***BUNEA ALEXANDRA-AMALIA***

***GRUPA: 232/1***

**TEMA 1. ANALIZOR LEXICAL**

* ***ENUNT***

Scrierea unui analizor lexical pentru un minilimbaj de programare (MLP), ales ca subset al unui limbaj existent.

1. **Se cer textele sursa a 3 programe** (versiune electronica) care respecta specificatiile MLP date si care rezolva urmatoarele probleme:

* Calculeaza perimetrul si aria cercului de o raza data
* Determina cmmdc a 2 nr naturale
* Calculeaza suma a n numere reale citite de la tastatura

1. **Specificarea minilimbajului de programare.** Limbajul trebuie sa contina anumite instructiuni si tipuri de date:

* 2 tipuri de date simple si un tip compus
* Instructiuni:
  + O instructiune de atribuire
  + O instructiune de intrare/iesire
  + O instructiune de selectie (conditionala)
  + O instructiune de ciclare

Pe langa acestea, vor exista unele restrictii suplimentare referitoare la identificatori si constant (vezi sectiunea 3.1). Se cere ca specificarea sa fie suficient de generala astfel incat sa descrie constructiile limbajului folosite pentru scrierea programelor de la punctul 1.

1. **Implementarea analizorului lexical.** Analizorul lexical accepta la intrare un fisier text reprezentand un program sursa si intocmeste ca date de iesire tabelele : FIP – forma interna a programului sursa si TS – tabela de simboluri. In plus, programul va trebui sa semnaleze erorile lexicale si locul in care apar. Analizoarele lexicale se vor diferentia dupa urmatoarele criterii:
2. Identificatori - de lungime oarecare nedepasind 250 caractere
3. Tabela de simboluri - separat pentru identificatori si constante
4. Organizarea tabelelor de simboluri - tabel de dispersie (hash)

* ***TEXTELE SURSA ALE PROGRAMELOR***
* Perimetrul si aria unui cerc de raza data

start\_program

aria : real ; perimetru : real ; raza : real ; PI : real ;

start

PI = 3.14 ;

read ( raza ) ;

aria = PI \* ( raza \* ( raza ) ) ;

perimetru = 2 \* ( PI \* ( raza ) ) ;

write ( aria ) ;

write ( perimetru ) ;

finish

finish\_program

* CMMDC a doua numere naturale

start\_program

a : positive\_int ; b : positive\_int ;

start

read ( a ) ;

read ( b ) ;

if ( a =:= 0 ) then

start

write ( b ) ;

finish

elif ( b =:= 0 ) then

start

write ( a ) ;

finish

else

start

while ( a =/= b )

start

if ( a > b ) then

start

a = a – ( b ) ;

finish

else

start

b = b – ( a ) ;

finish

finish

write ( a )

finish

finish

finish\_program

* SUMA A n NUMERE REALE CITITE DE LA TASTATURA

start\_program

n : positive\_int ; i : positive\_int ; numbers : real [ 100 ] ; sum : real ;

start

read ( n ) ;

i = 0 ;

while ( i < n )

start

read ( nr [ i ] ) ;

i = i + ( 1 ) ;

finish

sum = 0 ;

i = 0 ;

while ( i < n )

start

sum = sum + ( nr [ i ] ) ;

i = i + ( 1 ) ;

finish

write ( sum ) ;

finish

finish\_program

* ***SPECIFICAREA MINILIMBAJULUI DE PROGRAMARE***
* Alfabetul limbajului
* Litere mari ale alfabetului englez : A – Z
* Litere mici ale alfabetului englez : a – z
* Cifre ale sistemului zecimal : 0 – 9
* Atomi lexicali
* Identificatori : [a-zA-Z][ a-zA-Z0-9]\*{0,250}
* Constante : [0-9]+\.?[0-9]\*
* Cuvinte cheie/rezervate : start\_program, finish\_program, start, finish, var, positive\_int, real, real[], read, write, if, elif, else, then, while, for
* Operatori: =, +, =:=, =/=, <, >, -, \*, /, <=, >=
* Delimitatori: ;, (, ), [, ] , :
* Gramatica limbajului

<program> → <intrare\_prog> <lista\_decl> <bloc> <iesire\_prog>

<intrare\_prog> → start\_program

<iesire\_prog> → finish\_program

<lista\_decl> → <decl> ; <lista\_decl> | <decl> ;

<decl> → ID : <tip\_data>

<tip\_data> → <tip\_simplu> | <tip compus>

<tip\_simplu> → positive\_int | int | real | char

<tip\_compus> → positive\_int [ CONST ] | int [ CONST ] | real [ CONST ] | char [ CONST ]

<bloc> → <intrare\_bloc> <lista\_instr> <iesire\_bloc>

<intrare\_bloc> → start

<iesire\_bloc> → finish

<lista\_instr> → <instr> ; <lista\_instr> | <instr> ;

<instr> → <atribuire> | <instructiune\_IO> | <instructiune\_selectie> | <instructiune\_ciclare>

<atribuire> → ID = <expresie>

<expresie> → <termen> <op\_aritmetic> ( <expresie> ) | <termen>

<termen> → CONST | ID | <termen\_compus>

<termen\_compus> → ID [ ID | CONST ]

<op\_aritmetic> → + | - | \* | / | %

<instructiune\_IO> → read ( ID ) | write ( ID )

<instructiune\_selectie> → <ramura\_if\_1> | <ramura\_if\_1> <ramura\_if\_2> | <ramura\_if\_1> <ramura\_if\_3> | <ramura\_if\_1> <ramura\_if\_2> <ramura\_if\_3>

<ramura\_if\_1> → if ( <lista\_cond> ) then <bloc>

<ramura\_if\_2> → <ramura\_if\_2\_simplu> <ramura\_if\_2> | <ramura\_if\_2\_simplu>

<ramura\_if\_2\_simplu> → elif ( <lista\_cond> ) then <bloc>

<ramura\_if\_3> → else <bloc>

<lista\_cond> → <cond> <op\_logic> <lista\_cond> | <cond>

<cond> → <expresie> <op\_relational> <expresie>

<op\_logic> → && | ||

<op\_relational> → =:= | =/= | < | > | <= | >=

<instructiune\_ciclare> → while ( <lista\_cond> ) <bloc>

* Codificare

|  |  |
| --- | --- |
| **SIMBOL** | **COD** |
| CONST | 0 |
| ID | 1 |
| start\_program | 2 |
| finish\_program | 3 |
| positive\_int | 4 |
| int | 5 |
| real | 6 |
| char | 7 |
| start | 8 |
| finish | 9 |
| read | 10 |
| write | 11 |
| if | 12 |
| then | 13 |
| elif | 14 |
| else | 15 |
| while | 16 |
| ; | 17 |
| , | 18 |
| : | 19 |
| [ | 20 |
| ] | 21 |
| ( | 22 |
| ) | 23 |
| = | 24 |
| =:= | 25 |
| =/= | 26 |
| < | 27 |
| > | 28 |
| <= | 29 |
| >= | 30 |
| + | 31 |
| - | 32 |
| \* | 33 |
| / | 34 |
| && | 35 |
| || | 36 |