# Progetto di Programmazione mod. 1
## F# Code Formatter

v2.2.1

December 14, 2017

## Indice


**1 Introduzione** **2**

**2 Indicazioni generali** **2**

**3 Elementi di valutazione** **3**

**4 Specifiche base** **3**
- 4.1 Sintassi . . . 3
- 4.2 Formato dell'input di `indent` . . . 5
  - 4.2.1 `let` a top level e locali . . . 6
  - 4.2.2 `if` ed `elif` . . . 7
  - 4.2.3 `else` . . . 7
  - 4.2.4 `match` e pattern . . . 8
  - 4.2.5 lambda astrazioni . . . 9
  - 4.2.6 let locali con `in` . . . 9
  - 4.2.7 espressioni ignote . . . 10

**5 Esempi di input ed output base** **11**

**6 Versione avanzata (OPZIONALE)** **17**
- 6.1 Come spezzare le righe . . . 17
  - 6.1.1 `if` & `elif` . . . 17
  - 6.1.2 `else` . . . 17
  - 6.1.3 `match` e pattern . . . 18
  - 6.1.4 `let` . . . 18
  - 6.1.5 `fun` . . . 18
  - 6.1.6 `in` ed espressioni . . . 19

**7 Esempi di input output avanzati** **19**

# 1 Introduzione

Il progetto consiste nell'implementare un programma che, dato in input un programma F# indentato in modo scorretto, produce in output il medesimo programma F# indentato correttamente. Ad esempio, un input di 7 linee come:

```
1 let f x =
2 let a = x + 1
3 let b = 3
4 if a < b then a
5 else
6 let c = a + b
7 in c + 1
```

Deve produrre in output:

```
1 let f x =
2     let a = x + 1
3     let b = 3
4     if a < b then a
5     else
6         let c = a + b
7         in c + 1
```

Cioè il *medesimo* programma, costituito dalle *medesime 7 linee*, ma indentate opportunamente.

Il programma di input è garantito essere un programma F# che utilizza un piccolo sottoinsieme dei costrutti del linguaggio: F# è un linguaggio molto ricco e naturalmente non è richiesto supportare tutta la sua sintassi. Le specifiche esatte dei costrutti da supportare è discussa nel dettaglio in sezione 4.

# 2 Indicazioni generali

- Si accettano gruppi di massimo 2 persone.
- **Non** è possibile utilizzare **librerie** all'interno del progetto. Tutte le funzioni che utilizzate nella vostra soluzione devono essere **implementate da voi** ad esclusione delle funzioni che vi vengono fornite nel file **Lib.fs**.
- **Non** è possibile utilizzare **cicli** e altri costrutti imperativi di F# come ad esempio *let mutable*, *ref* ecc.
- Leggere con attenzione le istruzioni di consegna che trovate nella home del corso, progetti **consegnati in modo errato** verrano considerati **insufficienti**.

- Progetti **non compilanti** verranno considerati **insufficienti**.
- **Non** è consentito cambiare la struttura del progetto, è obbligatorio che il progetto compili nella sua interezza, potete modificare i file **Main.fs** e **Test.fs** a vostra discrezione al fine di testare il vostro programma.
- Il file **Lib.fs** contiene delle funzioni che possono aiutarvi nella realizzazione del vostro progetto, il loro utilizzo è **opzionale** anche se **consigliato**.

# 3 Elementi di valutazione

Sebbene il progetto sia un lavoro che può essere svolto in gruppo, la **valutazione** sarà **individuale**. Oltre alla discussione orale, saranno valutate:

- qualità del codice: oltre alla correttezza, saranno considerate l'eleganza della soluzione implementata, l'utilizzo di indentazione corretta e consistente, la suddivisione in sottofunzioni, etc;
- qualità della **documentazione**: codice **non commentato** non sarà valutato;

# 4 Specifiche base

Vi viene richiesto di implementare la seguente funzione:

```
indent : lines:string list -> (int * string) list
```

che data in input una lista che rappresenta le linee di un programma F# indentato in modo errato, restituisce la rappresentazione del programma indentato correttamente. L'output di questa funzione è una lista di coppie `int * string` in cui l'intero è il valore di tabulazione di ogni linea e la stringa è la linea stessa priva di caratteri di spazio o indentazione.

## 4.1 Sintassi

Un sorgente dato in input può essere rappresentato dalla seguente grammatica formale in notazione EBNF[1].

Una grammatica essenzialmente descrive le regole *sintattiche* di un linguaggio: nel nostro caso si tratta di un sottoinsieme del linguaggio F#, che il vostro programma deve essere in grado di riconoscere ed indentare.

- I simboli sottolineati sono parole chiave del linguaggio, inclusi i segni di punteggiatura. Per esempio la sintassi della tupla è un parentesi

[1] Per approfondire le grammatiche formali si consulti [2] e per la notazione EBNF [1].

Table 1: Sintassi dei programmi e delle espressioni F#..

| | | | |
|---|---|---|---|
| $P$ | $\rightarrow$ | $(Let\ (\hookleftarrow)+)+$ | **programma** |
| $Let$ | $\rightarrow$ | `let` [`rec`] $x_1$ .. $x_n$ `=` $E$ | **binding** |
| $E$ | $\rightarrow$ | | **espressioni** |
| | | $c$ | costante |
| | $\mid$ | `fun` $x_1$ .. $x_n$ `->` $E$ | lambda astrazione |
| | $\mid$ | $E\ E$ | applicazione |
| | $\mid$ | $Let$ `in` $E$ | binding locale |
| | $\mid$ | (`if` $\mid$ `elif`) $E$ `then` $E$ `else` $E$ | condizionale |
| | $\mid$ | `match` $E$ `with` (`\|` $P$ `->` $E$)+ | pattern matching |
| | $\mid$ | `(`$E$ (`,` $E$) + `)` | tupla |
| | $\mid$ | $E\ op\ E$ | operatore binario |

dove $x$ sono identificatori, $n \geqslant 1$, $c \in \mathbb{R} \cup \mathbb{S} \cup \{$ `true`, `false` $\}$ e $op \in \{$ `+`, `-`, `/`, `*` $\}$.

aperta `(` seguita da una espressione $E$ e seguita a sua volta da una serie di uno o più[2] virgole `,` seguite da un'espressione $E$. Ad esempio `(1, f x, let x = 1 in x + x)` è una tupla, per quanto possa sembrare strana. Lo è perché ognuna delle 3 sotto-espressioni sono valide: `1` è una costante riconducibile al caso `c`, `f x` è una applicazione e `let x = 1 in x + x` è un binding locale, a sua volta costituito da un binding `let x = 1` seguito da `in` e da una espressione $E$.

- Le costanti del caso $c$ possono essere numeri `int` o `float`, oppure stringhe ($\mathbb{S}$ è l'insieme delle stringhe alfanumeriche), oppure un valore booleano.
- Sia i let che le lambda supportano lo zucchero sintattico con identificatori multipli, come F#.
- I pattern di un `match` devono avere un simbolo di pipe `|` prima di ogni pattern, il primo incluso. Si noti che in F# il primo pipe è opzionale, mentre nel linguaggio che il vostro programma deve riconoscere il pipe c'è sempre, anche prima del primo pattern.
- La parola chiave `rec` è opzionale in un binding (la notazione $[A]$ indica una o nessuna occorrenza di un termine A).

[2]La notazione $(A)+$ indica una o più occorrenze di un termine $A$.

- `if` ed `elif` sono parole chiave interscambiabili dal punto di vista puramente sintattico - non si confonda la loro semantica (ovvero il significato che assumono in un programma) con le regole grammaticali o la sintassi.

A top level, un programma P è una serie di uno o più costrutti di binding separati da una linea vuota. Il carattere di end-of-line è ↵ nella nostra specifica (mentre in F# è la stringa `"\n"`) e la notazione (↵)+ indica appunto uno o più caratteri di end-of-line. Di conseguenza il seguente è un programma valido secondo la grammatica di cui sopra[3]:

```
let f x = x + 1

let g x = fun y z -> x * y * z


let h a b c = (a, b, c)
```

## 4.2 Formato dell'input di `indent`

La sintassi appena discussa descrive le regole grammaticali da rispettare affinché un programma sia formato correttamente, ma non è tutto. Ci sono delle regole di indentazione che il sorgente passato in input alla funzione `indent` deve rispettare. In altre parole questo significa che non tutti gli input che rispettano la grammatica in figura 1 sono davvero validi per la `indent`, ma solo gli input che soddisfano un certo numero di regole di formattazione lo sono. In particolare, la funzione `indent` prende come parametro una lista di stringhe: quindi si tratta di un input già suddiviso in linee, il che indica sono necessari alcuni caratteri di a-capo in alcuni punti specifici. Una funzione lunga **deve avere già degli a-capo** quando viene passata alla `indent` ed ogni funzione a top-level deve essere separata da almeno una linea vuota. Ad esempio il seguente input è valido per la funzione `indent`:

```
let f x =
let a = x + 1
let b = x * 2
in a + b

let g x y =
let r = x * y
in r * 2
```

Se `indent` funziona correttamente deve riconoscere che si tratta di 2 funzioni distinte e che i let dopo un a-capo sono let a top level mentre gli altri sono let locali, producendo un output che ha lo stesso numero di linee ma è indentato correttamente.

[3] Si noti che la doppia linea vuota è valida.

Seguono ora, caso per caso, le regole di indentazione per ciascun costrutto F# che il vostro programma deve supportare: se tali regole sono rispettate, allora la funzione `indent` può fare il suo lavoro di tabulazione delle linee e produrre un programma di output indentato correttamente.

### 4.2.1 `let` a top level e locali

Il costrutto `let` ha 2 variant: a top level o locale. In entrambi i casi la parola chiave `let` deve comparire sempre all'inizio di una riga, se dopo l'uguale c'è solo un'espressione essa può - non obbligatoriamente - essere scritta nella stessa linea.

```
let a = 3 + sqrt x  // questo let e' tutto in un linea
let b = 4           // anche questo
let c =             // questo va a capo ma e' valido
    in 3 + a + b
```

Se dopo l'uguale = sono presenti altri costrutti allora è obbligatorio andare a capo. Assumiamo il seguente input:

```
    let f x =
                    let a = x + 1
in x + a

            let g y =
        if y > 3 then 2
else 4
```

Si noti che la tabulazione di ogni linea può essere sbagliata: lo scopo della funzione `indent` è esattamente quello di prendere un sorgente già suddiviso in linee e ricalcolare correttamente la tabulazione di tali linee. Pertanto la tabulazione delle linee dell'input deve essere ignorata e ricalcolata sulla base dei costrutti presenti.

Il let della prima linea è per forza un let a top level perché è ad inizio input, così come il `let g` perché è preceduto da un linea vuota. Il let alla linea 2 invece è un let locale, semplicemente in virtù del fatto che non rispetta le regole del let a top level. L'espressione `x + a` sottointende un `in` - si veda la sezione 4.2.6 - quindi è di fatto il corpo della funzione `f`. Il costrutto `if` è una sottoespressione della funzione `g` e richiede di rispettare ulteriori regole. Quindi l'indentazione corretta è:

```
let f x =
    let a = x + 1
    in x + a

let g y =
    if y > 3 then 2
    else 4
```

Da questo punto in poi ci sofferemeremo su degli esempi che mostreranno i comportamenti dell'indentazione per i casi casi di sotto-espressioni, senza mostrare la funzione a top level che li contiene.

### 4.2.2 `if` ed `elif`

I costrutti `if` (o `elif`, che sono sintatticamente identici) devono comparire sempre nella stessa linea di codice del corrispettivo `then`, se dopo il `then` c'è solo un'espressione essa può essere scritta nella stessa linea del `then`.

```
if x > 3 then 3 + sqrt(x)
elif x > 5 then 2 + x * x - 6
else 4 - x

if x > 3 then
3 + sqrt(x)
else 4 - x
```

Se dopo il `then` sono presenti altri costrutti allora è obbligatorio andare a capo dopo il `then`.

```
if x > 3 then 3 + sqrt(x)
elif x > 5 then
let p = x * x
let d = 2 * x / (x + 1)
in p + d
else 4 - x
```

### 4.2.3 `else`

Il costrutto `else` si deve trovare sempre all'inizio di una linea. Se dopo l'`else` sono presenti altri costrutti allora è obbligatorio andare a capo subito dopo di esso,

```
if x > 3 then 3 + sqrt(x)
else
let a 0 3
let b = x * 3
in a + b
```

Se dopo l'else è presente solo un'espressione essa può essere contenuta nella stessa riga.

```
if x > 3 then 3 + sqrt(x)
else 4 - x

if x > 3 then 3 + sqrt(x)
else
4 - x
```

### 4.2.4 `match` e pattern

Il costrutto `match` deve **sempre** andare a capo dopo il `with`.

```
match l with
| _ -> []
```

Tutti i pattern inziano con il carattere di pipe | e deve essere il primo carattere della riga. Dopo la freccia del pattern se è presente solo un'espressione essa può essere contenuta nella stessa riga.

```
match l with
| [] -> []
| x :: xs -> (2 * x) :: xs
```

Se sono presenti altri costrutti allora è obbligatorio andare a capo dopo la freccia `->`.

```
match l with
| [] ->
    []
| x :: xs ->
if x = 0 then 3 :: xs
else
l
```

Se uno dei pattern contiene un altro match, tutti i pattern successivi apparterranno all'ultimo *match* ed andranno indentati di conseguenza.
Input:

```
let f x =
match a with
             | 1 ->
    match b with
             | 2 ->
    2
| 3 ->
    match c with
| 4 -> 4
    | 5 -> 5
```

Output:

```
let f x =
    match a with
    | 1 ->
        match b with
        | 2 ->
            2
        | 3 ->
            match c with
            | 4 -> 4
            | 5 -> 5
```

### 4.2.5 lambda astrazioni

I costrutti `fun` sono sempre all'inizio di una riga, se dopo la freccia `->` c'è solo un'espressione essa può essere scritta nella stessa linea.

```
fun x -> 3 + sqrt(x)
```

```
fun x ->
    x + 1
```

Se dopo la freccia `->` sono presenti altri costrutti allora è obbligatorio andare a capo subito dopo la freccia `->`.

```
            fun y ->
        if y > 3 then
2
else 4
```

### 4.2.6 let locali con `in`

E' importante chiarire che `in` **non è un costrutto** ma è solamente la parola chiave che separa la parte del binding dalla parte del corpo di un `let` locale, come appare chiaro dalla grammatica in figura 1. Sebbene in F# sia possibile omettere la keyword `in` in luogo dell'indentazione del codice, nel nostro linguaggio di input la keyword `in` è **obbligatoria**. Ad esempio, in F# reale è possibile scrivere:

```
let f x =
    let a = 3
    a + 1
```

In cui la keyword `in` è appunto omessa, ma il codice è esattamente identico a:

```
let f x =
    let a = 3
    in a + 1
```

Nel nostro linguaggio di input, che ricordiamo essere una semplificazione dei F#, ci sarà sempre la keyword `in` **almeno dopo l'ultimo** `let`. Per esempio questo è un input valido:

```
let f x =
    let a = 3
    let b = 4
    let c = 5
    in a + b + c
```

Come si può vedere i primi 2 `let` interni non hanno `in`, mentre il terzo ce l'ha. In generale, nei linguaggi funzionali ispirati ad ML (come F#, OCaml, SML ed altri), ogni `let` ha il suo `in`, ma nella nostra semplificazione di F# per questo progetto solo l'ultimo `in` di una serie di `let` è obbligatorio.

Il motivo per cui almeno l'ultimo `in` è obbligatorio è per permettere di scrivere il `let` anche tutto in un riga, rendendo `in` è necessario per separare il binding dal corpo:

```
let f x = let a = 3 in a + 1
```

Altrimenti, se fosse possibile omettere l'`in`, allora sarebbe possibile scrivere questo:

```
let f x = let a = 3 a + 1
```

Il che verrebbe confuso dal compilatore F# con l'*applicazione* `3 a`, il risultato della quale viene sommato alla costante `1`. Naturalmente l'applicazione `3 a` non ha senso dal punto di vista dei tipi, ma sintatticamente parlando è una applicazione valida: l'espressione di sinistra (`3`) è applicata a quella di destra (`a`). Per questo motivo la parola chiave `in` è necessaria.

### 4.2.7 espressioni ignote

Alcune regole grammaticali come l'applicazione, la tupla e le costanti non hanno una specifica keyword che le contraddistingue, pertanto *non* rappresentano dei casi di indentazione da gestire. Nella pratica è conveniente gestire tutti i costrutti che non hanno una keyword iniziale com caso rimanente dopo aver gestito tutti gli altri casi precedentemente.

Si badi inoltre che quando compaiono costrutti noti all'interno di applicazioni o tuple, l'indentazione non è possibile. Ad esempio:

```
let f x =
    let g x = x * 2
    let h x = x * 3
    in (if x < 0 then g else h) (x + 1)
```

L'espressione `(if x < 0 then g else h)` è il lato sinistro di una applicazione, cioè la funzione, mentre `(x + 1)` è l'espressione sul lato destro, cioè l'argomento. In tal caso è impossibile indentare correttamente, perché occorrerebbe implementare un vero e proprio algoritmo di *parsing* [3], che esula dagli obiettivi di questo progetto.

Espressioni simili vanno *lasciate come sono*.

# 5 Esempi di input ed output base

In questo capitolo vi vengono mostrati alcuni esempi di input ed output per il vostro programma.

Input:

```
let f x =
if x > 0 then 1
else
let y = x + 2
                            let z = x + 3
    let f x =
        if x > 0 then
let a = 1
             let b = 3
             in a + b
                 elif x > 0 then 3
elif x > 0 then
                                 let a = 1
             let b = 3
in a + b
else 3
in f y + f z

let fib n =
             let rec R n =
if n < 2 then 1
else R (n - 1) + R (n - 2)
in R n
```

Output:

```
let f x =
    if x > 0 then 1
    else
        let y = x + 2
        let z = x + 3
        let f x =
            if x > 0 then
                let a = 1
                let b = 3
                in a + b
            elif x > 0 then 3
            elif x > 0 then
                let a = 1
                let b = 3
                in a + b
            else 3
        in f y + f z

let fib n =
    let rec R n =
        if n < 2 then 1
        else R (n - 1) + R (n - 2)
    in R n
```

---

Input:

```
let f x a b c =
match a with
              | 1 ->
    match b with
              | 2 ->
    if x > 18 then 3
              else 4
| 3 ->
    match c with
| 4 -> 4
    | 5 -> 5
```

Output:

```
let f x a b c =
    match a with
    | 1 ->
        match b with
        | 2 ->
            if x > 18 then 3
            else 4
        | 3 ->
            match c with
            | 4 -> 4
            | 5 -> 5
```

---

Input:

```
let rec F y =
let f x = x + 1
let y = f 4
match f 3 with
| 2 ->
match y with
    | 3 -> true
| 4 ->
if p 3 then
let z = 8
in z < y
    else false
```

Output:

```
let rec F y =
    let f x = x + 1
    let y = f 4
    match f 3 with
    | 2 ->
        match y with
        | 3 -> true
        | 4 ->
            if p 3 then
                let z = 8
                in z < y
            else false
```

---

Input:

```
let f x =
let res =
match x with
| 0 -> 1
| 1 -> 18
| 2 ->
match x + 45 with
| 47 -> x * 2
| _ -> 7
in res + 8
```

Output:

```
let f x =
    let res =
        match x with
        | 0 -> 1
        | 1 -> 18
        | 2 ->
            match x + 45 with
            | 47 -> x * 2
            | _ -> 7
    in res + 8
```

---

Input:

```
let f x =
let res =
if x < 5 then
let a = 8
let b =
if x < 3 then 8
else 3
x + a * b
else
let c =
match x with
| 6 -> 7
| _ -> 78
let d = 1
in x * c + d
in res
```

Output:

```
let f x =
    let res =
        if x < 5 then
            let a = 8
            let b =
                if x < 3 then 8
                else 3
            in x + a * b
        else
            let c =
                match x with
                | 6 -> 7
                | _ -> 78
            let d = 1
            in x * c + d
    in res
```

# 6 Versione avanzata (OPZIONALE)

Vi viene chiesto di modificare la seguente funzione:

```
split : w:int -> s:string -> string list
```

La funzione deve permettere di spezzare le righe di codice che superano la lunghezza $w$ restando **compatibile con la indent implementata**. Le righe prodotte dalla split **devono rispettare** i vincoli sugli input specificati nella sezione 4.2. Non è detto che tutte le righe che superano la lughezza $w$ possano essere spezzate.

## 6.1 Come spezzare le righe

La `split` oltre ai vincoli definiti nella sezione 4.2, deve produrre linee che rispettano le seguenti regole.

### 6.1.1 `if` & `elif`

I costrutti `if` ed `elif` forniti in **input** non possono essere spezzati prima del loro corrispettivo `then`, se subito dopo il `then` è presente un altro costrutto è **obbligatorio** andare a capo dopo il `then`.

Nel seguente esempio potete notare che il costrutto `if` supera la lunghezza $w$ fornita (`if x > 0 then 3` é lunga 15 caratteri) ma invece di andare a capo dopo il `then` la split va a capo dopo l'espressione relativa al `then`.

Input con `w = 10`:

```
let f x = if x > 0 then 3 else 2
```

Output con `w = 10`:

```
let f x =
    if x > 0 then 3
    else 2
```

### 6.1.2 `else`

Se il costrutto `else` fornito in **input** é seguito da un costrutto, esso deve **obbligatoriamente** andare a capo.
Input con `w = 10`:

```
let funzione x = if x > 0 then
3
else f 2
```

Output con `w = 10`:

```
let funzione x =
    if x > 0 then 3
    else f 2
```

### 6.1.3 `match` e pattern

Il costrutto `match` deve **obbligatoriamente** andare a capo dopo il corrispondente `with`.
Input con `w = 10`:

```
match nome_molto_lungo with | _ -> []
```

Output con `w = 10`:

```
match nome_molto_lungo with
| _ -> []
```

Un pattern **non** può andare a capo prima della sua freccia. Se sulla stessa riga di una `|` è presente un costrutto essa deve **obbligatoriamente** andare a capo dopo la freccia `->`.
Input con `w = 15`:

```
match l with | [] -> [] | x :: xs -> (2 * x) :: xs
```

Output con `w = 15`:

```
match l with
| [] -> []
| x :: xs -> (2 * x) :: xs
```

### 6.1.4 `let`

I costrutti `let` **non** possono andare a capo prima del corrispondente carattere `=`. Se il costrutto `let` fornito in **input** é seguito da un costrutto, la riga deve **obbligatoriamente** andare a capo dopo il carattere `=`.

---

Input con `w = 10`:

```
let nome_molto_lungo = 3 + funzione_molto_lunga(x)
```

Output con `w = 10`:

```
let nome_molto_lungo =
    3 + funzione_molto_lunga(x)
```

### 6.1.5 `fun`

I costrutti `fun` non possono andare a capo prima della corrispondente freccia `->`. Se il costrutto `fun` fornito in **input** é seguito da un costrutto, la riga deve **obbligatoriamente** andare a capo dopo la freccia `->`.
Input con `w = 15`:

```
fun x -> if x then 1 else 0
```

Output con `w = 15`:

```
fun x ->
    if x then 1
    else 0
```

---

Input con `w = 10`:

```
             fun nome_molto_lungo ->
3 + sqrt(nome_molto_lungo)
```

Output con `w = 10`:

```
fun nome_molto_lungo ->
    3 + sqrt(nome_molto_lungo)
```

### 6.1.6 `in` ed espressioni

Se una espressione ha una lunghezza pari o inferiore a $w$ dato in input, essa deve **obbligatoriamente** stare nella stessa riga del comando che la precede, questo può richiedere di dover unire due righe distinte del codice in input.

```
fun nome_molto_lungo ->
3 + sqrt(4)
```

Output con `w = 15`:

```
fun nome_molto_lungo -> 3 + sqrt(4)
```

---

Input con `w = 15`:

```
let nome_molto_lungo =
3 + sqrt(x)
```

Output con `w = 15`:

```
let nome_molto_lungo = 3 + sqrt(x)
```

## 7 Esempi di input output avanzati

Input con `w = 15`:

```
1 let funzione x = if x > 0 then 3 else funzione 2
```

Output con `w = 15`:

```
1 let funzione x =
2     if x > 0 then 3
3     else funzione 2
```

---

Input con `w = 10`:

```
let rec foldl f z l = match l with | [] -> if x > 0 then
    1 else 3 | x :: xs -> foldl f (f z x) xs
```

Output:

```
let rec foldl f z l =
    match l with
    | [] ->
        if x > 0 then 1
        else 3
    | x :: xs ->
        foldl f (f z x) xs
```

---

Input con `w = 10`:

```
let rec map f l = let k = 1 let rec R c = match c with |
    [] -> [] | x :: xs -> match f x with | [] -> 3 | x ::
     xs -> match a with | [] -> [] | x :: xs -> 3 in R []
```

Output con `w = 10`:

```
let rec map f l =
    let k = 1
    let rec R c =
        match c with
        | [] -> []
        | x :: xs ->
            match f x with
            | [] -> 3
            | x :: xs ->
                match a with
                | [] -> []
                | x :: xs -> 3
    in R []
```

## References

[1] Extended backus–naur form. [Apri link].

[2] Formal grammar. [Apri link].

[3] Parsing. [Apri link].