Haskell

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldr :: (a -> b -> b) -> b -> [a] -> b

funName pattern1 | guardia11 = exp11
                 | ...
                 | guardia1n = exp1n
funName pattern2 | guardia21 = exp21
                 | ...
                 | guardia2m = exp2m
funName pattern3 ...
                 ...

case exp of
  pattern1 | guardia11 -> exp11
           | ...
           | guardia1n -> exp1n
  pattern2 | guardia21 -> exp21
           | ..
           | guardia2m -> exp2m
  pattern3 ...
           ...
```

Types

```
M ::= x       (* variabili *)
    | c       (* costanti *)
    | λx:A. M (* astrazione *)
    | M N     (* applicazione *)

A ::= Int | Bool   (* tipi base *)
    | A → B        (* funzione *)

(Var)
G, x:A, G' ⊢ x:A

(Fun)
   G, x:A ⊢ M:B
--------------------
G ⊢ (λx:A. M) : A→B

(App)
G ⊢ M : A→B      G ⊢ N:A
------------------------
      G ⊢ M N : B
```

```
(unit)
G ⊢ unit : Unit

G ⊢ true : Bool
G ⊢ false : Bool

(if-then-else)
G ⊢ M : Bool    G ⊢ N1 : A G ⊢ N2 : A
-------------------------------------
   G ⊢ if_A M then N1 else N2 : A
```

```
Naturali:
G ⊢ 0 : Nat
G ⊢ succ : Nat → Nat
G ⊢ pred : Nat → Nat
G ⊢ isZero : Nat → Bool

(+)
G ⊢ N1 : Nat  G ⊢ N2 : Nat
--------------------------
    G ⊢ N1 + N2 : Nat
```

```
(Pair)
G ⊢ M : A     G ⊢ N : B
-----------------------
    G ⊢ (M,N) : A*B

(First)
 G ⊢ M : A * B
----------------
G ⊢ first M : A

(Second)
 G ⊢ M : A * B
----------------
G ⊢ second M : B

(Case)
G ⊢ M : A1 * A2     G, x1:A1, x2:A2 ⊢ N:B
-----------------------------------------
   G ⊢ case M of (x1:A1,x2:A2) → N : B
```

```
Unione:
(InLeft)
     G ⊢ M : A
--------------------
G ⊢ inLeft_B M : A+B

(InRight)
     G ⊢ M : B
---------------------
G ⊢ inRight_A M : A+B

(AsLeft)
   G ⊢ M : A+B
----------------
G ⊢ asLeft M : A

(AsRight)
   G ⊢ M : A+B
-----------------
G ⊢ asRight M : B

(IsLeft)
    G ⊢ M : A+B
-------------------
G ⊢ isLeft M : Bool

(IsRight)
    G ⊢ M : A+B
--------------------
G ⊢ isRight M : Bool
```

```
Record
(Record)
G ⊢ M1 : A1     G ⊢ M2:A2     ...     G ⊢ Mn:An
-------------------------------------------------
G ⊢ { l1:M1, ..., ln:Mn } : { l1:A1, ..., ln:An }

(Record Select)
G ⊢ M : { l1 : A1, ... , ln : An }
----------------------------------
         G ⊢ M.li : Ai
```

```
Reference
(ref)
G ⊢ M : A   A memorizabile
--------------------------
    G ⊢ ref M : Ref A

(deref)
G ⊢ deref : (Ref A) → A

(Assign)
G ⊢ M : Ref A     G ⊢ N : A
---------------------------
     G ⊢ M := N : Unit
```

Linguaggio imperativo dentro uno funzionale

```
var x = M;
N

diventa (λx : (Ref A) . N) (ref M)


La composizione C1; C2 diventa (λy : Unit . C2) C1


(Composition)
G ⊢ C1 : Unit    G ⊢ C2 : Unit
------------------------------
     G ⊢ C1; C2 : Unit
(C1;C2) ::= (λy : Unit . C2) C1
```

Linguaggio imperativo, frammento di C

```
Espressioni
E ::= const | id | E binop E | unop E

Comandi
C ::= id = E
    | C; C
    | while E {C}
    | if E then C else C
    | I(E, ..., E)
    | {D ; C}

Dichiarazioni
D ::= A id = E
    | id(A1 id1, ... , An idn) { C }
    | epsilon
    | D; D
creano un ambiente: G ⊢ D :: G1


(Id, (Var))
G, id:A, G' ⊢ id : A
G ⊢ true : Bool         G ⊢ false : Bool

(ite)
G ⊢ (if _ then _ else _) : Bool → A → A → A
G ⊢ 1 : Nat     G ⊢ 2 : Nat     G ⊢ 3 : Nat ...

operazioni aritmetiche
G ⊢ E1 : Nat      G ⊢ E2 : Nat
------------------------------
     G ⊢ E1 + E2 : Nat

(Assign)
G ⊢ id : A     G ⊢ E : A
------------------------
   G ⊢ id = E : Unit

(Sequence)
G ⊢ C1 : Unit  G ⊢ C2 : Unit
----------------------------
     G ⊢ C1; C2 : Unit

(While)
G ⊢ E : Bool     G ⊢ C : Unit
----------------------------
   G ⊢ while E {C} : Unit

(If Then Else)
G ⊢ E : Bool    G ⊢ C1 : Unit   G ⊢ C2 : Unit
---------------------------------------------
     G ⊢ if E then C1 else C2 : Unit

(Procedure)
G ⊢ id:(A1 * ... * An)→Unit   G ⊢ E1 : A1 ... G ⊢ En : An
---------------------------------------------------------
             G ⊢ id (E1, ..., En) : Unit

(Blocco)
G ⊢ D :: G1     G, G1 ⊢ C : Unit
--------------------------------
       G ⊢ {D;C} : Unit

(Id)
G ⊢ E : A      (A tipo memorizzabile)
-------------------------------------
     G ⊢ A id = E :: (id : A)

(Proc)
       G, id1 : A1, ..., idn : An ⊢ C : Unit
-----------------------------------------------------------
G ⊢ id(A1 id1, ... idn){C} :: id : (A1 *...* An) → Unit

(Recursive Proc)
G, id1:A1, ... idn:An, id:(A1 *...* An)→Unit ⊢ C : Unit
------------------------------------------------------
G ⊢ id(A1 id1, ... An idn){C} :: id : (A1*...* An)→Unit

(Sequenza)
G ⊢ D1 :: G1         G, G1 ⊢ D2 :: G2
-------------------------------------
        G ⊢ D1; D2 :: G1, G2
```

Array

```
A[B] con A memorizzabile, B enumerazione
Nuove espressioni, destre o sinistre:
- LE ::= id | LE[RE]
- RE ::= LE | const | RE binop RE | unop RE
Assegnamento diventa: C ::= LE = RE | ...
Giudizi:
• G ⊢l LE : A (LE denota una locazione di tipo A)
• G ⊢r RE : A (RE denota un valore di tipo A)

(Assign)
G ⊢l LE : A         G ⊢r RE : A
------------------------------
     G ⊢ LE = RE : Unit

Left-part:
(Var)
G, x:A, G' ⊢l x : A

(Array)
G ⊢l E : A[B]         G ⊢r E1 : B
---------------------------------
        G ⊢l E[E1] : A

Right-part:
(Left-Right)
 G ⊢l E : A
-------------
 G ⊢r E : A

Definizione di un id di tipo vettore:
------------------------------
G ⊢ A[B] id :: id : A[B]
```

FLEX - YACC

YACC

```
// Modificare YYTYPE
%union {
  int v;
}

%token ABCD

%left  ABCD   // Precedenza
%right

%%
input:
     | input line
;

line: '\n'                      {}
    | P '\n'                    { printf("%d\n", $1); }
;

P: ABCD                         { $$ = $1; }
 | P OP_PLUS P                  { $$ = ... }
```

FLEX

```
DIGIT     [0-9]

%%
{DIGIT}+              { yylval = atoi(yytext); return NUMBER; }
"+"                   { return OP_PLUS; }
"("                   { return yytext[0]; }
%%
```