1. O que são classes primitivas e secundárias no Haskell? Cite 3 exemplos de classes primitivas e 3 exemplos de classes secundárias. Cite o papel de cada uma das classes citadas e alguns métodos existentes nelas, bem como o que esses métodos fazem.

Classes primitivas são classes “nativas” da linguagem, ou seja, não são definidas por outras classes. Exemplos:

- Eq: Abreviação de “equal”, trata sobre igualdade e desigualdade entre em todos os tipos, e instancia os metodos == (igual a) e /= (não igual a), responsáveis por, respectivamente, retornar true caso seja igual e false o contrário, e true caso seja diferente e false o contrário.
- Enum: Responsável pela enumerabilidade (ordenamento sequencial) dos tipos. Instancia métodos como “succ”, que retorna o sucessor, e “pred”, que retorna o predecessor.
- Read: Transforma caracteres em numeros, instancia o metodo readList, que recebe uma lista de caracteres e retorna uma de números. Sua classe antagonista é a Show, que transforma números em caracteres, e instancia o metodo show, que recebe uma lista de numeros e retorna uma de caracteres

Classes secundárias são aquelas que foram ou são “criadas” usando as primitivas como base, como é o caso de:

- Ord: classe construida a partir de Eq, usada para ordenação. Instancia os métodos < (menor que) , <= (menor ou igual a) , > (maior que) e >= (maior ou igual a), e os métodos min e max, que, respectivamente retornam o menor e o maior valor entre os recebidos.
- Real, Integral e Fractional: classes numéricas secundárias que instanciam, respectivamente, toRational, toInteger e fromRational, que são metodos de casting de tipos.

2. Faça um breve resumo sobre as classes numéricas do Haskell. Como elas estão relacionadas entre si? Tente entender os métodos disponíveis em cada classe.

Primeiro temos a classe Num, que define os metodos de operação (soma, subtração, multiplicação), e esta intancia os tipos numéricos mais primitivos (Int, Integer, Float, Double).

Depois temos as classes:

- Real: instancia todos os tipos acima (ou seja, instancia Num)
- Integral, Fractional, Floating, RealFrac e RealFloat? instanciam apenas Float

Estas duas instanciam mais metodos numéricos de operações, como por exemplo mod, div, quot, ** (exponencial), log, metodos geométricos (sin, cos, tan e extensoes), alem de métodos de conversão (toInteger, toRational)

3. Pesquise sobre os diversos tipos de polimorfismo existentes: polimorfismo universal e ad-hoc. O polimorfismo universal pode ser polimorfismo por inclusão e polimorfismo paramétrico, enquanto que o polimorfismo ad-hoc pode ser polimorfismo de sobrecarga e polimorfismo de coerção. Pesquise a diferença entre eles e exemplifique seu uso. Faça um breve resumo sobre o assunto

Vale começar por ressaltar que não existem apenas dois tipos de polimorfismo [1], e novos podem ser ainda criados.
O polimorfismo **ad-hoc por sobrecarga** é definido como sendo um polimorfismo entre duas ou mais instancias (neste caso, métodos ou funçoes) que diferenciam pelo número de parâmetros à eles passados. Por exemplo, digamos que queremos uma função que some dois numeros e outra função que soma 3 números.
Poderiamos fazer assim:

```
public int somaDois(int a, int b) {
    return a + b;
}

public int somaTres(int a, int b, int c) {
    return a + b + c;
}
```

Mas isso é uma maneira mais complicada de se fazer, por que poderiamos ter apenas um nome pra função, que somasse os valores passados. Usando polimorfismo por sobrecarga, fariamos assim:

```
public int soma(int a, int b) {
    return a + b;
}

public int soma(int a, int b, int c) {
    return a + b + c;
}
```

E pronto, temos agora duas funções com o mesmo nome, porém diferenciando pelas suas entradas, então chamar soma(1,2) vai ser diferente de soma(1,2,3).

Já o polimorfismo **ad-hoc por coerção** é um polimorfismo que faz uma conversão de tipo, caso seja necessário. Por exemplo, queremos uma função que retorne o peso total de sacos de farinha, ou seja, multiplicando o peso do saco pela quantidade de sacos. Porém, o peso é definido em float e a quantidade em inteiros, logo teriamos que fazer uma conversão explicita antes de fazer a multiplicação. Com o polimorfismo por coerção, não é necessário fazer a conversão, pois a coerção trata disso automaticamente, então ao invés de fazermos

```
public float total(int quant, float peso) {
    float quantFloat = toFloat(quant);
    float total = quantFloat * peso;
    return total;
}
```

podemos fazer a linha de “total” diretamente, e a coerção vai tratar a conversão de tipos

```
public float total(int quant, float peso) {
    float total = peso * quant // quant é casted pra float
    return total;
}
```

Já sobre os polimorfismos Universais, temos **inclusão** e **paramétrico**.
Começando pelo polimorfismo de **inclusão**, podemos citar duas coisas importantes:
1. Está diretamente relacionado com herança
2. É possívelmente o mais conhecido e usado.

Digamos que voce tenha uma superclasse Animal e a subclasse cachorro. Todos os animais respiram, logo todos os cachorros respiram. Porém, apesar de todos os cachorros latirem, nem todos os animais latem.
Com isso, podemos criar a superclasse Animal com o metodo respirar e fazer com que cachorros respirem:

```
public class Animal {
    private String nome;

    public Animal(String nome){
        this.nome = nome;
    }

    public void respirar() {
        System.out.println(this.nome + " está respirando!");
    }
}

public class Cachorro extends Animal {

    public Cachorro(String nome){
        super(nome);
    }

    public void latir(){
        System.out.println(this.nome + " está latindo!");
    }
}
```

E se fizermos as seguintes chamas, teremos os seguintes retornos:

```
Animal animal1 = new Animal("Animal qualquer");
Cachorro cachorro1 = new Cachorro("Rex");

animal1.respirar();
// saida: Animal qualquer está respirando!
cachorro1.latir();
// saída: Rex está latindo!
cachorro1.respirar();
// saída: Rex está respirando!
animal1.latir();
// Erro, pois a classe animal não tem o método "latir"
```

Afinal, sobre polimorfismo **paramétrico**, temos seu detalhe específico no parâmetro da função. Por exemplo, vamos criar função identidade para objetos int e objetos String, poderiamos fazer assim:

```
public int identidade(int x) {
    return x;
}

public String identidade(String s) {
    return s;
}
```

Mas fica claro que temos um problema de repetição de código e falta de abstração, certo? Então podemos criar uma função identidade que não se importe com o que ela recebe, e apenas retorne sua identidade. Ficaria assim:

```
public T identidade(T parametroGenerico) {
    return parametroGenerico;
}
```

Agora T pode ser tanto uma String quanto um int, e podemos chamar a mesma função identidade para ambos.

Felipe de Campos Santos
17200441
25/02/2021

Fontes:
[1] https://medium.com/red-ventures-br-tech/6-tipos-de-polimorfismo-7787080e8857
http://www.inf.ufes.br/~vitorsouza/archive/2020/wp-content/uploads/academia-br-lp-slides07-polimorfismo.pdf
http://albertocn.sytes.net/2012-1/plp/slides/Aula21-LingObjetos_Heranca_Poli_Inclusao.pdf
https://pt.wikipedia.org/wiki/Polimorfismo_param%C3%A9trico