Domain Specific Language - Probabilistic Regular Expressions

Assignment 4 - Programming Language Design (PLD)
University of Copenhagen

Cami Krogh Dalsgaard - rnq851

Friday 29th March, 2019

User guide

Use F#'s interpreter to run the code:

```
$ fsharpi regex.fsx
```

A41.1 - Embedded vs stand-alone language

I have chosen to implement this domain-specific language as an embedded language instead of a stand-alone language, because the programming language F# as a functional language has a lot of functionality which can be exploitet and that just makes it really easy to implement. This would be the possibility to define types, types that also can be recursive and hence a tree, and that it has pattern matching. There is no need to implement a full parser, when the DSL I have to implement is small and the syntax can be defined in terms of a simple expression tree. All in all it has two primary functionalities: a) Generate random strings b) Calculate the possibility that a string occurs from a given regular expression.

The only thing that for me cleary speaks for doing it as a stand-alone language, is the way I have to write the regular expressions. When I want to state the regular expression $(a|_{0.2}b\ |_{0.3}c\ |_{0.4}d\ |_{0.5}r)^*_{0.2}$, I have to write it in a terrible way like this:

$$(Star(Or(Sym('a'), Or(Sym('b'), Or(Sym('c'), Or(Sym('d'), Sym('r'), 0.5), 0.4), 0.3), 0.2), 0.2)).$$

I am sure that if I decided to do it as a standalone language it would take a lot of extra hours of work to accomplish, and I do not deem this necessary.

A41.2 - Syntax

The grammar I have decided on is written as a datastructure in F# being a discriminated union that is partly defined recursively.

```
type Probability = float

type Regex =
    | Sym of char
    | Epsilon
    | Conc of Regex * Regex
    | Or of Regex * Regex * Probability
    | Star of Regex * Probability
    | Maybe of Regex * Probability // matches Regex One time w/ prob. p, zero w p-1
    | Plus of Regex * Probability  // matches One or more times
```

Figure 1: Syntax of regular expressions

The regular expression $a\ |_{0.4} b$ is hence defined from the triple `Regex * Regex * Probability` being `Sym('a') * Sym('b') * 0.4`.

The reason I chose this is because it extremely easy to traverse a regular expression presented in this form using pattern matching.

I have decided that the characters allowed as single characters is all of the basic type `char`, being standard unicode characters. Line 2 in the syntax gurantees this. To make sure that all probabilities given in the regular expressions are strictly between 0 and 1 a function to check their validity is implemented and used:

```
// Checking that p is between 0 and 1

let valid (p : float) : bool =
    0. < p && p < 1.

// Checking validity of probability of probabilistic regular expression

let validity (reg : Regex) : bool =
    match reg with
    | Or (_,_, p) -> valid p
    | Star (_, p) -> valid p
    | Maybe (_, p)-> valid p
    | Plus (_, p) -> valid p
    | _ -> true
```

Figure 2: Syntax of regular expressions

Whenever a probability is given as part of a regex the function `valid` is called, and if there is no probability involved the `validity` function just returns true.

A41.3 - Semantics to generate random strings

The semantics to generate sequences of random strings have been implemented as the function `rndStr` (figure 3) using recursion and pattern matching on the regular expression given as argument. If the regex matches on `Epsilon` the empty string is returned and if it is a symbol, `Sym c`, the symbol casted to a string is returned. If it is a concatenation of two regular expressions, then I generate a string from the first regex and appends this string with the string generated from the second regex.

The fun part is when the probabilities has to be considered, like in `Or(r1, r2, p)`. I simply generate a random float between 0 and 1 using the library `System.Random` and if this is below or equal to `p : Probability : float` the string generated from the recursive call with `r1` is returned and if above the string generated from `r2`.

`Star(r, p)` is implemented in a similar way except that it makes two recursive calls to match on `r` and on `Star(r, p)`, if the random number generator generates a float above `p`.

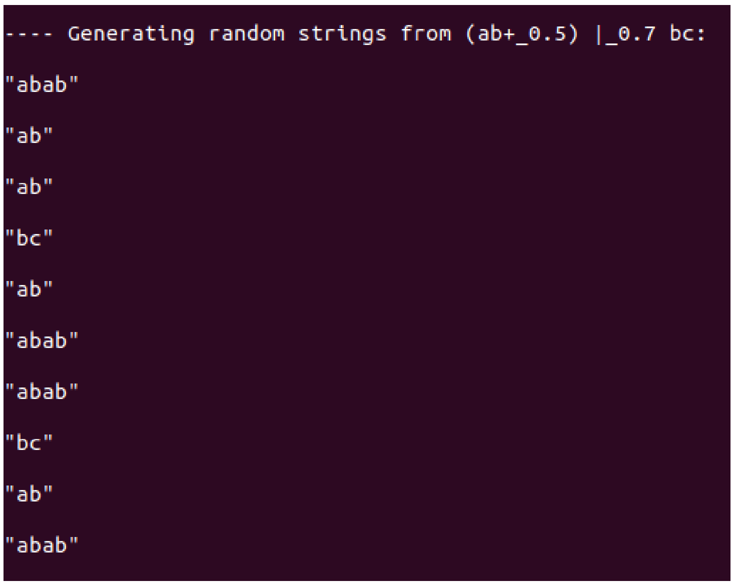
```
// Generating random strings from a regex
let rec rndStr (reg : Regex) : string =
    match reg with
    | Epsilon -> ""
    | Sym c    -> string(c)
    | Conc (r1, r2)  -> rndStr r1 + rndStr r2
    | Or (r1, r2, p) -> if rnd.NextDouble() <= p then rndStr r1 else rndStr r2
    | Star (r, p)    -> if rnd.NextDouble() <= p
                        then ""
                        else rndStr r + rndStr (Star (r, p))
    | Maybe (r, p) -> if rnd.NextDouble() <= p then rndStr r else ""
    | Plus (r, p)  -> if rnd.NextDouble() <= p
                      then rndStr r // match one time
                      else rndStr r + (rndStr <| Plus (r, p)) // more times
```

Figure 3: Random string generator using probabilistic regular expressions

The following code in figure 4 can result in a output like the one shown in figure 5 below.

```
    prs "Generating random strings from (ab+_0.5) |_0.7 bc:"
    for i = 1 to 10 do
      prn <| rndStr (Or(Plus(Conc(Sym('a'), Sym('b')), 0.5),
                                              (Conc(Sym('b'), Sym('c'))), 0.7))
```

Figure 4: Random string generation



```
---- Generating random strings from (ab+_0.5) |_0.7 bc:

"abab"

"ab"

"ab"

"bc"

"ab"

"abab"

"abab"

"bc"

"ab"

"abab"
```

Figure 5: Random string generation

Furthermore I have made a function `genNTimes` which generates `n` random strings from a regex consing them into a list and returning the list. Then another function, `countAsAndBs` calls `genNTimes` giving it the expression $a|_{0}.4\ b$, checks the amount of a's and b's in the list and returns the result as a tuple. When invoked with `n` = 1000 the result is always close to 400, 600.

```
// Counts how many a's and how many b's is generated when calling genNTimes w/
// regex: a|0.4b and 'n' and returns result as a tuple: a * b

let countAsAndBs (n : int) : int * int =
    let cntA = genNTimes (Or(Sym('a'), Sym('b'), 0.4)) n
               |> List.filter (fun e -> e = "a")
               |> List.length
    (cntA, n-cntA)
```

Figure 6: Random string generation

A41.4 - Semantics for probability of given string

The semantics that finds the probability of deriving a specific string from a regular expression is implemented using pattern matching on the regular expression and the string both given as argument to the function `prb : (reg : Regex) -> (w : string) -> Probability`. I simply follow hint i) that Torben has been so kind to provide.

Following the hint was straight forward and there is really nothing much to say about implementing it, the only interesting thing occurs when I have to calculate the probability when s_p^* is given as argument. I return p if the string if empty and otherwise I remember that $1-p$ is the probability that it matches on `w` to begin with. Then to find the probability that it matches on whatever s is and that it might match on s^* more times line 7 in the figure below is written. It finds the probability of the Concatenation of `r` and `Star(r,p)` on the whole of the string `w` and multiplies this with $1-p$.

```
1     | Conc (r1, r2), w  -> let mutable p = 0.
2                            for i = 0 to w.Length-1 do
3                                p <- p + (prb r1 w.[0..i]) * (prb r2 w.[i+1 ..])
4                            p
5     | Star (r, p), w -> match w with
6                         | "" -> p
7                         | _  -> (1.-p) * prb (Conc(r, (Star(r, p)))) w
```

Figure 7: Handling Star

To see it in use I have called the function `prb` as follows (which is not pretty), when the code is executed it prints 0.000000000336331991 in the terminal:

```
    prs "Abracadabra, my dudes"
    printfn "%.18f"
     <| prb (Star(Or(Sym('a'),
                     Or(Sym('b'),
                        Or(Sym('c'),
                           Or(Sym('d'), Sym('r'), 0.5), 0.4), 0.3), 0.2), 0.2))
                             "abracadabra"
```

Figure 8: Probability for generating "abracadabra" from regex

A41.5 - Extensions

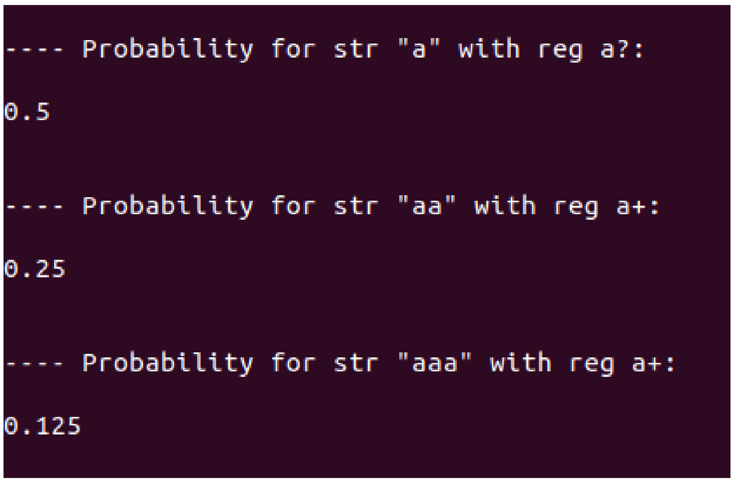
I have implemented two extra regular expressions as extension to the DSL. I decided on s^+ (one or more times) and $s?$ (zero or one time) since they are two very common used regular expressions, which we have been so lucky to work on in a former assignment in programming language design this year. s^* tells if something matches zero or more times, but it is nice to have something to tell if it matches at least one time, therefore s^+ is a good choice.

```
    prs "Probability for str \"a\" with reg a?:"
    prn <| prb (Maybe(Sym('a'), 0.5)) "a"

    prs "Probability for str \"aa\" with reg a+:"
    prn <| prb (Plus(Sym('a'), 0.5)) "aa"

    prs "Probability for str \"aaa\" with reg a+:"
    prn <| prb (Plus(Sym('a'), 0.5)) "aaa"
```

Figure 9: Simple use of Plus (s+) and Maybe (s?)



```
---- Probability for str "a" with reg a?:

0.5

---- Probability for str "aa" with reg a+:

0.25

---- Probability for str "aaa" with reg a+:

0.125
```

Figure 10: Output to terminal from code in figure 9

A41.6 - Evaluation

Writing regular expressions in my embedded language is a hurtful experience. So I would consider writing it as a stand-alone language just from this experience. But because I never am to use this code again, and it is a small scope the language is covering, I think my solution is just fine. It calculates the right numbers for all the examples Torben have given in the assignment text, so that is something to celebrate.

The main problem I experienced, was getting the probability calculator, `prb`, right for `Star`. It took me some time and a lot of thinking before I got it. I also had a lot of doubts about that it cannot derive the empty string. So there may be something I failed on. The other regular expressions were straight forward to calculate.

I am glad that I am fairly strong in F# and that the language simply suited this assignment perfectly. I would not write it in any other language if I could (ok, maybe Haskell, but I have not learned that one yet). All in all it did turn out good.