

Lösung Übungsblatt Nr. 6 / ALP 1 zur Abgabe 16.01.2004

```
-- Algorithmen und Programmieren I
-- Wintersemester 2003/2004
-- Raul Rojas / Alexander Gloye
-- Übung 6
-- Freie Universität Berlin

-----
-- Das ist die Hauptfunktion. Ein in Lambda-Schreibweise übergebener
-- Ausdruck wird reduziert und das Ergebnis in Lambda-Schreibweise
-- zurück gegeben.
-----

reduce::String->String
reduce x | rest == [] = show_eval (normalize (head treeinlist))
          | otherwise = error ("***Eingabe konnte nicht vollständig bearbeitet werden. Rest = '"+rest+"'***")
          where (treeinlist,rest) = parse x

-----

data Expr = Lam String Expr | App Expr Expr | Paren Expr | Var String
          deriving (Eq)

-----
-- Die folgenden Funktionen sind nur für den Parser notwendig.
-----

-- Gibt den Wert einer Ziffer zurück
ctoi::Char->Int
ctoi x = (ord x) - (ord '0')

-- Fügt rechts einen Ausdruck ein
compose::Expr->Expr->Expr
compose x (App y z) = App (compose x y) z
compose x y = (App x y)

-- Makronamen suchen
parseName name (' ':rest) = (reverse name,rest)
parseName name (x:rest) = parseName (x:name) rest

-- Makronamen suchen
parseNum::Int->String->(Int,String)
parseNum num (' ':rest) = (num,rest)
parseNum num (x:rest) = parseNum (num*10+(ctoi x)) rest

-- Test auf Ziffer
isCypher x = x>= '0' && x<= '9'

-- Makronamen suchen
parseSName name ([]) = (reverse name,[])
parseSName name (' ':rest) = (reverse name,rest)
parseSName name (x:rest) | isUpper x = parseSName (x:name) rest
                          | otherwise = (reverse name,(x:rest))

-- Makronamen suchen
parseSNum::Int->String->(Int,String)
parseSNum num ([]) = (num,[])
parseSNum num (' ':rest) = (num,rest)
parseSNum num (x:rest) | isCypher x = parseSNum (num*10+(ctoi x)) rest
                       | otherwise = (num,x:rest)

-----
-- Der Parser: Wandelt einen String in einen Ausdruck um
-----

parse::String->([Expr],String)

parse [] = ([],[])

parse (' ':xs) = parse xs -- Leerzeichen werden übersprungen.

parse ('{':y:xs) | isCypher y = parse ((nummacro num)++nrest)
                 | otherwise = parse ((macro mname)++rest)
                 where (num,nrest) = parseNum 0 (y:xs)
                       (mname,rest) = parseName "" (y:xs)

parse ('/':x:'.':xs) | expr == [] = error ("***Lambda '"+[x]+"' without a body (from '"+xs+"'").
rest='"+rest+"'***")
                    | otherwise = ([Lam [x] (head expr)], rest)
                    where (expr,rest)= parse xs

parse ('/':x:xs) = parse ('/':x:"./"++xs)
```

Lösung Übungsblatt Nr. 6 / ALP 1 zur Abgabe 16.01.2004

```
parse (')':xs) = ([],')':xs)

parse ('(':xs) | rest == [')'] = ([Paren (head treeinlist)],[])
  | head rest == ')' = if resttreeinlist /= []
    then ([compose (Paren (head treeinlist)) (head resttreeinlist)],restrest)
    else ([Paren (head treeinlist)],restrest)
  | otherwise = error "***Vermisste eine ')'"***
  where (treeinlist,rest) = parse xs
        (resttreeinlist,restrest) = parse (tail rest)

parse (x:xs) | isCppher x = parse ((nummacro num)++nrest)
  | isUpper x = parse ((macro mname)++mrest)
  | treeinlist /= [] = ([compose (Var [x]) (head treeinlist)],rest)
  | otherwise = ([Var [x]],rest)
  where (treeinlist,rest) = parse xs
        (num,nrest) = parseSNum 0 (x:xs)
        (mname,mrest) = parseSName "" (x:xs)

-----

-- Übung 6 Aufgabe 1
show_expr::Expr->String
show_expr (Var x) = x
show_expr (Lam x y) = "(" ++ x ++ "." ++ show_expr y ++ ")"
show_expr (App x y) = "(" ++ (show_expr x) ++ (show_expr y) ++ ")"

-----

-- Eine kleine Erleichterung
show_eval::Expr->String
show_eval x = show_expr (eval x 0)

-----

-- Parst einen String und gibt eine Ausdruck zurück.
parse_expr x = (normalize (head treeinlist)) where { (treeinlist,rest) = parse x }

-----

-- Entfernt Klammern aus einem Ausdruck. Die Klammern werden nur zum
-- Aufbau des Baumes beim Parsen gebraucht.
normalize::Expr->Expr
normalize (Paren e) = normalize e
normalize (App e1 e2) = App (normalize e1) (normalize e2)
normalize (Lam v e) = Lam v (normalize e)
normalize e = e

-----

-- Übung 6 / Aufgabe 2
freie::Expr->[String]->[String]
freie (Var x ) list | elem x list = []
  | otherwise = [x]
freie (Lam x e) list = freie e (x:list)
freie (App x y) list = (freie x list) ++ (freie y list)

-----

-- Übung 6 / Aufgabe 3
bounded::Expr->[String]->[String]
bounded (Var x ) = []
bounded (Lam x y) = (x:(bounded y))
bounded (App x y) = (bounded x) ++ (bounded y)

-----

-- Übung 6 / Aufgabe 1
is_lambda :: Expr -> Bool
is_lambda (Lam _ _) = True
is_lambda _ = False

-----

-- Übung 6 / Aufgabe 4
find_new_name :: String -> [String] -> String
find_new_name (testthis:_) liste | elem [testthis] liste = find_new_name [(chr(ord(testthis)+1))] liste
  | otherwise = [testthis]

find_new_name :: [String] -> String
```

Lösung Übungsblatt Nr. 6 / ALP 1 zur Abgabe 16.01.2004

```
find_new_name liste = find_new_name x "a" liste
```

```
-----  
-- Übung 6 / Aufgabe 5  
rename :: String -> String -> Expr -> Expr  
rename s r (Var x) | x == s = Var r  
                  | otherwise = Var x  
rename s r (Lam x e) | x == s = Lam r (rename s r e)  
                    | otherwise = Lam x (rename s r e)  
rename s r (App x y) = App (rename s r x) (rename s r y)
```

```
-----  
-- Ersetzt alle x in y durch a  
subst x a y = substitute x a y (freie a [])
```

```
-----  
-- Übung 6 / Aufgabe 6  
-- substitute :: String -> Expr -> Expr -> [String] -> Expr
```

```
-- nicht geschafft, sorry
```

```
-----  
-- Auswerten eines Lambda-Ausdrucks.
```

```
eval :: Expr -> Int -> Expr
```

```
eval (Var x) n = Var x  
eval (App (Lam x y) a) n = eval (subst x a y) n  
eval (Lam x y) n  
    | n == 0 = (Lam x (eval y 0))  
    | otherwise = (Lam x y)
```

```
eval (App x y) n  
    | is_lambda eval_x = eval (App eval_x y) n  
    | otherwise = (App eval_x (eval y 0))  
    where eval_x = eval x 1
```

```
-----  
-- Vordefinierte Makros
```

```
nummacro :: Int -> String  
nummacro 0 = "(/s.(/z.z))"  
nummacro x = "({S}++nummacro (x-1)++)"
```

```
macro ("T") = "(/a./b.a)"  
macro ("F") = "(/a./b.b)"
```

```
macro ("S") = "(/w./x./y.x(wxy))"
```

```
macro ("AND") = "(/x.(/y.xy{F}))"  
macro ("OR") = "(/x.(/y.x{T}y))"  
macro ("NOT") = "(/x.x{F}{T})"
```

```
macro ("Z") = "(/x.x{F}{NOT}{F})"
```

```
macro ("PAR00") = "(/z.z{0}{0})"  
macro ("PHI") = "(/pz.z({S}(p{T})) (p{T}))"  
macro ("P") = "(/n.n{PHI}{PAR00}{F})"
```

```
macro ("GE") = "(/x.(/y.{Z}(x{P}y))"
```

```
macro ("SUM") = "(/rn.{Z}n{0}(n{S}(r({P}n)))" -- Summe
```

```
macro ("REC") = "(/y.(/x.y(xx))(/x.y(xx))" -- Rekursion
```

```
macro ("FAKUL") = "(/rn.{Z}n{1}({MUL}n(r({P}n)))" -- mit (z.B.) REC FAKUL 5  
macro (x) = error ("***Makro '++x++' undefiniert.***")
```