Laboration #3: ***Interpretator för imperativa språket MAP***

Deadline 1: 211018 klockan 13:00

Syftet med den här laborationen är att ge er träning i att stegvis utveckla ett program vilket utnyttjar rekursiva datatyper. Då programmet och dess funktion är relativt komplext sker utvecklingen stegvis. Det är både möjligt och lämpligt att testa det utvecklade programmet efter varje steg.

Laborationen innebär att skapa en interpetator för ett enkelt imperativt språk kallat MAP. Varför språket kallas MAP – alltså vad MAP betyder utläst – överlåter jag åt er studenter att välja/gissa. (Det har absolut ingenting med map i Haskell att göra utan det är en akronym.) Mer specifikt skall bara semantiken och meningsfunktionerna kodas. Laborationen utgår från en semantik där den abstrakta syntaxen ges som rekursiva datatyper. Observera att programmet därmed skall utgå från ett abstrakt syntaxträd, dvs. lexikalanalysen och parsningen förutsätts vara gjord. I språket som vi skall implementera, alltså MAP, är alla variabler och värden av typen heltal, dvs. det finns inga Booleska variabler. Detta innebär att en lösning där vi tydligt skiljer på aritmetiska och Booleska uttryck är önskvärd.

Introduktion

De flesta programspråk är imperativa, vilket innebär att ett program i princip körs stegvis, ett kommando (eller en sats) i taget, från början av källkoden till slutet. Det finns dessutom ett antal kontrollstrukturer som t.ex. gör att vissa satser upprepas ett visst antal gånger. Det finns även selektion i form av en if-sats, där exekveringen fortsätter med olika delar av programmet beroende på utfallet av ett visst test, dvs. värdet på ett Booleskt uttryck.

En central sak för imperativa språk är dess hantering av variabler. En avgörande skillnad mot Haskell är att variabler kan ändra värden under programmets körning. Därmed blir också tilldelningar, alltså att ge en variabel ett värde, en viktig del av imperativa program.

I MAP är de enda variablerna av heltalstyp, och variabelnamnen representeras som strängar. Vi behöver i varje steg av programmet hålla reda på exakt vilka värden varje variabel har. Vi kallar detta för en *omgivning* eller ett *tillstånd* (state). Vi väljer att i Haskell representera omgivningen som en lista av par, där första delen av paret är variabelnamnet, och andra variabelns värde.

Vi kommer i den här laborationen att använda type-definitioner flitigt, och vi väljer även att tydligt separera semantiken från den interna representationen. Detta innebär att vi börjar med följande fyra deklarationer:

```
type Binding = (Variable, Value)
type State = [Binding]
```

```
type Variable = String
data Value = Intval Integer deriving (Eq, Ord, Show)
```

Så, ett tillstånd består av ett antal bindningar mellan variabelnamn och värden i en lista. Notera även att vi alltså inte direkt använder en inbyggd heltalstyp, utan skapar en egen typ kallad Value, som i sin konstruktor Intval tar en Integer. Det här innebär att talet 5 representeras som Intval 5.

I språket MAP måste alla variabler vara deklarerade och initierade. Detta innebär enkelt uttryckt att de finns i omgivningen och har värdet 0. Det vi skall kunna göra med vår omgivning är därmed följande: hämta ut ett värde för en viss variabel, samt ändra värdet för en variabel som finns i omgivningen. Notera alltså att vi kan förutsätta att den variabeln vi skall hämta värdet för, alternativt skall ändra värdet på, finns i omgivningen. Om så inte är fallet är MAP-programmet inte korrekt, vilket er interpretator inte behöver hantera.

Uppgift 1

- Skriv en funktion get som tar en Variable och ett State och returnerar motsvarande Value. Signaturen blir alltså: `get :: Variable -> State -> Value`.

- Skriv en funktion onion (betyder overwriting union) som tar en Variable, ett Value samt ett State och returnerar ett nytt State där variabelns värde ändrats.

Notera att i båda funktionerna kan ni alltså anta att variabeln ni letar efter finns i omgivningen, dvs. ni behöver inte på något sätt hantera fallet att den inte hittas.

Testa så dessa båda funktioner fungerar som avsett.

Uppgift 2

Vi skall nu börja implementera några av nyckeldelarna av vår interpretator. Vi väljer att börja med en datatyp för uttryck, vilket alltså handlar om aritmetiska uttryck, och deklarerar följande datatyp (jämför gärna med föreläsningen).

```
data Expression = Var Variable |
      Lit Value |
      Aop Op Expression Expression-- Aritmetic operators
      deriving (Eq, Ord, Show)

type Op = String
```

Funktionen eval nedan tar in ett uttryck (Expression) och ett tillstånd (State) och returnerar ett värde (Value). Notera att Variable alltså är en sträng, medan Var Variable är ett uttryck. Funktionen mönstermatchar mot de tre fall som finns i datatypen Expression ovan.

```
eval:: Expression -> State -> Value
eval (Var v) state = get v state
eval (Lit v) state = v
eval (Aop op e1 e2) state = apply op (eval e1 state) (eval e2 state)
```

Det är centralt för resten av laborationen att ni förstår hur eval är tänkt att fungera. Ni skall nu skriva en hjälpfunktion med signaturen `apply :: Op -> Value -> Value -> Value` som utför själva operationen. De aritmetiska operationer som skall finnas med är addition, subtraktion, multiplikation och heltalsdivision. Dessa motsvaras i MAP av operatorerna ”+”, ”-”, ”*” och ”/”. apply kommer då såklart att mönstermatcha på dessa fyra operatorer.

Pröva nu en stor mängd uttryck för att se att allting verkligen fungerar. Nedan ses några exempel.

```
ghci> s1=[("x",(Intval 1)) ,("y",(Intval 5))]
ghci> eval (Var "x") s1
Intval 1
ghci> eval (Aop "+" (Var "x") (Lit (Intval 4))) s1
Intval 5
ghci> eval (Aop "+" (Var "x") (Aop "*" (Lit (Intval 4)) (Var "y"))) s1
Intval 21
```

Uppgift 3

Vi väljer därefter att skapa en datatyp för Booleska uttryck.

```
data Bexpression = Blit Bvalue |
   Bop Op Bexpression Bexpression | -- Boolean operators
   Rop Op Expression Expression   -- Relational operators
   deriving (Eq, Ord,Show)

data Bvalue = Boolval Bool deriving (Eq, Ord,Show)
```

Notera att det alltså enligt ovan inte finns några Booleska variabler.

- Skriv nu en funktion beval:: Bexpression -> State -> Bvalue med tillhörande hjälpfunktioner. Funktionen skall utgå från datatypen ovan, och mönstermatcha mot de tre olika fallen. De Booleska operatorerna som ingår i MAP är ”&&” (betyder and) och ”||” dvs. or. De relationella operatorer som ni skall implementera är ”<”, ”<=”, ”==”, ”!=”, ”>=” och ”>” där ”!=” betyder ”inte lika med” och alla andra har sin uppenbara betydelse.

Det här kan kännas som en svår och omfattande uppgift, men jämför med lösningen för de numeriska uttrycken ovan, och tänk noga efter vilken betydelse de olika operatorerna har.

TIPS: Dela upp det hela i tre funktioner; beval enligt ovan samt bapply och rapply som hanterar Booleska respektive relationella operator på motsvarande sätt som ni gjorde med apply för numeriska uttryck.

Ni bör nu igen testa ordentligt för att säkerställa att allt fungerar. Nedan är ett par exempel.

```
ghci> beval (Bop "&&" (Blit (Boolval True)) (Rop "<" (Var "x") (Lit (Intval 3)))) s1
Boolval True
ghci> beval (Bop "&&" (Rop "<" (Var "y") (Lit (Intval 2))) (Rop "<" (Var "x") (Lit
(Intval 3)))) s1
Boolval False
```

Uppgift 4

Nu har vi lagt grunden för vår interpretator, dvs. det är dags att börja med vår meningsfunktion. Denna funktion tar ett kommando och ett tillstånd för att returnera ett tillstånd. Vi representerar våra kommandon som ytterligare en datatyp. Vi kommer beskriva de olika alternativen nedan.

```
data Statement    = Skip |
        Assignment Target Source |
        Block Blocktype |
        Loop Test Body |
        Conditional Test Thenbranch Elsebranch
        deriving (Show)

type Target = Variable
type Source = Expression
type Test = Bexpression
type Body = Statement
type Thenbranch = Statement
type Elsebranch = Statement
```

Det som nu skall göras är att skriva själva meningsfunktionerna, dvs. vad som görs i de olika alternativen.

```
m:: Statement -> State -> State
m  (Skip) state  ??
m  (Assignment target source) state ??
```

```
m  (Loop t b) state ??
m  (Conditional test thenbranch elsebranch) state ??
```

De kommandon som finns är följande:

- Tilldelning: en variabel ges ett nytt värde. Värdet fås från att ett godtyckligt komplext uttryck beräknas.
- Loop: En sats körs upprepat så länge ett villkor är sant.
- If-sats: Om ett villkor är sant exekveras ett kommando, om det är falskt ett annat.

Resultatet av att exekvera en sats är ofta att variablers värden (alltså omgivningen) ändras.

Här är tre exempel med `s1=[("x",(Intval 1)) ,("y",(Intval 5))]`

p0::Statement -- An assignment
p0 = (Assignment "x" (Aop "+" (Var "x") (Lit (Intval 1))))

ghci> m p0 s1
[("x",Intval 2),("y",Intval 5)]

p1::Statement -- A loop.
p1=(Loop (Rop "<" (Var "x") (Lit(Intval 10))) (Assignment "x" (Aop "+" (Var "x") (Lit(Intval 1)))))

ghci> m p1 s1
[("x",Intval 10),("y",Intval 5)]

p2::Statement -- An IF-statement.
p2=(Conditional (Rop ">" (Var "x") (Var "y")) (Assignment "x" (Var "y")) (Assignment "x" (Aop "+" (Var "x") (Var "y"))))

ghci> m p2 s1
[("x",Intval 6),("y",Intval 5)]

Återigen bör ni testa att allt fungerar som det skall!

Uppgift 5

I sista steget behöver vi kunna sätta ihop flera satser – som därmed skall köras i sekvens – till ett program. Vi inför därför en sista typ Block:

```
data Blocktype = Nil |
 Nonnil Statement Blocktype
 deriving (Show)
```

Betydelsen av Nil är att blocket är slut. Om vi t. ex. skall köra två satser c1 och c2 i sekvens blir blocket:

```
p=(Block (Nonnil c1 (Nonnil c2 Nil)))
```

Ni behöver nu utöka meningsfunktionen så att den hanterar block. Notera att när ni är färdiga med detta kan ni i princip skapa godtyckligt långa sekvenser av kommandon, dvs. ett block är i själva verket ett MAP-program, vilket ni kör genom att anropa meningsfunktionen med blocket som Statement och en lämplig omgivning.

Då MAP inte har någon I/O blir resultatet av att köra programmet helt enkelt en ny omgivning, dvs. vilka värden de olika variablerna har.

Konstruera ett antal MAP-program och testkör dem. Se även avsnittet om körningsexempel i inlämningen.

Organisation

Laborationen skall lösas i par. Lämna in laborationen från gruppen i Canvas.

Inlämningen skall bestå av två filer:

1. Källkoden.
2. Väl valda körningsexempel. Ett körningsexempel här innebär att ni testkör hela MAP-program. Ni behöver inte visa separat att varenda konstruktion i er lösning fungerar, utan välj hellre några illustrativa exempel – alltså sådana som utnyttjar många av konstruktionerna i MAP - vilka ni beskriver och kommenterar. Försök att redovisa åtminstone två MAP-program som gör något ”vettigt”. Ett exempel kan vara att det beräknar x!, vilket då innebär att x ligger i variabeln x från början och värdet på x! ligger i t. ex. y då programmet är klart. En lämplig mängd körningsexempel kan vara fem – tre väldigt enkla som visar olika konstruktioner med små MAP-program, och då två lite större.

Ni har tillgång till två handledningar per grupp. Dessa genomförs online och bokas via Canvas. På Canvas finns även ett diskussionsforum kopplat till laborationen där ni kan ställa generella frågor så att alla får samma information. Notera att ingen extra handledning kommer ges, exempelvis via e-mail. Slutligen, kom ihåg att samarbeten mellan grupper är förbjudet då laborationen utgör del av examinationen.

Lycka till!