Contents

1 Tokens

1.1 Breking tokens

pensiamo innazitutto a come si potrebbe definire un token, potremmo definirlo come

1.1.1 Massimale?

> famo che un token come una sequenza massimale di caratteri o tutti speciali o tutti normali

in questo caso una possibile implementazione sarebbe

```
tokens = foldr (++) [] -- take the twice nested and "flatten" into a once nested
  . map (groupBy ((==) `on` isSpecial)) -- separate special characters
  . words -- break on whitespace
isSpecial c = elem c "(),`'"
```

questa purtroppo non non funziona in casi quali, ad esempioo

```
tokens "(defun il-cazzo (che me frega) `(walu ,igi)))"
```

si nota infatti, che il risultato di questa viene

```
["(","defun","il-cazzo","(","che","me","frega",")","`(","walu",",","igi",")))"]
```

si noti soprattutto il `"`("` , questo non vale come token, e vorremmo separarlo nei due token `"`` ``, e `"("`

1.1.2 Logica di Break

possiamo definire una qualche logica di break per cui, andando da sinistra a destra col classicone (un mangialiste)

- se stiamo "costruendo" un token speciale (se $\exists$ *spe* $\in$ *speciali t.c. spe* prefisso di *sta cazzo di string*
  allora
  - se il prossimo carattere continua la cosa, continua
  - altrimenti spacca
- se non stamo costruendo un token speciale allora
  - se incontri un carattere speciale allora spaccca
  - se incontri uno spazio allora spacca
- se siamo su uno spazio bianco
  - snobbalo

devo dire non mi sarei mai aspettato di scrivere pseudocodice hakell con delle liste, devo rivedere lisp sti giorni, altrimenti cazzo se divento un riso al curry[1]

1. Un minimo di astrazione ai fini di rompermi meno il cazzo con certe cose, visto che tanto la logica di break e la logica di ignoramus fanno il tokenizer, facciamo prima a specificare le due logiche, e fare un `definisciTokenizer <quella di break> <quella di ignorare>` così non dobbiamo romperci il cazzo a riscrivere tutto il tokenizer ogni volta che ne cambia una.

   ai fini di fare sta cosa, eccovi una funzione *di ordine superiore* idiota

```
-- brk :  break, separa il primo token dal resto della stringa
-- ignr : ignore, ignora eventuali caratteri inutili all'inizio del resto
-- parecchio ad hoc
collectBreakIgnore :: ([a] -> ([a],[a])) -> ([a] -> [a]) -> [a] -> [[a]]
collectBreakIgnore _ _ [] = []
collectBreakIgnore brk ignr lst = let (a,b) = (brk lst)
                                  in a : (collectBreakIgnore brk ignr (ignr b))
```

[1] noto anche come xmonad

si descrive abbastanza da sola ecco un'implementazione di `words` usando questa cosa, per dare un'idea

```
words' = collectBreakIgnore (break isSpace) ignoreWhite
  . ignoreWhite -- aggiunto per gestire leading whitspace
  where ignoreWhite :: String -> String
        ignoreWhite = dropWhile isSpace
```

2. Applicata alla logica di break qui ho fatto troppo il `REPL`-one per documentarla bene, rtfs

2 Tree

`Tree Be Determined`