> rep (str "b"))
True
```

Remember you can check your results for an arbitrary regular expression `re` by comparing `emptyOK re` to `match re ""`.

6 Remainders

The most important component of this procedure is the function that reduces a regular expression by a single initial character. We'll call this the `remainder` of `re` after `c`. Given a regular expression `re` and a character `c`, it should return a new regular expression that matches exactly the strings that follow `c`, according to `re`.

Start with the base cases. Let's say `re` is `Zero`. This matches nothing, so there are no strings that come after `c`, i.e., no `c`-remainders. So the result is still `Zero`. Moving on, `One` matches only the empty string. Since this does not start with `c`, there are again no `c`-remainders, so the result is also `Zero`. How about `Lit a`? This matches only the string whose only character is a. If a is `c`, then there is exactly one `c`-remainder: the empty string (since `"c" = 'c':""`). But if a is not `c`, then there are again no `c`-remainders.

```
remainder :: Regexp -> Char -> Regexp
remainder re c = case re of
  Zero    -> Zero
  One     -> Zero
  Lit a   -> undefined
```

Now things get interesting. Every string that matches `Alt r s` is either an `r`-type string or an `s`-type string. So the `c`-suffixes of this set are all either a `c`-suffix of an `r`-string or a `c`-suffix of an `s`-string.

```
  Alt r s -> undefined
```

Next up, `Cat r s` matches an `r`-type string followed by an `s`-type string. How would you characterize the `c`-remainders of such `r ++ s`-strings? Make sure you think about the case where `r` can match the empty string. Write a regular expression to characterize these `c`-remainders.

```
  Cat r s -> undefined
```

Finally, `Star r` matches any number of consecutive `r`-type strings. How would you characterize the set of strings that you get by lopping off the initial `c`s from this set (and eliminating those strings that don't start with `c`)?

```
  Star r  -> undefined
```

Test your function on a number of examples:

```
ghci> re = ...
ghci> and [ match (remainder re c) u | (c:u) <- take 100 (mset re) ]
```

Actually, while we're here, might as well write a new match function, since the `remainder` notion opens up a new very simple recursive definition.

```
matchRem re ""     = emptyOK re
matchRem re (c:cs) = undefined
```

7 Building the machine recursively

As described above, we will convert a regular expression to an automaton by calculating successive remainders. To do this recursively, we'll build the machine up as we whittle down the expression.

7.1 Preliminaries

First, let's fix some alphabet here that will be shared across your regular expressions and your machines. (Feel free to change this.)

```
type Alphabet = [Char] -- Σ
sigma :: Alphabet
sigma = "abcd"
```

We'll be associating states in the machine we build with regular expressions. So, as with the `compress` function in our transducers, we'll want to keep a dictionary around mapping `Regexp`s to `State`s. And at any given point, we'll have built part of the total eventual machine, so a dictionary of the states and transitions assembled so far.

```
type REDict = [(Regexp, State)]
type PartialMachine = (REDict, TransF)
```

Here's the idea. Say we're standing at a state `q` associated with an expression `re`. For each character `c` in the alphabet, we need to build a state representing the remainder of `re` after `c`. Then we need to move to those new states and repeat the process. But we have to be a little bit clever about how we do this or else it would just go on forever. We might be at a state representing `Zero`, find out its `c`-remainder is `Zero`, add a new state also representing `Zero`, and then repeat. To make sure this terminates, instead of adding a *new* state also representing `Zero`, we should just add a loop from the state we already have back to itself. More generally, whenever we find out that we already have a state associated with `remainder re c`, we should add an arc back to that existing state, rather than creating a new one.

We'll do this with a pair of mutually recursive functions `along` and `across` (quite similar to the `combine` and `parse` functions for CFGs, or even more similar to the simple `even` and `odd` functions). The first one, `along`, calculates a single remainder for a single character, adds a single transition to that remainder, and then hands control over to `across`. That latter function, `across`, calls `along` for each character in the alphabet to add the relevant next transitions. The result is a ratcheting action:

Importantly, you want to define `across` so that it calls `along c1`, `along c2`, …, in sequence, passing the updated machine after each `along` in as input to the next `along`. This is because you might need to connect states you build in later steps to states you build in earlier ones (remember, the goal here is to avoid adding duplicate states). You might think of `along` as taking a single step in the machine-building process, and `across` as taking repeated steps until you run out of symbols in the alphabet, at which point the machine can be returned as is.

```
across :: (Regexp, State) -> (PartialMachine -> PartialMachine)
across qi = go qi sigma
  where -- recurse over the alphabet, building a big machine update pipeline
        -- that updates by running along the first symbol, then updating
        -- the resulting machine by running along the second, and so on
        go :: (Regexp, State) -> Alphabet -> (PartialMachine -> PartialMachine)
        go qi []     = undefined
        go qi (c:cs) = undefined
```

Now, onto the dirty work. `along` builds out the machine from the current RE/state by transitioning along a given character to its remainder. If that remainder is already in the machine, then it will just need to add the new arc to it. But if it's not already in the machine, then it will need to add it together with the arc to it, and then start the whole `across`/`along` building operation at the new state.

```haskell
along :: (Regexp, State) -> Char -> (PartialMachine -> PartialMachine)
along (q0, i0) c = \(dict, arcs) ->
  -- calculate the re corresponding to the strings you can accept after c
  let qc = undefined

  -- check to see if there is already a state
  -- in the machine corresponding to that remainder
  in case undefined of

    -- if so, leave the states as they are (nothing to add)
    -- and add a new arc from the current state to the remainder state
    Just ic ->
      undefined

    -- if not, then you'll need to add it to the machine's dictionary
    -- in addition to adding the new edge
    Nothing ->
      let -- allocate a new state not yet assigned to any RE in the dictionary
          ic    = undefined
          -- add the new (remainder, state) pair to the dictionary
          dict' = undefined
          -- add an edge from the current state to the new one
          arcs' = undefined
      in
      -- now build up the rest of the machine starting from the new state
      undefined
```

7.2 Putting it together

Define the initial configuration of the machine, with a single state for the RE you're aiming to convert.

```haskell
initialState :: Regexp -> (Regexp, State)
initialState re = undefined -- the first state of the partial machine

-- Note that you don't need to (and shouldn't!) change this instance of 'undefined',
-- since there are no transitions yet, so the function is in fact undefined
initialMachine :: Regexp -> PartialMachine
initialMachine re = ([initialState re], undefined)
```

All that's left is to write the function. Remember, an RE/state is final if it matches the empty string (at which point you've consumed all the characters you need, so you should stop).

```haskell
reToDFA :: Regexp -> DFA
reToDFA re = DFAWith qs sigma q0 fs arcs
  where
    -- build the machine out from the initial configuration
    (dict, arcs) = undefined
    -- start with the state you assigned to the initial RE
    q0 = undefined
    -- extract the states from the dictionary
    qs = undefined
    -- decide which of those states are final
    fs = undefined
```

Test your results on a few REs. Make sure you test both that the DFA resulting from conversion accepts the strings that the RE matches and that it rejects the strings that the RE doesn't match.

```
ghci> re = ...
ghci> and [ daccepts (reToDFA re) u | u <- take 100 (mset re) ]
ghci> and [ not (daccepts (reToDFA re) u) | u <- [.. some strings re doesn't match ...]]
```