Exemplo:

```
type Notas = [(Integer,String,Int,Int)]

texto = "1234\tPedro\t15\t17\n1111\tAna\t16\t13\n"
```

```
leFich :: IO ()
leFich = do file <- dialogo "Qual o nome do ficheiro ? "
            s <- readFile file
            let l = map words (lines s)
                notas = geraNotas l
            print notas
```

```
geraNotas :: [[String]] -> Notas
geraNotas [] = []
geraNotas ([x,y,z,w]:t) = let x1 = (read x)::Integer
                              z1 = (read z)::Int
                              w1 = (read w)::Int
                          in (x1,y,z1,w1):(geraNotas t)
```

```
escFich :: Notas -> IO ()
escFich notas = do file <- dialogo "Qual o nome do ficheiro ? "
                   writeFile file (geraString notas)
```

```
geraStr :: Notas -> String
geraStr [] = "\n"
geraStr ((x,y,z,w):t) = (show x) ++ ('\t':y) ++ '\t':(show z) ++
                        '\t':(show w) ++ "\n" ++ (geraStr t)
```


O mónade Maybe

A declaracão do construtor de tipos `Maybe` como instância da classe `Monad` é muito util para trabalhar com computações parciais, pois permite fazer a propagação de erros.

```
instance Monad Maybe where
    return x       = Just x
    (Just x) >>= f = f x
    Nothing >>= _  = Nothing
    fail _         = Nothing
```

Exemplo:

```
exemplo :: Int -> Int -> Int -> Maybe Int
exemplo a b c = do x <- return a
                   y <- return b
                   z <- divide x y
                   w <- soma c z
                   return w
```

Podemos simplificar ?

```
divide :: Int -> Int -> Maybe Int
divide _ 0 = Nothing
divide x y = Just (div x y)
```

```
soma :: Int -> Int -> Maybe Int
soma x y = Just (x+y)
```


Módulos

Um programa Haskell é uma colecção de **módulos**. A organização de um programa em módulos cumpre dois objectivos:

- criar componentes de software que podem ser usadas em diversos programas;
- dar ao programador algum control sobre os identificadores que podem ser usados.

Um módulo é uma declaração "gigante" que obdece à seguinte sintaxe:

module *Nome (entidades_a_exportar)* **where**

declarações de importações de módulos

declarações de: tipos, classes, instâncias, assinaturas, funções, ...
(por qualquer ordem)

Cada módulo está armazenado num ficheiro, geralmente com o mesmo nome do módulo, mas isso não é obrigatório.

Na declaração de um módulo:

- pode-se indicar explicitamente o conjunto de tipos / construtores / funções / classes que são exportados (i.e., visíveis do exterior)

  *Aos vários items que são exportados ou importados chamaremos **entidades**.*

- por defeito, se nada for indicado, todas as declarações feitas do módulo são exportadas;
- é possível exportar um tipo algébrico com os seus construtores fazendo, por exemplo: `ArvBin(Vazia, Nodo)`, ou equivalentemente, `ArvBin(..)`;
- também é possível exportar um tipo algébrico e não exportar os seus construores, ou exportar apenas alguns;
- os métodos de classe podem ser exportados seguindo o estilo usado na exportação de construtores, ou como funções comuns;
- declarações de instância são sempre exportadas e importadas, por defeito;
- é possível exportar entidades que não estão directamente declaradas no módulo, mas que resultam de alguma importação de outro módulo.

Qualquer entidade visível no módulo é passível de ser exportada por esse módulo.

Na importação de um módulo por outro módulo:

- é possível fazer a importação de todas as entidades exportadas pelo módulo fazendo

```
import Nome_do_módulo
```

- é possível indicar explicitamente as entidades que queremos importar, fazendo

```
import Nome_do_módulo (entidades a importar)
```

- é possível indicar selectivamente as entidades que não queremos importar (importa-se tudo o que é exportado pelo outro módulo excepto o indicado)

```
import Nome_do_módulo hiding (entidades a não importar)
```

- é possível fazer com que as entidades importadas sejam referenciadas indicando o módulo de onde provêm como prefixo (seguido de '.') fazendo

```
import qualified Nome_do_módulo (entidades a importar)
```

(Pode ser util para evitar colisões de nomes, pois é ilegal importar entidades diferentes que tenham o mesmo nome. Mas se for o mesmo objecto que é importado de diferentes módulos, não há colisão. Uma entidade pode ser importada via diferentes caminhos sem que haja conflitos de nomes.)

Um exemplo com módulos

Considere os módulos: `Listas`, `Arvores`, `Tempo`, `Horas` e `Main`, que pretendem ilustrar as diferentes formas de exportar e importar entidades.

```
module Listas where

soma [] = 0
soma (x:xs) = x + (soma xs)

conta = length

naLista x [] = False
naLista x (y:ys) = if x==y then True
                           else naLista x ys

mult = product

cauda (_:xs) = xs
```



```
module Arvores(ArvBin(Vazia,Nodo), naArv, soma, mult) where

data ArvBin a = Vazia
              | Nodo a (ArvBin a) (ArvBin a)
  deriving Show

conta Vazia = 0
conta (Nodo _ e d) = 1 + (conta e) + (conta d)

soma Vazia = 0
soma (Nodo x e d) = x + (soma e) + (soma d)

mult Vazia = 1
mult (Nodo x e d) = x * (mult e) * (mult d)

naArv :: (Eq a) => a -> ArvBin a -> Bool
naArv _ Vazia = False
naArv x (Nodo y e d) | x==y      = True
                     | otherwise = (naArv x e) || (naArv x d)
```



```
module Tempo(Time, horas, minutos, meioDia, cauda) where

import Listas

data Time = Am Int Int
          | Pm Int Int
          | Total Int Int   deriving Show

hValida (Total h m) = 0<=h && h<24 && 0<=m && m<60
hValida (Am h m)    = 0<=h && h<12 && 0<=m && m<60
hValida (Pm h m)    = 0<=h && h<12 && 0<=m && m<60

horas (Am h m)    = h
horas (Pm h m)    = h + 12
horas (Total h m) = h

minutos (Am h m)    = m
minutos (Pm h m)    = m
minutos (Total h m) = m

meioDia = (Total 12 00)

ex = cauda “experiencia”
```

```haskell
module Horas(Hora(..), Tempo(manha)) where

data Hora = AM Int Int
          | PM Int Int


class Tempo a where
    manha :: a -> Bool
    tarde :: a -> Bool
    tarde t = not (manha t)


instance Tempo Hora where
    manha (AM _ _) = True
    manha (PM _ _) = False
```



```haskell
module Main where

import Arvores (ArvBin(..), soma, naArv)

import qualified Listas (soma, mult, conta)

import Tempo

import Horas

import Char hiding (toUpper, isDigit)

arv1 = Nodo 5 (Nodo 3 Vazia (Nodo 4 Vazia Vazia))
              (Nodo 2 (Nodo 1 Vazia Vazia) Vazia)

lis1 = [1,2,3,4]

minTotal :: Time -> Int
minTotal t = (horas t)*60 + (minutos t)

testeC = cauda lis1

toUpper :: Num a => ArvBin a -> ArvBin a
toUpper Vazia = Vazia
toUpper (Nodo x e d) = Nodo (x*x) (toUpper e) (toUpper d)

test = map toLower "tesTAnDo"
```


Após carregar o módulo `Main`, analise o comportamento do interpretador.

```
*Main> soma arv1
15
*Main> mult arv1
   Variable not in scope: `mult'
*Main> conta arv1
   Variable not in scope: `conta'
*Main> Listas.soma lis1
10
*Main> mult lis1
   Variable not in scope: `mult'
*Main> Listas.mult lis1
24
```

```
*Main> testeC
[2,3,4]
*Main> hValida meioDia
   Variable not in scope: `hValida'
```

```
*Main> isDigit 'e'
   Variable not in scope: `isDigit'
*Main> isAlpha 'e'
True
*Main> toUpper arv1
Nodo 25 (Nodo 9 Vazia (Nodo 16 Vazia Vazia))
        (Nodo 4 (Nodo 1 Vazia Vazia) Vazia)
*Main> test
"testando"
```

```
*Main> minTotal meioDia
720
*Main> minTotal (Am 9 30)
   Data constructor not in scope: `Am'
*Main> manha (AM 9 30)
True
*Main> tarde (PM 17 15)
   Variable not in scope: `tarde'
```


Compilação de programas Haskell

Para criar programas ***executáveis*** o compilador Haskell precisa de ter definido um módulo **`Main`** com uma função **`main`** que tem que ser de tipo **`IO`** .

A função `main` é o ponto de entrada no programa, pois é ela que é invocada quando o programa compilado é executado.

A compilação de um programa Haskell, usando o *Glasgow Haskell Compiler*, pode ser feita executando na shell do sistema operativo o seguinte comando:

`ghc -o` *nome_do_executável* **`--make`** *nome_do_ficheiro_do_módulo_principal*

Exemplo: Usando o último exemplo para testar a compilação de programas definidos em vários módulos, podemos acrescentar ao módulo `Main` a declaração

```haskell
main = print "OK"
```

Assumindo que este módulo está guardado no ficheiro **`Main.hs`** podemos fazer a compilação assim:

```
ghc -o testar --make Main
```

Exemplo: Assumindo que o módulo do próximo slide está no ficheiro **roots.hs**, podemos gerar um executável (chamado raizes) fazendo

```
ghc -o raizes --make roots
```