TPFI 2023/24
Hw 1: Programmazione su Liste & Ordine Superiore

docente: I. Salvo – Sapienza Università di Roma

assegnato: 12 marzo 2024, consegna 24 marzo 2024

Nota: Consegnare un unico file testo con estensione `.hs` (preferibilmente di nome `HW1-NomeCognome.hs`). Scrivere la soluzione di eventuali punti 'teorici' (esempio, esercizio **3.4**) semplicemente come commento nel codice Haskell.

1. Rimozione di Duplicati

Nel `Prelude` di Haskell sono predefiniti i funzionali:

takeWhile :: $(a \to \text{Bool}) \to [a] \to [a]$ e *dropWhile* :: $(a \to \text{Bool}) \to [a] \to [a]$

dove *takeWhile p xs* ritorna il segmento iniziale di xs formato dagli elementi che soddisfano p, mentre *dropWhile p xs* rimuove tale segmento iniziale. Esattamente come *take* e *drop* si ha che è sempre soddisfatta l'equazione algebrica: *xs = takeWhile p xs ++ dropWhile p xs*. Ad esempio:

```
Prelude> takeWhile (<3) [1,2,5,7,2,1]
[1,2]
Prelude> dropWhile (<3) [1,2,5,7,2,1]
[5,7,2,1]
```

1. Dare la definizione di *myTakeWhile* e *myDropWhile*;

2. scrivere una funzione ricorsiva *myRemoveDupsOrd* che rimuove i duplicati da una lista *ordinata* xs di lunghezza n in tempo $\mathcal{O}(n)$.

3. scrivere una funzione *myRemoveDups* che rimuove i duplicati da una *qualsiasi* lista xs di lunghezza n in tempo $\mathcal{O}(n \log n)$, preservando l'ordine originale delle prime occorrenze degli elementi rimasti. Ad esempio:

```
Prelude> myRemoveDups [5,2,1,2,5,7,2,1,2,7]
[5,2,1,7]
```

2. Interdefinibilità di Funzionali

1. Definire il funzionale *zipWith f xs ys* senza decomporre liste, ma usando un'espressione che contenga *zapp*, *f* ed eventualmente *xs* e *ys*.

2. Abbiamo visto che *zipWith* è più generale di *zip*. Tuttavia si può definire *zipWith f xs ys* usando *zip* e un paio di altri funzionali visti nella Lezione **3**.

3. Definire il funzionale *map f xs* senza decomporre xs, ma usando un'espressione che contenga *foldr*, *f* e *xs*. Fare lo stesso usando *foldl*.

4. Argomentare brevemente sul perché non sia possibile definire *foldl* e *foldr* usando *map*.

3. Segmenti e sottoliste

1. Scrivere una funzione *prefissi* :: $[a] \to [[a]]$ che ritorna tutti i segmenti iniziali di una lista (vedi funzione *suffissi* nella Lezione **2**).

2. Senza preoccuparsi dell'efficienza, ma usando i funzionali *prefissi*, *suffissi* e altri funzionali dello standard `Prelude`, scrivere una funzione *segSommaS* :: (Num a) $\Rightarrow [a] \to a \to [[a]]$ che data una lista numerica xs e un valore s restituisce tutti i segmenti (cioè sottoliste di elementi *consecutivi*) di xs di somma s.

3. Scrivere una funzione *sublSommaS* :: (Num a) $\Rightarrow [a] \to a \to [[a]]$ che data una lista numerica e un valore s restituisce tutte le sottoliste (anche di *elementi non consecutivi*) di somma s.

4. Partizioni (Facoltativo)

Dato un numero intero positivo n, le *partizioni* di n sono tutti i modi in cui è possibile scrivere n come somma di altri numeri interi positivi.

Ad esempio, le partizioni di 4 sono le sequenze [1,1,1,1], [1,1,2], [1,3], [2,2], [4]. Sono considerate uguali partizioni che differiscono solo per l'ordine, quindi non vanno considerate nelle partizioni di 4 anche [3,1] oppure [1,2,1].

1. Scrivere una funzione Haskell *part* :: Int → Integer, che calcola il numero di partizioni di un certo numero n. Ad esempio, `part 4` calcola 5.

2. Se invece considero diverse tra loro anche partizioni che differiscono solo per l'ordine, quante sono?

3. Scrivere poi una funzione Haskell *parts* :: Int → [[Int]]. che calcola la lista delle partizioni di n. Ad esempio, `parts 4` calcola la lista `[[1,1,1,1], [1,1,2], [1,3], [2,2], [4]]` (potrebbe ovviamente essere diverso l'ordine in cui si scrivono, ma suggerisco di seguire un ordine tanto nella generazione che nel conteggio).

4. (Facile) Ma scrivere `part` usando `parts`? E la complessità è molto maggiore della `part` originaria?