INF122: Oblig 1

2023

Universitetet i Bergen

Regneark

I denne oppgaven skal vi lage et program for å evaluere regneark.

Et regneark består av en mengde celler som hver har en unik adresse. I cellen står det et aritmetisk uttrykk som kan refere til andre celler. Vi bruker følgende typer til å modellere et regneark:

```haskell
-- | A data type to represent expressions that can be evaluated to a number.
-- This type is parametric over both the number type and the cell reference type.
data Expression number cell
  = Ref cell                      -- ^ Reference to another cell
  | Constant number               -- ^ Constant numerical value
  | Sum (CellRange cell)          -- ^ Summation over a range of cells
  | Add
      (Expression number cell)    -- ^ Left operand of addition
      (Expression number cell)    -- ^ Right operand of addition
  | Mul
      (Expression number cell)    -- ^ Left operand of multiplication
      (Expression number cell)    -- ^ Right operand of multiplication
  deriving (Eq, Ord)

data CellRange cell = Box { upperLeft  :: cell
                          , lowerRight :: cell}
    deriving (Eq, Ord, Show)

-- | A data type representing a sheet.
-- It consists of a name and a mapping from cell references to expressions.
data Sheet number cell = Sheet
  { name :: String,
    -- ^ The name of the sheet
    dimension :: Dimension cell,
    -- ^ The dimension of the sheet
    content :: Map cell (Expression number cell)
    -- ^ The content of the sheet as a mapping from cell references to expressions
  }
```

`Expression` Datatypen `Expression` tar en type for tall og en type for celler og representerer uttrykk som vi kan putte i cellene i regnearket. Det finnes fire forskjellige typer uttrykk vi kan lage:

- En referense til en celle (`Ref cell`).
- Et tall (`Constant number`).
- En sum av celler (`Sum (Set cell)`).
- Addisjon av to uttrykk (`Add (Expression number cell) (Expression number cell)`).
- Multiplikasjon av to uttrykk (`Mul (Expression number cell) (Expression number cell)`).

`Sheet` Typenkonstruktøren `Sheet` tar en type for tall og en type for celler og representerer et regneark. Et regneark består av et navn, dimensjoner og så inneholdet til cellene. Dette inneholdet er lagret i et `Data.Map` som forteller hvilket uttrykk som ligger i hver utfylt celle.

Tekst-representasjon av regneark

Gitt et regneark så kan vi lage en tekstrepresentasjon av det. Vi skriver hver celle i hakeklammer ("[EXPR]") og skriver hver rad med celler som en linje i tekstdokumentet. Den første linjen er spesiell og inneholder navnet på regnearket, fulgt av navnene på kolonnene. Hver linje etter den første begynner med navnet på raden, etterfulgt av cellene i den raden.

For eksempel, følgende ark:

```haskell
-- | A sample spreadsheet using Double for numeric type
sheet1 :: Sheet Double CellRef
sheet1 =
  Sheet
    { name = "Sheet1", -- ^ Name of the sheet
      dimension = Dimension "ABC" [1..3],
      content =
        Map.fromList
          [ ((Cell 'A' 1), Constant 12),
            ((Cell 'B' 1), Mul (Ref (Cell 'A' 1)) (Ref (Cell 'A' 2))),
            ((Cell 'C' 1), Ref (Cell 'C' 3)),
            ((Cell 'A' 2), Constant 4),
            ((Cell 'B' 2), Add (Ref (Cell 'B' 1)) (Ref (Cell 'A' 2))),
            ((Cell 'C' 2), Constant 0),
            ((Cell 'A' 3), Constant 9),
            ( (Cell 'B' 3),
              Sum (Box (Cell 'A' 1) (Cell 'B' 2))
            ),
            ((Cell 'C' 3), Constant 0)
          ]
    }
```

representerer vi som teksfilen:

```
[Sheet1] [A]   [B]           [C]
[1]      [12]  [!A1*!A2]     [!C3]
[2]      [4]   [!B1+!A2]     [0]
[3]      [9]   [SUM(A1:B2)]  [0]
```

Oppgaver

Oppgavene skal leveres individuelt, og du skal kunne gjøre rede for løsningen du har levert. Dersom du sammarbeider med noen andre under løsningen av oppgavene, spesifiser det i en kommentar øverst i filen du leverer inn.

Filen du leverer på CodeGrade skal hete `Oblig1.hs` og inneholde modulen `Oblig1`. Du finner en mal med funksjonssignaturer på MittUiB.

Det er et par steder det står markert "*Ekstrautfordring*", hvor det er en ekstra vanskelighet som er helt frivillig å løse (og som ikke gir ekstra poeng).

Oppgave 0: evaluering av celler

I denne oppgaven skal du skrive en funksjon `evaluate` som tar inneholdet til et regneark og et uttrykk og prøver å evaluere uttrykket og returnere et tall (`number`). Det finnes et par måter evalueringen kan mislykkes, så resultatet av evalueringen blir av typen `Maybe number`.

Før vi skriver selve `evaluate`-funksjonen, så skal vi se på typeklassen `Ranged` som er definert som følger:

```
-- | The ranged class gives a method of indexing ranges of cells
--   in the spreadsheet
class (Ord cell,Show cell) => Ranged cell where
    data Dimension cell
    cellRange :: Dimension cell -> CellRange cell -> Set cell
```

Hensikten med å introdusere typeklassen er at vi ikke nødvendigvis vil fiksere hvilken type som representerer celler i regnearket vårt. Senere i oppgaven bruker vi par av `Char` og `Integer` verdier for enkelhets skyld. Men dersom vi ønsker store regneark med mange kolonner må vi velge en annen type. Typeklassen `Ranged` abstrahérer det vi behøver fra en celletype for å kunne skrive funksjonen `evaluate`, nemlig en funksjon `cellRange` som gir oss alle cellene i en boks utspent av to celler.

Her er den konkrete typen `CellRef` som vi skal bruke til å representere celler:

```
-- | CellRef is the standard way to refer to a cell in the spreadsheet.
data CellRef = Cell { column :: Char, row :: Integer }
  deriving (Eq,Ord)
```

Av dette kan vi se at dimensjonene på et regneark representeres av en liste med kolonner og en liste med rader. Disse listene må være sortert, men ikke komplette, i den forstand at et regneark kan ha kolonner 'A','B' og 'D' uten at kolonne 'C' finnes.

a) Gjør ferdig typeklasseinstansen `Ranged CellRef` ved å skrive funksjonen `cellRange`. Husk å kun returnere celler som passer med dimensjonen. *Hint: Bruk listekomprehensjon og Set.fromList*

Nå skal vi skrive et første utkast av funksjonen `evaluate`. Det er fem ulike tilfeller vi må håndtere:

- Hvis uttrykket er en referanse til en annen cell vil vi evaluere den cellen.
- Hvis uttrykket er et konstant tall skal vi returnere det tallet.
- Hvis uttrykket er en sum av celler skal vi rekursivt evaluere alle cellene og så summere resultatene.
- Hvis uttrykket er addisjon av to uttrykk skal vi rekursivt evaluere enhvert uttrykk og så addere resultatene.
- Hvis uttrykket er multiplikasjon av to uttrykk skal vi rekursivt evaluere enhvert uttrykk og så multiplisere resultatene.

En potensiell feilkilde i rekursjonen er at det kan finnes referenser til en cell som ikke eksisterer i regnearket. Hvis dette skjer skal `evaluate` returnere `Nothing`.

b) Skriv funksjonen `evaluate` som spesifiert ovenfor.

Hint: Bruk pattern matching på uttrykket og rekusjon. Det er praktisk å bruke do-notasjon for Maybe i denne oppgaven. Når du skal evaluere Sum-uttrykk kan funksjonen `sequence :: [Maybe a] -> Maybe [a]` *komme til nytte.*

c) Det finnes en måte til som evalueringen av celleuttrykk kan feile. Dette er demonstrert av regnearket nedenfor. Skriv inn dette arket som `sheet2 :: Sheet Double CellRef` og forsøk å kjøre `evaluate sheet2 (Cell 'C' 2)`.

```
[Sheet2] [A]     [B]     [C]
[1]      [12]    [4*!A2] [!A1+!C2]
[2]      [2]     [4]     [SUM(A1:C1)]
```

d) Tenk over hva som er problemet som får evalueringen du gjorde i c) til å sette seg fast. Finn ut hvordan man kan oppdage slike tilfeller og modifisér `evaluate` slik at den i stedet for å sette seg fast returnerer `Nothing`. Du skal beholde den samme typesignaturen til `evaluate` funksjonen. *Hint: Dersom du behøver å huske mer informasjon i hvert rekursivt kall kan du lage en hjelpefunksjon som tar et ekstra argument.*

Ekstrautfordring: Tenk igjennom hva som skjer dersom en celle referes til av flere ganger. Vil uttrykket i cellen regnes ut flere ganger? Forsøk å gjøre slik at `evaluate` funksjonen kun evaluerer hver celle én gang. Du kan for eksempel bruke et ekstra parameter av typen `Map cell number` for å huske tidligere utregnede verdier. PS: Hvis du synes det er mye bokføring for å gjøre denne implementasjonen, så er det noe som kan løses av StateT-monaden (som vi ikke går igjennom i dette kurset).

Kjøreeksempler Her er noen eksempler som du kan kjøre i GHCi for å teste implementasjonen din i denne oppgaven:

```
-- Test cellRange for values inside and outside the given dimensions
> cellRange (dimension sheet1) (Box (Cell 'A' 1) (Cell 'B' 2))
fromList [A1,A2,B1,B2]
> cellRange (dimension sheet1) (Box (Cell 'A' 1) (Cell 'X' 2))
```

```
fromList [A1,A2,B1,B2,C1,C2]
> cellRange (Dimension "AC" [1..3]) (Box (Cell 'A' 1) (Cell 'C' 2))
fromList [A1,A2,C1,C2]
> cellRange (Dimension "ABC" [1,3]) (Box (Cell 'A' 1) (Cell 'C' 3))
fromList [A1,A3,B1,B3,C1,C3]

-- Test evaluation function on the example sheet
> evaluate sheet1 (Ref $ Cell 'A' 1)
Just 12.0
> evaluate sheet1 (Ref $ Cell 'X' 1)
Nothing
> evaluate sheet1 (Add (Ref $ Cell 'A' 1) (Ref $ Cell 'B' 2))
Just 64.0
> evaluate sheet1 (Ref $ Cell 'B' 3)
Just 116.0
> sheet2
[Sheet2] [A]    [B]          [C]
[1]      [12.0] [(4.0*!A2)] [(!A1+!C2)]
[2]      [2.0]  [4.0]       [SUM(A1:C1)]

-- Test the improved implementation in d)
> evaluate sheet2 (Ref $ Cell 'C' 2)
Nothing
> evaluate sheet2 (Ref $ Cell 'C' 1)
Nothing
> evaluate sheet2 (Ref $ Cell 'B' 1)
Just 8.0
```

Oppgave 1: Parsing

Vi må lage en parser som kan lese en tekstrepresentasjon av et regneark og lage et element av typen `Sheet`. Vi bruker samme teknikk som vi har brukt i forelesning, med parser kombinatoreren `<|>` ("eller") og do-notasjon.

Vi bruker følgende type for parsere (en forenkling av parseren i artikelen Monadic Parsing in Haskell):

```haskell
newtype Parser a = Parser {runParser :: String -> Maybe (String, a)}
```

En slik parser tar en streng som input og prøver å parse et element av typen `a`. Hvis begynnelsen av strengen kan parses som et element av type `a` returnerer elementet sammen med resten av strengen, hvis ikke returneres `Nothing`

`Parser` er en monade, som betyr at vi kan bruke `do`-notasjon, slik at koden blir mer lettleselig. Typeklasseinstansene for Parser er inkludert i det gitte oppgavekoden.

Som oppvarming skal vi se på et par av parserne som er gitt i kildekoden fra før, og se om vi kan abstrahére dem litt. Vi begynner med parserne for space og newline.

```haskell
pSpace :: Parser ()
pSpace = do
```

```
    c <- pChar
    guard (c == ' ')

-- | Eat a single newline
pNewLine :: Parser ()
pNewLine = do
    c <- pChar
    guard (c == '\n')
```

Disse følger det samme mønsteret, så det kan gi mening å lage en funksjon som abstraherer dette mønsteret.

a) Skriv en funksjon `exactChar :: Char -> Parser ()` slik at `pNewLine=exactChar '\n'` og `pSpace = exactChar ' '`.

Ekstrautfordring: Skriv om `keyword` parseren ved hjelp av `foldr`, `exactChar` og `>>`-operatoren. Klarer du å skrive den på en punktfri måte?

To andre funksjoner som følger et mønster er `inBrackets` og `inParenthesis`:

```
-- | Parse parenthesis
inParenthesis :: Parser a -> Parser a
inParenthesis p = do
    keyword "("
    x <- p
    keyword ")"
    return x

-- | Parse brackets
inBrackets p = do
    keyword "["
    x <- p
    keyword "]"
    return x
```

b) Skriv en funksjon `between :: Parser a -> Parser b -> Parser c -> Parser c`, slik at `inParentehsis = between (exactChar '(') (exactChar ')')` og `inBrackets = between (exactChar '[' ) (exactChar ']')`.

Nå skal vi skrive parseren som parser et celleuttrykk. Vi skal gjøre det i flere steg, ved å skrive en parser for hver konstruktør av `Expression`. Men først behøver vi en parser for celleadresser, slik som `A1` eller `X19`.

c) Skriv `pCell :: Parser CellRef` som parser kolonnenavn (som en `Char`) og radnummer (som en `Integer`).

d) Skriv `pConstant :: (Read number) => Parser (Expression number CellRef)` som parser tall og gjør dem til konstanter. *Hint: Du kan bruke hjelpefunksjonen* ***`pRead`*** *som konverterer en* ***`Read`****-instans til en parser.*

e) Skriv `pRef :: Parser (Expression number CellRef)` som parser en cellereferanse på formen `!C9`. *Hint: Husk at du kan bruke* ***`pCell`*** *til å parse celleadressen.*

f) Skriv `pSum :: (Read number) => Parser (Expression number CellRef)` som parser summer på formen `SUM(A1:C8)`.

De tre delparserene av `Expression` som vi har skrevet frem til nå er ikke-rekursive, og har prefix syntaks. For eksempel begynner alle cellereferanser på `!` og alle summer med `SUM`.

De neste to, `pAdd` og `pMul`, er rekursive og infix. I uttrykkene kommer `+` og `*` i midten, og de kan være nøstet slik som `3+4*(!A1+!B2)`. Det gjør at man må være litt forsiktig slik at ikke parseren ender opp med å gå i en uendelig rekursjon. Dessuten må parseren bli klar over presedensreglene mellom `+` og `*`.

Løsningen er å sørge for at man alltid leser inn minst en av de andre tre typene uttrykk, eller en parentes, før man leser `+` eller `*`. Dette gjør vi ved å definere en hjelpe-parser:

```
pTerm :: Read number => Parser (Expression number CellRef)
pTerm =  inParenthesis pExpression
     <|> pConstant
     <|> pRef
     <|> pSum
```

Deretter bruker vi `pTerm` på venstresiden av `+` eller `*` før vi gjør resten av rekursjonen. Vi fanger dette mønsteret ved hjelp av følgende funksjon:

```
-- | Parse an operator
pOperator :: String -> (t -> t -> t) -> Parser t -> Parser t
pOperator symbol constructor pOperand = do
  a <- pOperand
  rest a
  where
    rest a = (do
      keyword symbol
      b <- pOperand
      rest (constructor a b)) <|> return a
```

Legg merke til at `rest` er rekursiv slik at vi kan fange opp både lengre addisjoner/multiplikasjoner, slik som "1+2+3+4", og nøstede uttrykk "(1+2)*(3+4)".

g) Bruk `pOperand` for å implementere `pAdd` og `pMul`, som begge skal ha typen `Read number => Parser (Expression number CellRef)`. Sørg for at presedensreglene for addisjon og multiplikasjon overholdes. *Hint: Den ene av disse to parserene må refere til den andre for at det skal bli korrekt.*

h) Skriv ferdig parseren for celleuttrykk, `pExpression`. *Hint: Sett delparserene sammen i riktig rekkefølge.*

Kjøreeksempler

```
> runParser pCell "A1" :: Maybe (String,CellRef)
Just ("",A1)
> runParser pCell "1A" :: Maybe (String,CellRef)
Nothing
> runParser pRef "!X7" :: Maybe (String,Expression Integer CellRef)
```

```
Just ("",!X7)
> runParser pConstant "99" :: Maybe (String,Expression Integer CellRef)
Just ("",99)
> runParser pSum "SUM(A1:X3)" :: Maybe (String,Expression Integer CellRef)
Just ("",SUM(A1:X3))
> runParser pMul "3*3" :: Maybe (String,Expression Integer CellRef)
Just ("",(3*3))
> runParser pMul "1*2*3*4" :: Maybe (String,Expression Integer CellRef)
Just ("",(((1*2)*3)*4))
> runParser pMul "(1*!A2)*(3*!B4)" :: Maybe (String,Expression Integer CellRef)
Just ("",((1*!A2)*(3*!B4)))
> runParser pAdd "3+3" :: Maybe (String,Expression Integer CellRef)
Just ("",(3+3))
> runParser pAdd "1*2+3*4" :: Maybe (String,Expression Integer CellRef)
Just ("",((1*2)+(3*4)))
> runParser pExpression "SUM(A1:B3)*3+(1+!A1)*(3*!A2+-3)+1"
    :: Maybe (String,Expression Integer CellRef)
Just ("",(((SUM(A1:B3)*3)+((1+!A1)*((3*!A2)+-3)))+1))
```

Her er et komplett eksempel:

```
*Oblig1> sheet3 <- getSpreadSheet "sheet-3.txt"
*Oblig1> sheet3
[Sheet3] [A]    [X]          [Y]
[1]      [12.0] [(4.0*!A2)] [(!A1+!Y1)]
[2]      [2.0]  [4.0]        [SUM(A1:X2)]

*Oblig1> runSpreadSheet "sheet-3.txt"
[Sheet3] [A]    [X]   [Y]
[1]      [12.0] [8.0]
[2]      [2.0]  [4.0] [26.0]
```