INF122 Obligatorisk Oppgave 2

Frist: **7. november, 23:59**

Innlevering forutsetter at du har programmert alt alene og ikke har brukt andres kode (utenom evt. kode fra forelesninger, boken eller gruppeøvelser).

Introduksjon

Regulære uttrykk (regex) koder enkle språk brukt til å søke etter og manipulere tekst. I denne oppgaven skal du bygge et verktøy som bruker regulære uttrykk for søking og redigering av tekst i strenger og filer.

Regulære uttrykk er ganske enkle å forstå. Hvis du ønsker å søke etter strengen "hello", ville det regulære uttrykket vært `hello`. Hvis du ønsker å finne alle forekomster av ordet 'hello', dvs., matche både "Hello" og "hello", kan du bruke regex-uttrykket `h|Hello`. Dette betyr "match tegnet 'h' eller 'H', etterfulgt av 'ello'". Som et annet eksempel: si du har et dokument, og du vil finne noens navn i det. Det fulle navnet er John Kevin Smith, men du vet at han ikke alltid bruker mellomnavnet. Du kan lage et regulært uttrykk som: `(John Kevin)|(John) Smith` som betyr "match enten 'John Kevin' eller 'John', etterfulgt av ' Smith'".

Andre morsomme ting du kan gjøre med regex:

- `.` matcher hvilket som helst tegn
- `R*` betyr "match 'R' 0 eller flere ganger" — * kalles Kleene-stjernen

Så for eksempel:

- `.*` betyr match hvilken som helst streng
- `I .... jazz music` vil matche både 'I love jazz music' og 'I hate jazz music'
- `loo*ng` vil matche 'long', 'loong', 'looong', ..., 'looooooooooong',... osv.

Oppgaven har tre deler:

1. skrive en parser for regulære uttrykk,
2. programmere forskjellige funksjoner som jobber på strenger med bruk av regulære uttrykk,
3. programmere to aksjoner som benytter funksjoner fra punkt 2 for å finne eller erstatte strenger i filer ved hjelp av regulære uttrykk.

Del I

I denne delen skal du programmere en parser for regulære uttrykk, gitt av følgende grammatikk. Terminalsymboler `Char` er vilkårlige tegn, med unntak av de som er terminalsymboler brukt med spesiell betydning av regulære uttrykk. De siste er merket med enkeltanførselstegn, dvs., '|', '(', ')', '.'

og '* '. "Empty" står for det tomme uttrykket som er lagt til for å forenkle deler av programmering. (Merk presedensen: `|` binder sterkere enn `*`, som binder sterkere enn konkatenering `Reg1 Reg`. F.eks., betegner uttrykket `A|aB` det samme som `AB|aB` (nemlig, enten `AB` eller `aB`) og noe annet enn `A|(aB)` som står for enten `A` eller `aB`.)

```
Reg -> Re1 Reg | Re1 | Empty
Re1 -> Re2 '*' | Re2
Re2 -> Re3 '|' Re2 | Re3
Re3 -> Char | '.' | '(' Reg ')'
```

Datatype nedenfor skal brukes for å representere regulære uttrykk:

```
data Regex = Atom Char | Both Regex Regex | After Regex Regex | Kleene Regex | Empty |
Any
```

Spørsmål 1

Implementer funksjonene `reg, re1, re2, re3 :: String → (Regex,String)` som en rekursiv descent parser som konverterer inputstrenger til regulære uttrykk:

```
ghci> reg "a"
(Atom 'a', "")
ghci> reg "(a|b)*"
(Kleene (Both (Atom 'a') (Atom 'b')), "")
ghci> reg "ab."
(After (Atom 'a') (After (Atom 'b') Any), "")
ghci> reg " ab"
(After (Atom ' ') (After (Atom '') (Atom 'b')), "")
```

Del II

Nå skal du bygge den delen av programmet som søker i tekst etter regulære uttrykk. I filen har du fått typealiasen:

```
type Transition = String -> [(String,String)]
```

For hver type Regex lager vi én overgang (`Transition`). Den tar en streng som input, og returnerer en liste med tupler. Det første elementet i hver tuple er den delen av strengen som regexen vår har matchet, og det siste er resten av strengen. De fleste tilfeller bruker kun lister med ett slikt par, men lengre lister blir nødvendig for å behandle alternativer.

Spørsmål 2

Implementer funksjonen `matchChar :: Char → Transition` som lager en overgang som matcher det oppgitte tegnet.

```
ghci> let trans = matchChar 'z'
ghci> trans "zebra"
[("z", "ebra")]
ghci> trans "hello"
[]
```

Spørsmål 3

Implementer funksjonen `matchAny :: Transition` som alltid matcher det første tegnet:

```
ghci> matchAny "Hello!"
[("H", "ello!")]
ghci> matchAny "Bye!"
[("B", "ye!")]
ghci> matchAny ""
[]
```

Implementering av en regulær uttrykksmotor blir mye enklere hvis vi har den *tomme strengen*, som matcher enhver streng uten å konsumere noen av den. Dette var grunnen til at `Empty` ble inkludert i grammatikk. (Eksempelet for Spørsmål 3 burde hjelpe deg til å forstå).

Spørsmål 4

Implementer funksjonen `matchEmpty :: Transititon` som matcher den tomme strengen.

```
ghci> matchEmpty "Hello!"
[("", "Hello!")]
```

Måten vi skal gjøre dette til en regexmotor på, er ved å komponere `Transition`-ene sammen.

Spørsmål 5

Implementer funksjonen `matchBoth :: Transition → Transition → Transition` som tar inn to overganger, og produserer en overgang som matcher begge:

```
ghci> match = matchBoth matchAny matchEmpty
ghci> match "hello"
[("h", "ello"), ("", "hello")]
```

Du kan også tenke på signaturen som `matchBoth :: Transition → Transition → String → [(String,String)]`!

Spørsmål 6

Implementer funksjonen `matchAfter :: Transition → Transition → Transition` som matcher den første overgangen og deretter den andre:

```
ghci> let match = matchAfter (matchChar 'h') (matchChar 'e')
ghci> match "hello"
[("he", "llo")]
ghci> match "hurra"
[("", "hurra")]
```

Spørsmål 7

Ved å bruke parseren fra Del I, implementer funksjonen `regex2trans :: Regex → Transition` som konverterer regulære uttrykk til overganger.

```
ghci> match = regex2trans $ fst $ reg "(h|H)ello"
ghci> match "Hello"
[("Hello", "")]
ghci> match "hello world"
[("hello", " world")]
ghci> match "Alpha Omega"
[]
```

Spørsmål 8

Våre `Transition`-funksjoner returnerer en liste med treff, men vi er normalt bare interessert i det lengste treffet.

Implementer funksjonen `longest :: [[a]] → [a]` som, når den får en liste med lister (for eksempel `[String]`), returnerer den lengste indre listen. Hvis det finnes like lange lister i hovedlisten, returnerer den den første.

```
ghci> longest ["bye", "hello"]
"hello"
ghci> longest ["love", "hate"]
"love"
```

Spørsmål 9

Implementer funksjonen `matchStart :: String → String → String`, som tar inn en regex og en streng, og returnerer det lengste treffet fra starten:

```
ghci> match = matchStart "h|H.*"
ghci> match "Haskell"
"Haskell"
ghci> match "hello"
"hello"
ghci> match "aH"
""
```

`tails :: [a] → [[a]]` er en funksjon i `Data.List`. Den tar en liste og returnerer en liste med iterative halvstykker:

```
ghci> tails [1,2,3]
[[1,2,3], [2,3], [3], []]
ghci> tails "haskell"
["haskell", "askell", "skell", "kell", "ell", "ll", "l", ""]
```

Spørsmål 10

Implementer funksjonen `matchLine :: String → String → String` som bruker `tails` til å finne det lengste treffet i inputstrengen

```
ghci> matchLine "help" "Haskell is very helpful"
"help"
```

Spørsmål 11

Implementer funksjonen `replaceLine :: String → (String → String) → String → String` Det første argumentet er et regulært uttrykk, det andre er en funksjon som tar treff-strengen og gir en streng som skal erstatte den. Det siste argumentet er strengen vi ønsker å gjøre en søk og erstatning på. Vi ignorerer overlapp (akkurat som i Uke42), dvs. erstatter kun den første lengste delen som matcher, før vi fortsetter.

```
ghci> replaceLine "o|O" (const "0") "Look Out!"
"L00k 0ut!"
ghci> replaceLine "j|h" (map toUpper) "james john hobson"
"James JoHn Hobson"
ghci> replaceLine "aa" (const "bb") "aaa"
"bba"
```

Del III

Vi skal bruke det vi har gjort til søking og maipulering av tekstfiler. Vi antar at filer ligger i den samme katalogen hvor aksjoner utføres, slik at `FilePath` er bare filnavn.

Spørsmål 12

Implementer funksjonen `grep :: String → FilePath → IO ()`, som skal skrive ut hver linje fra filen som inneholder et treff.

```
ghci> grep "::" "Oblig2.hs"
reg, re1, re2, re3 :: String -> (Regex, String)
matchChar :: Char -> Transition
matchAny :: Transition
...
```

Spørsmål 13

Implementer funksjonen `sed :: String → (String → String) → FilePath → FilePath → IO ()`, som tar inn et regulært uttrykk, erstatningsfunksjon, input- og outputfil. Den leser inputfilen, utfører søk og erstatning på hver linje, og skriver resultatet til outputfilen. De to filnavnene kan være foskjellige, men aksjonen skal fungere også når de er identiske.

```
ghci> writeFile "test.txt" "this is a test"
ghci> readFile "test.txt"
"this is a test"
ghci> sed " ." (map toUpper) "test.txt" "test.txt"
ghci> readFile "test.txt"
"this Is A Test\n"
ghci> sed "." (map toUpper) "test.txt" "test.txt"
ghci> readFile "test.txt"
"THIS IS A TEST\n"

ghci> writeFile "files.txt" "index.html page1.html"
ghci> sed ".html" (const ".tex") "files.txt" "tex.txt"
ghci> readFile "tex.txt"
"index.tex page1.txt"
```