INF122 Obligatorisk Oppgave 1

Innleveringsfrist: **13 Oktober, kl. 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

Oppgaven går ut på å lage en liten kalkulator, som jobber med språket gitt ved grammatikken under. Den skal parse lovlige uttrykk i språket, bygge et abstrakt syntakstre for input-uttrykket og evaluere det eller skrive det ut på forskjellige måter. Alfabetet av terminalsymboler består av tegn: `*`, `/`, `)`, `(`, `-`, `+`, samt sifrene `0-9`. (I Del III tillater vi også `Term` bestående av bokstaver, så din parser kan fra starten av ta hensyn til det.) Grammatikkens regler er som følger:

```
Expr   = Faktor + Expr | Faktor - Expr | Faktor
Faktor = Term * Faktor | Term / Faktor | Term
Term   = (Expr)        | Num
Num    = 0 | (1|...|9){0-9}
```

Startsymbolet er `Expr`. Du skal ikke endre på grammatikken. Enhver funksjon som du er bedt om å implementere må ha nøyaktig samme navn og type som angitt i oppgaveteksten. I tillegg kan du implementere så mange hjelpefunksjoner som du trenger. Ekstra poeng vil bli gitt for fornuftig håndtering av mellomrom og relevante feilmeldinger ved feil input.

Del I

Leksikalsk analyse

Det første steget av parsing er leksikalsk analyse, som programmeres som en tokeniser. Denne tar en strøm av tegn og grupperer dem i en liste av tokens. Gitt hvor enkelt språket vårt er, kan vi bruke `String` til å representere tokens. Disse er:

— enkle tegn `*`, `/`, `)`, `(`, `-`, `+`,

— sammenhengende sekvenser av sifre,

— sammenhengende sekvenser av bokstaver.

Ingen andre tegn skal forekomme i resultatet av et vellykket kall til tokenise. Mellomrom brukes som et lovlig, men ikke nødvendig, skilletegn. Merk at tallene som har mer enn ett siffer ikke kan starte med 0, men du får noen poeng også hvis parseren din aksepterer f.eks., 0023 som tallet 23.

Spørsmål 1

Implementer `tokenise :: String → [String]` som går gjennom input-strengen og deler den opp i relevante tokens.

```
ghci> tokenise "(+-/)"
["(", "+", "-", "/", ")"]
ghci> tokenise "123 + 4)"
["123", "+", "4", ")"]
ghci> tokenise "38-9  * (46 + 4)"
["38", "-", "9", "*", "(", "46", "+", "4", ")"]
```

Syntaktisk analyse

I denne fasen konverterer parseren listen av tokens til et abstrakt syntakstre, AST, for input-uttrykket.

Du skal bruke følgende datatyper. Et Ast er et tre der bladnoder lagrer Int, og mellomnoder, merket med BinOp konstruktøren, lagrer en av Operatorene, som har to Ast-subtrær som argumenter.

```
data Op = Add | Sub | Mult | Div deriving (Eq, Show)

data Ast = BinOp Op Ast Ast | Tall Int deriving (Eq, Show)
```

En recursive descent parser har en funksjon for hvert ikke-terminal symbol, og disse funksjonene kaller hverandre (er gjensidig rekursive). Vi forsøker å dele det opp i mindre spørsmål, hvor det siste spørsmålet vil knytte det hele sammen.

Hver funksjon har typen `[String] → (Ast, [String])`, inndataen er vår token-liste, utdataen er et par, hvor første del er resultatet av parsing, og andre del inneholder halen av token-listen som ikke ble brukt.

Spørsmål 2

Implementer `parseTerm :: [String] → (Ast, [String])` som (foreløpig) bare parser tall. (Tom. spørsmål 5 kan eksempler og tester bruke kun uttrykk uten parenteser.)

```
ghci> parseTerm ["42", "*", "2"]
(Tall 42, ["*", "2"])
```

Spørsmål 3

Implementer `parseFactor :: [String] → (Ast,[String])` som parser `Faktor`-linjen i grammatikken.

```
ghci> parseFactor ["42","*","2","-", "12"]
(BinOp Mult (Tall 42) (Tall 2), ["-", "12"])
```

Spørsmål 4

Implementer `parseExpr :: [String] → (Ast, [String])` som parser `Expr`-linjen i grammatikken.

```
ghci> parseExpr ["42", "*", "2", "-", "12"]
((BinOp Sub (BinOp Mult (Tall 42) (Tall 2)) (Tall 12)), [])
ghci> parseExpr ["42", "-", "2", "*", "12"]
((BinOp Sub (Tall 42) (BinOp Mult (Tall 2) (Tall 12))), [])
```

Spørsmål 5

Modifiser `parseTerm`-funksjonen din til å også kunne parse `( Expr )`. Du har nå fullført parseren.

```
ghci> parseExpr ["(", "22", "-", "1", ")", "*", "2"]
(BinOp Mult (BinOp Sub (Tall 22)  (Tall 1))  (Tall 2),[])
```

Spørsmål 6

Bruk funksjonene `tokenise` og `parseExpr` for å programmere funksjonen `parse :: String → Ast` som returnerer et AST for en korrekt input streng. Vi forutsetter at den er korrekt mht. grammatikken vår, men du får ekstra poeng dersom parseren din fanger feil og gir relevante feilmeldinger.

Del II

I denne delen skal du implementere 4 funksjoner som bruker Ast for å evaluere input-uttrykket, eller skrive det ut i forskjellige format. Hver funksjon kan implementeres separat, og det er tilstrekkelig for å passere alle testene. Ekstra poeng blir gitt hvis du, istedenfor, implementerer bare én generisk funksjon som traverserer Ast-et og kaller passende parameterfunksjoner, slik at hver av de fire funksjonene blir en instans med passende funksjoner brukt som parametere til denne generiske traverseringsfunksjonen.

Evaluering

Spørsmål 7

Implementer funksjonen `eval :: Ast → Int`.

```
ghci> eval $ parse "(22-1)*2"
42
ghci> eval $ parse "22-1*2"
20
```

Pretty Printing

En pretty printer tar et AST og gjør det om til en streng som er lett å lese for mennesker. Du skal lage tre slike som skriver ut uttrykk i forskjellige formater.

Spørsmål 8

Implementer funksjonen `ppInfix :: Ast → String` som skriver ut uttrykket i samme format som vår grammatikk spesifiserer:

```
ghci> ppInfix (BinOp Mult (Tall 2)  (Tall 3))
"(2 * 3)"
ghci> ppInfix (BinOp Mult (BinOp Add (Tall 2) (Tall 3)) (Tall 4))
"((2 + 3) * 4)"
ghci> ppInfix (BinOp Add (Tall 2) (BinOp Mult (Tall 3) (Tall 4)))
"(2 + (3 * 4))"
```

Du får ekstra poeng hvis funksjonen ikke skriver ut unødvendige parenteser. F.eks., kan det første uttrykket skrives ut som "2 * 3", det andre som "(2 + 3) * 4" og det siste som "2 + 3 * 4".

I polsk notasjon (også kjent som prefiksnotasjon) er matematiske uttrykk skrevet entydig uten noen parenteser. Operatorene plasseres der før argumentene, f.eks., `(2 + 3) * 5` blir `* + 2 3 5`, mens `2 + 3 * (40 - 1)` blir `* + 2 3 - 40 1`.

Spørsmål 9

Implementer funksjonen `ppPN :: Ast → String` som pretty printer AST i polsk notasjon:

```
ghci> ppPN $ parse "(1 + 2) * 13 - 5"
"- * + 1 2 13 5"
ghci> ppPN $ parse "(1 + 2) * (13 - 5)"
"* + 1 2 - 13 5"
ghci> ppPN $ parse "1 + 2 * 13 - 5"
"- + 1 * 2 13 5"
```

I omvendt polsk notasjon (også kjent som postfixnotasjon) kommer operatorene etter argumentene, f.eks., `(2 + 3) * 5` blir `2 3 + 5 *`, mens `2 + 3 * 5` blir `2 3 5 * +`.

Spørsmål 10

Implementer funksjonen `ppOPN :: Ast → String` som pretty printer AST-en i omvendt polsk notasjon:

```
ghci> ppOPN $ parse "(1 + 2) * 13 - 5"
"1 2 + 13 * 5 -"
ghci> ppOPN $ parse "(1 + 2) * (13 - 5)"
"1 2 + 13 5 - *"
ghci> ppOPN $ parse "1 + 2 * 13 - 5"
"1 2 13 * 5 - +"
```

Del III

En utvidelse

Vi legger til variabler i kalkulatorspråket vårt ved å endre `Term`-linjen i grammatikken til:

```
Term = (Expr) | Num | VariabelNavn
```

Nå kan vi skrive uttrykk som `1 + 42 - x`. Variabelnavn kan være en vilkårlig streng bestående utelukkende av bokstaver. Verdien for `x`, og evt. andre variabler, blir gitt som et ekstra argument til evalueringsfunksjonen.

Spørsmål 11

Utvid `Ast`-typen med konstruktøren `Var String`, for å støtte variabler.

Spørsmål 12

Modifiser tokeniser og parser for å håndtere variabler på riktig måte:

```
ghci> parse "(1 + x) * (y - x)"
BinOp Mult (BinOp Add (Tall 1) (Var "x")) (BinOp Sub (Var "y") (Var "x"))
```

Spørsmål 13

Implementer funksjonen `findVar :: [(String,Int)] → String → Int` som slår opp et variabelnavn i en liste med nøkkel-verdi-par. Funksjonen gir helst en passende feilmelding hvis variabelnavn ikke finnes i listen.

```
ghci> findVar [("z", 17), ("x", 2), ("y",4)] "x"
2
```

Spørsmål 14

Implementer en ny evalueringsfunksjon, `evalVar :: Ast → [(String,Int)] → Int` som evaluerer et uttrykk med hensyn til en variabeltilordning. Funksjonen gir helst en passende feilmelding hvis en variabel som trengs for evaluering ikke får verdi listen.

```
ghci> evalVar (BinOp Add (Tall 41) (Var "x")) [("x", 1)]
42
ghci> evalVar (parse "(1 + x) * x") [("x", 2)]
6
ghci> evalVar (parse "(1 + x) * y") [("x", 2),("y",4)]
12
```

Spørsmål 15

Utvid pp-funksjonene dine slik at de kan skrive ut uttrykk med variabler.

```
ghci> ppPN $ parse "(1 + a) * (13 - b)"
"* + 1 a - 13 b"
ghci> ppOPN $ parse "(1 + a) * (13 - b)"
"1 a + 13 b - *"
ghci> ppOPN $ parse "(1 + a) * 13 - b"
"1 a + 13 * b -"
ghci> ppInfix $ parse "((1 + a) * 13) - b"
"(((1 + a) * 13) - b)"  eller
"(1 + a) * 13 - b"
```

NB! En velorganisert løsning trenger ikke mer enn ca. 100-120 linjer kode.