Analizzatori lessicali

ESERCIZIO 6:
Selezionare in un file di testo le stringhe di caratteri che rappresentano in notazione binaria un numero multiplo di 3. Solo in numeri multipli di 4 vengono stampati in uscita, separati da uno spazio, la restante parte del testo viene eliminata.
Il controllo di divisibilità deve essere implementato usando opportune espressioni regolari, e non l'operazione di divisione-resto.
Multipli di 4: terminano con "00".

```
multipliTre (0|(1(01*0)*1))*
nonValidi .

%%
{multipliTre}(" "|\n|\t) {if((yytext[yyleng-2] == '0') && (yytext[yyleng-
3] == '0'))
                            printf("%s ",yytext);
                        }
{nonValidi} {printf("");}

%%
void yyerror (const char *str) { fprintf(stderr, "errore: %s\n", str); }
int yywrap () { return 1; }
int main () {
    yylex();
    return 0;
}
```

Analizzatori sintattici

LEX:

ESERCIZIO 2:
Il linguaggio formato da espressioni aritmetiche scritte in notazione polacca diretta e costruite a partire dalle costanti intere e le 4 operazioni aritmetiche.
L'analizzatore deve valutare l'espressione ricevuta in ingresso.
I numeri con il (-) davanti vanno per forza messi attaccati

```
%{
#include <stdio.h>
#include "y.tab.h"
extern yylval;
%}
```

```
%%
[0-9]+ {yylval = atoi(yytext); return NUMBER;}
[a-zA-Z0-9]+ { printf("Questo non è un numero \n");}
-[0-9]+ {yylval = atoi(yytext); return NUMBER;} //i numeri con il (-)
davanti vanno per forza messi attaccati
"+" {return yytext[0];}
"-" {return yytext[0];}
"/" {return yytext[0];}
"*" {return yytext[0];}
[ \t]+ /* ignora spazi bianchi e fine linea e stampa ili risultato se non
è vuota*/
\n     { return yytext[0]; }

%%
```

YACC:

```
%{
    #include <stdio.h>
    int yylex();
    int yyerror(char *s);
%}

%token NUMBER
%left '+' '-'
%left '*' '/'

%%
input:
      | input line
;
line : '\n'
       | exp '\n'{ printf("Result = %d\n", $$);}
;
exp :
  '+' exp  exp { $$ = $2 + $3; }
  |'-' exp  exp { $$ = $2 - $3; }
  |'*' exp  exp { $$ = $2 * $3; }
  |'/' exp  exp { $$ = $2 / $3; }
  | '(' exp ')' { $$ = $2; }
  | NUMBER { $$ = $1; }
;
%%

int main() {
  printf("Enter the expression\n");
  yyparse();
}

int yyerror(char* s) {
  printf("\nExpression is invalid\n");
}
```

Haskell

liste e matrici

```
-- 3. Si scriva una funzione che calcoli una lista con tutte le
combinazioni su n elementi. Si usi opportunamente la map.
-- map :: (a -> b) -> [a] -> [b]
allComb :: (Integral a) => a -> [(a,a)]
allComb n = foldl (++) [] (map (\ x -> [(x,y) | y <- [1..n]]) [1..n])


quickSort :: (Integral a) => [a] -> [a]
quickSort [] = []
quickSort (x:xs) = quickSort [y | y <- xs , y<x] ++ (x:quickSort[y | y <-
xs, y>=x]) --lo applico sulla prima meta e sulla seconda meta


-- 6. Scrivere una funzione che costruisce, a partire da una lista di
numeri interi (provate poi a genera- lizzare), una lista di coppie in cui
-- (a) il primo elemento di ogni coppia `e uguale all'elemento di
corrispondente posizione nella lista originale e
-- (b) il secondo elemento di ogni coppia `e uguale alla somma di tutti
gli elementi antecedenti della lista originale.

--faiCoppie' :: (Integral a) => [a] -> [(a,a)]
faiCoppie' lista = [(x, foldl (+) 0 (take ((getIndex lista x) - 1) lista)
) | x <- lista]   --prendo tutto l'array prima dell'indice


-- 7. Si scriva una funzione Haskell shiftToZero che data una lista
costruisce un nuova lista che contiene gli elementi diminuiti del valore
minimo.

shiftToZero :: (Integral a) => [a] -> [a]
shiftToZero lista = let min = minimum lista     --salvo il valore del
minimo all'inizio
                    in
                    map (\ x -> x-min) lista   --lo sottraggo da ogni
cella


-- 1. Si scriva una funzione matrix_dim che data una matrice ne calcola le
dimensioni, se la matrice è ben formata, altrimenti restituisce (-1,-1).
-- una matrice è una lista di liste, quindi ogni elemento della lista è
una lista a sua volta

matrixDim matrice = (foldl (\ acc new -> acc + 1) 0  matrice,   -- conto
le righe
```

```
                    foldl (\ acc new -> if ((length new) == acc) then acc
else -1) (length (head matrice)) (tail matrice)) --conto le colonne
```

alberi

```
-- BST

data (Ord a, Read a, Show a) => BST a = Void | Node {
    val :: a,
    left, right :: BST a
} deriving (Eq, Ord, Read)

-- 5. Si scriva una funzione per eseguire l'inserimento di un dato x in un
albero t.
bstInsert :: (Integral a, Num a, Ord a, Read a, Show a) => BST a -> a ->
BST a
bstInsert Void x = (Node x Void Void)
bstInsert (Node val left right) x
            | val > x = (Node val left (bstInsert right x))
            | val < x = (Node val (bstInsert left x) right)
            | otherwise = (Node val left right)

-- 6. Si scriva una funzione bst2List che calcola la lista ordinata degli
elementi di un BST. Ci si assicuri di scrivere una funzione lineare.
bst2List :: (Integral a, Num a, Ord a, Read a, Show a) => BST a -> [a]  --
INORDER
bst2List Void = []
bst2List (Node val left right) = bst2List left ++ [val] ++ bst2List right

-- Alberi generici

data (Eq a,Show a) => Tree a = Void | Node a [Tree a] deriving Eq

-- Es1: si scriva una generalizzazione della funzione foldr delle liste
per Alberi Generici che abbia il seguente tipo:
--  treefold :: (Eq a, Show a) => (a->[b]->b) -> b -> Tree a -> b
treefold :: (Eq a, Show a) => (a -> [b] -> b) -> b -> Tree a -> b
treefold _ z Void = z   --nodo foglia, è bottom up
treefold f z (Node val []) = f val [z]
treefold f z (Node val children) = f val (map (treefold f z) children)


-- Es2: si scriva una funzione height per calcolare l'altezza di un albero
usando opportunamente la treefold dell'Esercizio 1.
--  Si attribuisca altezza -1 all'albero vuoto
height :: (Eq a, Show a) => Tree a -> Int
height albero = treefold (\ new acc -> maximum acc + 1) (-1) albero --
l'albero parte dalle foglie, essendo che e' foldR allora quelle saranno le
prime ad essere
```

```
-- analizzate. Risalendo per ogni nodo che trovo incremento. E' bottom up
non top down
```