UNIVERSIDADE FEDERAL DE SANTA MARIA

Curso: Ciência da Computação

Disciplina: Paradigmas de Programação
Professora: Juliana Kaizer Vizzotto

0.1 Definindo Novos Tipos de Dados

Embora lista e tuplas sejam tipos de dados muito interessantes, frequentemente precisamos construir novos tipos de dados, possibilitando adicionar estrutura aos valores em nossos programas. Definir os próprios tipos de dados facilita a programação e também proporciona maior segurança através da verificação de tipos.

Como exemplo, considere uma pequena livraria online

```
data BookInfo = Book Int String [String]
                deriving (Show)
```

O nome `BookInfo` é o nome do nosso novo tipo de dado (i.e., um *construtor de tipo*). A palavra `Book` é o nome do *construtor de dado ou valor*. Utilizamos para criar um valor do tipo `BookInfo`. Na sequência existem os *componentes* do tipo. Um componente em Haskell tem o mesmo propósito que um campo em uma estrutura

Neste exemplo, o `Int` representa o identificador do livro, a `String` representa o título do livro e a `[String]` representa a lista de autores. Por exemplo:

```
myInfo = Book 0 "The Book of Imaginary Beings" ["Jorge Luis Borges"]
```

Porque utilizamos a palavra `Book` para construir um novo valor do tipo `BookInfo`? Tente:

```
> :type Book
```

Entretanto em Haskell, os nomes de tipos e valores são independentes. Somente utilizamos um construtor de tipo na declaração do tipo ou na assinatura de tipo de funções. Somente utilizamos o construtor do valor no código. Como os usos são distintos, podemos usar o mesmo nome para as duas coisas:

```
data BookReview = BookReview BookInfo CustomerID String
```

0.1.1 Tipos de Dados Algébricos

O tipo muito conhecido `Bool` é o exemplo mais simples de uma categoria de tipos chamada de **tipos de dados algébricos**. Um tipo de dado algébrico pode ter mais de um valor construtor.

```
data Bool = False | True
```

O tipo `Bool` tem dois construtores de valores, `True` e `False`. Cada valor construtor é separado pelo caracter `|`, o qual é lido como *ou*.

Tipos de dados algébricos também servem para *tipos enumerados*, para representar um intervalo de valores simbólicos.

```
enum roygbiv {
    red,
    orange,
    yellow,
    green,
    blue,
    indigo,
    violet,
};
```

Em Haskell, uma definição equivalente seria:

```
data Roygbiv = Red
             | Orange
             | Yellow
             | Green
             | Blue
             | Indigo
             | Violet
               deriving (Eq, Show)
```

Utilizando o novo tipo no ghci:

```
ghci> :type Yellow
Yellow :: Roygbiv
ghci> :type Red
Red :: Roygbiv
ghci> Red == Yellow
False
ghci> Green == Green
True
```

Se um tipo de dado algébrico tem múltiplas alternativas, podemos pensar nele como o tipo *union* em C ou C++. Uma grande diferença entre os dois é que o *union* em C não diz qual alternativa estamos usando realment. Precisamos manualmente e explicitamente cuidar de qual alternativa estamos usando.

```
enum shape_type {
    shape_circle,
    shape_poly,
};

struct circle {
    struct vector centre;
    float radius;
};

struct poly {
    size_t num_vertices;
    struct vector *vertices;
};
```

```
struct shape
{
    enum shape_type type;
    union {
struct circle circle;
struct poly poly;
    } shape;
};
```

Nesse exemplo, o *union* pode contar dados válidos para ou `struct circle` ou `struct poly`. Temos que usar o `enum shapetype` manualmente para indicar qual o valor está armazenado no *union*.

Em Haskell, esse código é dramasticamente mais simples:

```
type Vector = (Double, Double)

data Shape = Circle Vector Double
           | Poly [Vector]
```

Agora que aprendemos como construir valores com tipos de dados algébricos, temos que aprender como trabalhar com esses valores: **patter matching**.

```
bookID :: BookInfo -> Int
bookID      (Book id title authors) = id

bookTitle   (Book id title authors) = title

bookAuthors (Book id title authors) = authors
```

Escrever uma função de acesso para cada um dos componentes do tipo de dado pode deixar o código repetivo. Uma alternativa é utilizar a seguinte notação:

```
data BookInfo = Book {
    bookId  ::  Int,
    bookTitle :: String,
    bookAuthors :: [String]
    } deriving (Show)
```

a qual permite a definição do tipo de dado e de suas funções ao mesmo tempo.

Exercício 1

Define a função:

```
maiorId :: [BookInfo] -> Int
```

a qual retorna a maior identificação em uma lista de livros (use compreensão de listas).