Monadi

Una delle caratteristiche principali di Haskell è l'essere un linguaggio **lazy**.

Questa caratteristica porta ad alcune conseguenze positive, come:

- Purezza: in un linguaggio completamente puro è presente la **trasparenza referenziale**.

  La trasparenza referenziale si può definire come:

  Se `a = b` posso usare `a` al posto di `b` e viceversa.

  Un possibile esempio di questo concetto può essere:

  ```
  f x = a + a
      where
          a = g x
          b = h 3
  ```

  il quale viene tradotto dal compilatore Haskell in:

  ```
  f x = g x + g x
  ```

  E' quindi possibile osservare come `a` venga sostituita con la sua definizione sottostante (ottimizzazione, **inlining**), andando a rimuovere completamente la definizione di `b` (non utilizzata).

  Questo è possibile solo se il linguaggio è puro, dove non è possibile alterare lo stato esterno.

  Se il linguaggio non fosse stato puro, `g` ed `h` avrebbero potuto causare effetti collaterali con la loro esecuzione. In questo caso, la traduzione effettuata dal compilatore non sarebbe stata corretta.

- Modularità dei programmi: è possibile dividere il programma in moduli differenti, richiamati solo se strettamente necessario.
- Normalizzazione (ad esempio, nel caso della funzione `const 1 undefined`, dove `undefined` non verrà mai valutato perché non utilizzato).
- Parallelismo: nei linguaggi pigri, e di conseguenza puri, è semplice parallelizzare l'esecuzione.

ma anche conseguenze negative, come:

- Purezza: in un linguaggio completamente puro non è possibile stampare a schermo, stabilire connessioni di rete, ecc. Questo rende il linguaggio abbastanza inutile.
- Imprevedibilità: ad esempio, nel caso della funzione `const (print 1) (print 2)`, non si riesce a capire guardando solamente il codice quale sarà l'esito dell'esecuzione.

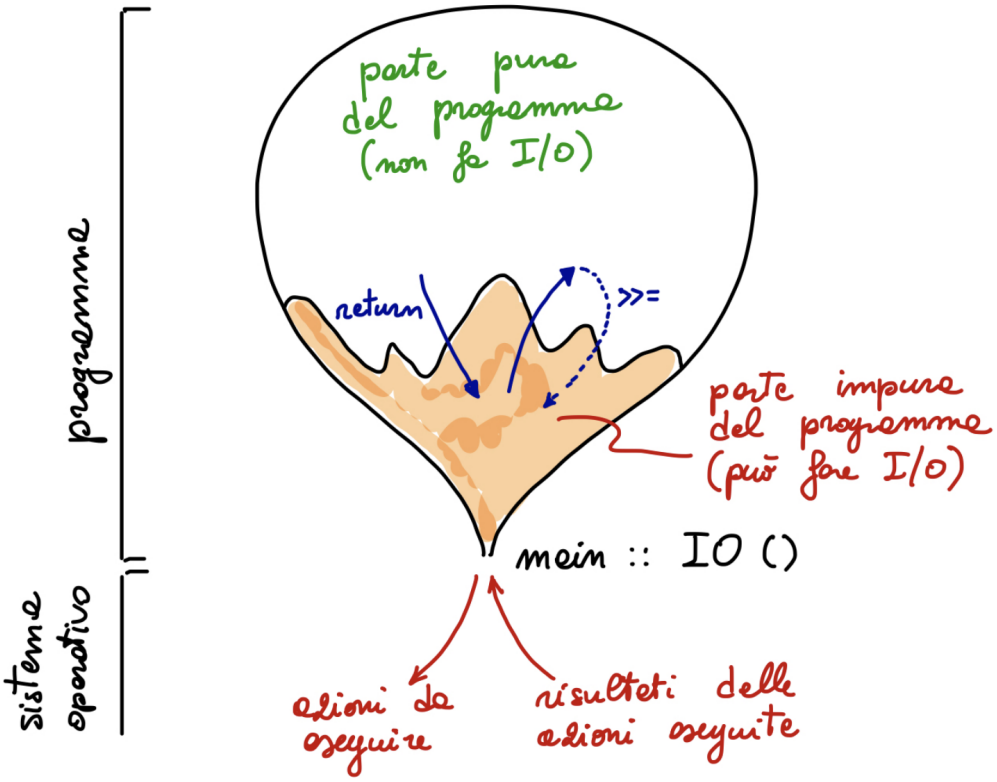
Un possibile esempio di purezza come conseguenza negativa in Haskell è la gestione di Input/Output (dialoghi). Prima delle monadi, questo procedimento richiedeva una logica contorta.

Prima di introdurre il concetto di **monade**, bisogna fare una distinzione tra il fare qualcosa rispetto al pensare di fare qualcosa.

Nella programmazione tradizionale viene scritto in maniera esplicita quello che il programma deve fare, passo dopo passo.

Nella programmazione con monadi, il progrmama **crea delle azioni che se eseguite hanno un effetto collaterale** (come una stampa a schermo).

Una possibile rappresentazione grafica di questo concetto può essere:



E' quindi presente una suddivisione tra la parte *pura* e *impura* del programma. Il nostro programma rimane puro, delegando al sistema operativo l'esecuzione di funzionalità impure.

Inizialmente l'idea delle monadi era stata pensata per gestire il problema dell'input/output. Successivamente ci si è resi conto che poteva essere utilizzata anche per altri scopi (alcuni possono essere eccezioni, parallelismo, transazioni).

Una monade può essere anche definita come: concetto nell'area della semantica dei linguaggi per descrivere matematicamente **computazioni** che possono avere **effetti collaterali**.

E' possibile trovare monadi anche in linguaggi moderni, come Scala e Javascript, fornendo un modo strutturato di organizzare computazioni con effetti collaterali.

Di seguito vengono mostrati esempi pratici di applicazione di monadi.

Valutatore di espressioni

Si vuole realizzare un valutatore per semplici espressioni aritmetiche.

```
data Expr = Const Int | Div Expr Expr

eval :: Expr -> Int
eval (Const n) = n
eval (Div t s) = eval t `div` eval s
```

`Expr` indica una struttura dati custom, la quale può avere un valore `Int` oppure rappresentare una espressione aritmetica.

Vengono successivamente definiti due casi possibili: restituzione del valore inviato e calcolo dell'espressione.

Si vuole ora raffinare il valutatore in modo che supporti:

- Gestione delle **divisioni per zero**
- **Conteggio** delle operazioni "difficili"
- **Tracciamento** dei passi di valutazione

Se usassimo un linguaggio impuro sarebbe tutto facile.

Partiamo dalla divisione per zero.
Per implementare la gestione della divisione per zero viene modificato il codice nel seguente modo:

```
eval :: Expr -> Maybe Int
eval (Const n) = Just n
eval (Div t s) =
    case eval t of
        Nothing -> Nothing
        Just m -> case eval s of
            Nothing -> Nothing
            Just 0 -> Nothing
            Just n -> Just (m `div` n)
```

Ciò che era facile fare in un linguaggio impuro ha stravolto la struttura del programma in un linguaggio puro. La logica del programma si perde nella gestione dell'eccezione.

Si passa ora al conteggio delle operazioni "difficili".
Per implementare il conteggio delle operazioni viene modificato il codice nel seguente modo:

```
type Counter a = Int -> (a, Int)

eval :: Expr -> Counter Int
eval (Const n) x = (n, x)
eval (Div t s) x =
    let (m, y) = eval t x in
        let (n, z) = eval s y in
            (m `div` n, z + 1)
```

Ciò che era facile fare in un linguaggio impuro ha stravolto la struttura del programma in un linguaggio puro. La logica del programma si perde nella gestione del contatore.

Per concluduere, si passa al tracciamento dei passi di valutazione.
Per implementare il tracciamento dei passi di valutazione viene modificato il codice nel seguente modo:

```
type Output a = (String, a)

eval :: Expr -> Output Int
eval (Const n) = (line (Const n) n, n)
eval (Div t s) =
    let (x, m) = eval t in
        let (y, n) = eval s in
            (x ++ y ++ line (Div t s) (m `div` n), m `div` n)

line :: Expr -> Int -> String
line t n = "eval (" ++ show t ++ ") = " ++ show n ++ "\n"
```

Ciò che era facile fare in un linguaggio impuro ha stravolto la struttura del programma in un linguaggio puro. La logica del programma si perde nella gestione della traccia.

In tutti e tre i casi, l'utilizzo di un linguaggio puro porta ad un codice più complesso.

CAPIRE COSA METTERE, SIAMO A SLIDE 20 DELLE MONADI. LUI HA PARLATO UN PO DELL'ESEMPIO DELLA DIVISIONE PER ZERO CON MONADI, MA NON SI CAPISCE BENE A COSA FACCIA RIFERIMENTO (NO CONDIVISIONE SCHERMO).