Local Variables

Haskell não possui a definição de variável. Nós não podemos, por exemplo, declarar uma variável global e alterá-la dentro de funções. Também não podemos declarar uma variável em um certo ponto da função e alterá-la ainda dentro do escopo dela.

O que podemos é declarar valores que podem ser usados no decorrer do código, mas são valores definidos em um único ponto. Valores estáticos, como a string abaixo:

```
melhorLinguagem = "Haskell"
```

Se tentarmos verificar o tipo de melhorLinguagem, veremos que trata-se de uma String, ainda que não tenhamos dito isso a ela. No entanto, podemos também dizer qual o tipo dela, de forma semelhante a que usamos para definir os tipos de entrada e saída de uma função:

```
melhorLinguagem :: String
```

Ainda sobre variáveis, o que podemos fazer é, no máximo, declarar "variáveis" locais que carregam consigo definições.

Ao invés de um código cheio de cálculos e expressões gigantes cujo entendimento requer pausa e atenção, é preferível um código limpo, organizado com entendimento descomplicado. Um bom alvo, inclusive, é produzir código que seja tão fácil de ser lido quanto a própria linguagem nativa. Na tentativa de prover essa fluidez, surgem as variáveis locais. A expressão abaixo recebe dois argumentos e utiliza duas keywords: let, marcando o início da declaração das variáveis e in, iniciando o código propriamente dito. Em muitos casos, essas "variáveis locais" tem mais cara de função. Uma função que só pode ser acessada dentro do escopo da função na qual foi criada.

```
lend amount balance = let reserve = 100
                          newBalance = balance - amount
                      in if balance < reserve
                         then Nothing
                         else Just newBalance
```

Outra forma seria utilizando where, onde a declaração das variáveis é feita posteriormente, o que, na opinião de alguns, deixa o código mais flúido que a primeira. let inicia a definição e in indica em quais expressões utilizaremos essas definições. No caso de where, as expressões aparecem antes e as definições são feitas nele.

```
lend2 amount balance = if amount < reserve * 0.5
                       then Just newBalance
                       else Nothing
        where reserve = 100
              newBalance = balance - amount
```

Aqui, temos outra aplicação de where criando algo que se assemelha a casamento de padrões.

```
pluralise :: String -> [Int] -> [String]
pluralise word counts = map plural counts
       where plural 0 = "no " ++ word ++ "s" -- Plural is a local function
             plural 1 = "one " ++ word
             plural n = show n ++ " " ++ word ++ "s"
```

Já nos exemplos abaixo, vemos a mesma função feita de formas diferentes. Na primeira, utilizando let e in enquanto que, na segunda, é feito o uso de map e where com o auxílio de uma função lines que recebe uma String e separa o que precede '\n' do que o sucede, caso o encontre.

```
splitLines :: String -> [String]
splitLines [] = []
splitLines cs =
    let (pre, suf) = break isLineTerminator cs
        isLineTerminator c = c == '\r' || c == '\n'
    in pre : case suf of
               ('\r':'\n':rest) -> splitLines rest
               ('\r':rest) -> splitLines rest
               ('\n':rest) -> splitLines rest
               _ -> []

splitLineS :: String -> [String]
splitLineS [] = []
splitLineS cs = map deleteLineTerm (lines cs)
    where deleteLineTerm x | isLineTerminatoR (last x) = deletLeast x
                           | otherwise = x
          isLineTerminatoR c = c == '\r'
          deletLeast m = take ((length m) - 1) m
```