INF-122 Oblig 1, H-2018

- Les hele oppgaven før du begynner å besvare enkelte spørsmål.
- Du må skrive og kommentere en selvstendig løsning på egen hånd. Innlevering med deler av kode kopiert fra andre steder eller identiske med andre innleveringer blir avvist uten videre.
- Løsning leveres per epost til **Bram.Burger@uib.no** senest på

  tirsdag, 16 Oktober 2017, kl 11:00.
- Du innleverer en enkel fil, `Oblig1.hs`, med kjørbar kode. Filen starter med
  - `module Oblig1 where`
  - etterfulgt av en kommentar med ditt etternavn.
  - Deretter følger hele koden skrevet på vanlig måte.
- Det anbefales ikke å bruke andre pakker eller moduler enn Prelude (spesielt, ikke monadisk parsing omtalt i bokens kapittel 13).
- Spørsmål om oppgaven kan stilles på grupper på fredag 5.10, tirsdag 9.10 og fredag 12.10.
- Resultater hentes og gjennomgås på gruppen tirsdag, 23.10.
- Neste forelesning blir på mandag, 22.10.

1 For pass with up to C:

1.1 parse :: String -> AST

The following grammar *Greg*, with the start symbol `Re`, defines the language $L(Greg)$ of simple regular expressions:

```
Re -> Sq | Sq + Re
Sq -> Ba | Ba Sq
Ba -> El | Ba*
El -> lower-or-digit | (Re).
```

Blanks are ignored. By `lower-or-digit` are denoted lowercase letters or digits, which are the only terminal symbols besides +, *, (and). Nonterminals are denoted by strings starting with uppercase letter, and this is assumed about nonterminals used as input or output of your functions. Thus `aP` denotes the terminal `a` followed by the nonterminal `P`, but `Pa` denotes a single nonterminal. (In the implementation, these are all strings, i.e.,, "aP", "P", "Pa".) Sequencing and + associate to the right. As an example of a string belonging to the language

$L(Greg)$, we take `ex = "(a+b)*dc*"`, which can be derived as follows:

```
Re -> Sq -> Ba Sq -> Ba Sq Sq -> Ba* Sq Sq ->
  (Re)* Sq Sq -> (Sq + Re)* Sq Sq -> (Ba + Re)* Sq Sq ->
  (El + Re)* Sq Sq -> (a + Re)* Sq Sq -> (a + Sq)* Sq Sq ->
  (a + Ba)* Sq Sq -> (a + El)* Sq Sq -> (a + b)* Sq Sq ->
  (a + b)* Ba Sq -> (a + b)* El Sq -> (a + b)* d Sq ->
  (a + b)* d Ba -> (a + b)* d Ba* -> (a + b)* d El* -> (a + b)* d c*
```
(1)

The first task is to build a recursive descent parser

`parse::String -> AST`, **1!**

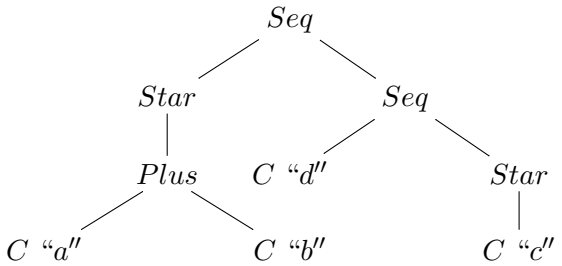
which (a) for a string not belonging to the language $L(Greg)$ gives an error message, while (b) for a correct regular expression from $L(Greg)$ builds an abstract syntax tree of the type

`data AST = C String | Seq AST AST | Star AST | Plus AST AST`.

For instance, for the expression `ex = "(a+b)*dc*"` derived in (1), we should obtain

`parse ex = Seq (Star( Plus(C "a")(C "b") )) (Seq (C "d") (Star(C "c")))`,

which represents the tree:



Each regular expression itself represents a language. For instance, the example expression `ex` represents the language $L(ex)$ with strings which start with an arbitrary, possibly empty, sequence of a and b, followed by a single d, followed by zero or more c. We will come back to these languages in problem 1.3 and, to some extent, in 1.2. Here, we are concerned only with the language consisting of regular expressions, and not with the regular languages denoted by such expressions.

1.2 gr :: AST -> Gr

Regular grammars provide an alternative way of presenting regular languages. For each regular expression there is a regular grammar which determines the same language, and vice versa. A regular grammar is one where each production is `N -> RS`, where `N` is a single nonterminal, while the right side `RS` has one of the following forms (lowercase letters range over terminals and uppercase over nonterminals; ϵ is the empty string):

- r1. `a` – a single terminal, or
- r2. `aP` – a single terminal followed by a single nonterminal, or
- r3. ϵ – the empty string, or
- r4. `P` – a single nonterminal.

The productions of kind r4 are usually excluded and we will get rid of them in Part 2. For the example expression `ex` above, a corresponding regular grammar G_{ex} (generating the same language, i.e., $L(G_{ex}) = L(ex)$) can be the following, with the start symbol `S`:

`S` $\rightarrow$ `aS` | `bS` | `dC`
`C` $\rightarrow$ `cC` | ϵ. (2)

For instance, the string `aabdcc` belongs to $L(ex)$, being derivable in this grammar, e.g.:

`S -> aS -> aaS -> aabS -> aabdC -> aabdcC -> aabdccC -> aabdcc`.

The task now is to convert an `AST` representing a regular expression in the language $L(Greg)$, to a regular grammar which determines the same language. Implement function

`gr::AST -> Gr` **2!**

which, given a `t::AST`, (i.e., `t=parse(ex)` for some `ex` $\in L(Greg)$) returns a regular grammar such that $L(gr(t)) = L(ex)$. To represent grammars, introduce two type abbreviations

```
type Rules = [(String,String)]
type Gr = (String,Rules).
```

The first component in the type `Gr` is the start symbol, while the second one is a list with all the productions. A production $X \to Y$ is represented by the pair (X, Y), where X is a string containing a nonterminal symbol and Y the right side of the rule. For instance, the grammar G_{ex} from (2) could be represented as the pair

$$(\texttt{"S"}, [(\texttt{"S"}, \texttt{"aS"}), (\texttt{"S"}, \texttt{"bS"}), (\texttt{"S"}, \texttt{"dC"}), (\texttt{"C"}, \texttt{"cC"}), (\texttt{"C"}, \texttt{""})]). \quad (3)$$

In the sketch below and further down, we use the following notation:

1. (S, R, N) – a grammar with the start symbol S, productions R and nonterminals N. A pair (X, Y) denotes production $X \to Y$, i.e., $X \in N$ and Y is one of the cases r1-r4.
2. Uppercase letters like A, B, S are metavariables denoting nonterminals and lowercase letters b, c metavariables for the terminal symbols. (Terminals are not listed explicitly but are assumed to be all such symbols occurring in the rules.)
3. For an $x \in L(Greg)$ or $x :: AST$, by $G_x = (S_x, R_x, N_x)$ we denote the respective grammar for which $L(G_x) = L(x)$.
4. $\{f(x) \mid x \in R\}$ denotes the set of $f(x)$, i.e., results of applying a function f to the argument x, for all $x \in R$, i.e., all x from a given set R. It can be compared to Haskell's list comprehension `[f(x)|x<-R]`, with the difference that in sets – unlike in lists – ordering and multiplicity of elements do not count, e.g., $\{1, 3, 1, 2, 1\} = \{1, 2, 3\}$.
5. $X \cup Y$ denotes the set union of sets X and Y. More generally, $\bigcup\{X_i \mid i \in I\}$ denotes the union of sets X_i, for all $i \in I$ with some index set I identifying the relevant sets X_i.

The conversion of (an `AST` for) a regular expression to an appropriate grammar, giving the same language, can be performed by composing partial grammars into bigger ones, traversing the AST bottom-up with an implementation of the following pseudocode.

`gr (C "a")` $= (A, \{(A, a)\}, \{A\})$ – fresh A.

`gr (Plus x y)` $= (S, R, N)$, where

$N = N_x \cup N_y \cup \{S\}$ – fresh S

$R = R_x \cup R_y \cup \{(S, X) \mid (S_x, X) \in R_x\} \cup \{(S, Y) \mid (S_y, Y) \in R_y\}$.
X, Y denote arbitrary right sides of the rules present in R_x, R_y, which are copied to the new productions from the fresh start symbol S. From S, there is now a production to every right side X of any production of `gr(x)` from its start symbol S_x, and to every right side Y of a any production of `gr(y)` from its start symbol S_y. These earlier start symbols S_x and S_y are retained, since they may occur in the rules of R_x and R_y, but they are no longer start symbols.

`gr (Seq x y)` $= (S_x, R, N_x \cup N_y)$ – old S_x, where

$R = R_y \cup \{(A, cB) \mid (A, cB) \in R_x\} \cup \{(A, bS_y) \mid (A, b) \in R_x\} \cup \{(A, S_y) \mid (A, \epsilon) \in R_x\}$.
Rules of `gr(y)`, as well as those of `gr(x)` which have a terminal followed by nonterminal in the right side, remain unchanged. Rules enabling completion of a derivation in `gr(x)`, namely, those with a single terminal or empty string in the right sides, obtain in addition the start symbol of `gr(y)`, to enable continued derivation with the rules of `gr(y)`.

`gr (Star x)` $= (S_x, R, N_x)$ – old S_x, where

$R = R_x \cup \{(S_x, \epsilon)\} \cup \{(A, bS_x) \mid (A, b) \in R_x\} \cup \{(A, S_x) \mid (A, \epsilon) \in R_x\}$.
We can derive all `gr(x)` could using R_x, the empty string and, whenever we had a completed derivation in `gr(x)` (with a single terminal or empty string in the right side of a production), we can now continue starting another derivation from S_x.

The last kinds of the new rules for `Seq` and `Star` exemplify the case r4 of a production from a nonterminal to a single nonterminal without reading any input.

Example 1.1 *The AST for the regular expression* `a*b` *is* `Seq(Star (C"a")) (C"b")`. *We construct a grammar for it starting at the bottom of this tree:*

```
                gr(C"a") = ("A",[("A","a")])
          gr(Star (C"a")) = ("A",[("A","a"),("A",""),("A","aA")])
                gr(C"b") = ("B",[("B","b")])
gr(Seq(Star(C"a"))(C"b")) = ("A",[("B","b"),("A","aA"),("A","aB"),("A","B")])
```

Use first the pseudocode for `gr` to obtain a grammar for the example `ex` from point 1.1. It will be different from G_{ex} in (2)! Then implement `gr`. Note that the implementation has to generate fresh nonterminal symbols, due to cases `'a'` and `Plus`. The number of such symbols is unknown and cannot assume any bound for it.

1.3 mem :: String -> String -> Bool

Implement a "membership checker", i.e., function `mem::String->String->Bool` which, for any string s and $ex \in L(Greg)$, decides if $s \in L(ex)$. For instance, for the language defined by our expression `ex`, the following shall hold `mem "aabdcc" ex = True`, `mem "d" ex = True`, `mem "a" ex = False`, `mem "aabdcd" ex = False`. This function shall be a wrapper around a generic parser

```
gpr::String -> Gr -> Bool
```

3!

which, taking as input a string `s` and a description of a regular grammar generated in 1.2 (i.e., a `gr(parse ex)::Gr`), checks if `s` $\in L($`gr(parse ex)`$)$, i.e., if `s` $\in L($`ex`$)$.

Note that this parsing may require backtracking. For instance, if the grammar contains two rules $A \to aB \mid aC$, then the parser reading 'a', while processing the nonterminal A, may first try the transition to B, but if this fails, then it has to try C.

2 dfa :: Gr -> Gr (for better than C)

As noted, the grammar resulting from the `gr` function, like (3), requires in general backtracing. You shall improve the grammar so that parsing becomes completely deterministic. The main task is to program a function `dfa::Gr -> Gr` which, given a regular grammar, like a one produced by `gr`, turns it into a deterministic one defining the same language.

Nondeterminism of the (strictly speaking, an automaton for the) grammar, which we want to eliminate, amounts to two possible things:

N1. If the grammar contains two productions, $A \to B \mid aC$, for nonterminals A, B, C and terminal a, then having the symbol a as the next in the input, while processing A, we may either read this a and continue with C or else, nondeterministically, jump to B and continue from there.

N2. If the grammar contains two productions $A \to aB \mid aC$, then the choice between these remains undetermined when reading input a while processing the nonterminal A.

2.1 eps :: String -> Rules -> [String]

The simple idea of getting rid of such nondeterminism (and hence of backtracking in parsing) is to form a new grammar whose nonterminals are subsets of nonterminals of the original one. Given a situation like in N1, any transition to A will actually go to $\{A, B\}$, since reaching A in the original grammar, we can always jump directly to B. Let $G = (S, R, N)$ be the original grammar and $GG = (SS, RR, NN)$ a new one which we want to construct (denoted with double symbols). The new nonterminals NN are subsets of N. For each nonterminal $X \in N$, we obtain a nonterminal $XX \in NN$ as the subset of N that contains all $Y \in N$ reachable from X by such "empty productions" in the original grammar (not reading any input due to the form r4, with right side being a single nonterminal).

Example 2.1 *For the grammars to the left, the resulting sets are shown to the right. The grammar (A) was obtained in Example 1.1:*

(A)	$A \to aA \mid aB \mid B$	$AA = \{A, B\},$
	$B \to b$	$BB = \{B\}.$
(R)	$S \to aS \mid bS \mid bQ \mid D$	$SS = \{S, D\},$
	$Q \to cQ \mid dD$	$QQ = \{Q\},$
	$D \to dD \mid d$	$DD = \{D\}.$

Program a function

```
eps::String -> Rules -> [String]
```

4!

which, for a nonterminal `X` provided as the first string argument and productions `R` of the (original) grammar as the second argument, returns the set `XX`, that is, all nonterminals reachable from `X` by the "empty productions" from `R`. The start symbol of the new grammar is thus `eps(S)`, where `S` is the start symbol of the original grammar. A nonterminal `X` belongs always to its set `eps(X)`.

2.2 dfa :: Gr -> Gr

The new nonterminals as subsets of the old ones suggest also a solution to the problem N2. Given multiple transitions on the same input symbol from a single nonterminal, like on a from A in N2, we go to the nonterminal coding the set of the original nonterminals reachable from A on reading a. The same is done when considering now a production from a new nonterminal, that is, a subset of old ones. Begining with the new start symbol $SS = eps(S\ R)$, where S is the start symbol of the original grammar and R are its rules, for each such subset $XX = \{X_1, ..., X_n\}$ and input symbol a, we add the production

$$XX \to aYY, \text{ where } YY = \bigcup\{(\texttt{eps}\ Y_i\ R) \mid X_i \in XX, X_i \to aY_i \in R\}, \tag{4}$$

and each Y_i is a nonterminal. In more detail, if $XX = \{X_1, ..., X_n\}$ is a new nonterminal, while the original grammar contains productions with input symbol a from, say, exactly $X_1, ..., X_m, m \leq n$, to $Y_1, ..., Y_m$, i.e., $X_i \to aY_i$ for $1 \leq i \leq m$, then we obtain the production

$XX \to aYY$, where YY is a new nonterminal, $YY = \texttt{eps}(Y_1\ R) \cup ... \cup \texttt{eps}(Y_m\ R)$. If some X_i has a production to the empty string or a single terminal symbol, then each new nonterminal containing X_i also has this production. If some X_i has several productions on input symbol a, their targets are gathered into the resulting set according to (4), for each new nonterminal containing X_i. In N2 above, this gives the transition $\{A\} \to a\{B, C\}$. New type for the rules may be needed, e.g., `[([String],Char,[String])]`. Whatever format you use to represent these rules internally, `dfa` shall return a resulting grammar of type `Gr`.

Example 2.2 *For the grammar with the rules (A) from Example 2.1, we obtain the grammar with the start symbol AA and productions $AA \to aAA \mid b$. (Strictly speaking, we obtain also the production $BB \to b$, but since the nonterminal BB is not reachable from the start symbol AA, it can be dropped without changing the language.)*

For the grammar (R) from Example 2.1, we noted that $eps(S\ R) = [S, D]$, while $eps(Q\ R) = [Q]$ and $eps(D\ R) = [D]$. (Sets are now Haskell lists as results of eps.) So $[S, D]$ is the start symbol of the new grammar. Reading b in S or D we can only go to S or Q, but since S has a silent transition to D, D accompanies S in every set, as shown to the left. The same grammar is given to the right with the nonterminals renamed [S,D]=S and [S,D,Q]=T:

$$\begin{array}{rcl} [S,D] & \to & a[S,D] \mid b[S,D,Q] \mid d[D] \mid d \\ [S,D,Q] & \to & a[S,D,Q] \mid b[S,D,Q] \mid c[Q] \mid d[D] \\ [Q] & \to & c[Q] \mid d[D] \\ [D] & \to & d[D] \mid d \end{array} \qquad \begin{array}{l} S \to aS \mid bT \mid dD \mid d \\ T \to aT \mid bT \mid cQ \mid dD \\ Q \to cQ \mid dD \\ D \to dD \mid d. \end{array}$$

The two productions from S, to dD and to d, do not introduce nondeterminism. Reading d in S, one can check whether there remains anything of the input and, if so, continue with D. Program the function

`dfa::Gr -> Gr` **5!**

which performs the transformation as described and, from an arbitrary regular grammar given as input, produces one which defines the same language but avoids both problems N1 and N2, i.e., has no empty productions nor any multiple productions from any nonterminal to different nonterminals on the same input symbol.

Your function `gpr` from 1.3 should give the same results on a given regular grammar `gra` and on its deterministic version `dfa(gra)`, i.e., for every string `s`, it should hold that:

`gpr s gra == gpr s (dfa gra)`.

2.3 dgpr :: String -> Gr -> Bool

Assuming that input grammars are deterministic – that is, avoid both N2 and N1 (containing no "empty productions" of the form r4), for instance, because they result from `dfa` – implement a new generic parser

`dgpr::String -> Gr -> Bool`, **6!**

which does not perform any backtracking and is such that for each string and grammar `gra` with regular rules of the kind r1–r4, `gpr s gra == dgpr s (dfa gra)`.