Cuprins :

Introducere

Calculatorul (PC-ul), odata cu raspandirea sa la scara larga si folosirea cat mai eficienta, a facilitat dezvoltarea rapida, a adus inovatii si a revolutionat mai toate domeniile: de la medicina la aeronautica, de la industria filmului la cea a telecomunicatiilor si cercetare stiintifica.Practic in ziua de azi nu se poate concepe viata fara acest instrument care s-a integrat in viata noastra si care a devenit indispensabil.

Dar ca orice instrument acesta trebuie ,,manuit" cum trebuie pentru a fi eficient. O problema majora ce a stat in calea utilizarii la scara cat mai larga a calculatorului personal a fost si inca este faptul ca PC-ul nu intelege limbajul uman (e pana la urma urmei doar masina), si de aceea pentru a-l utiliza, omul trebuie sa-i dea comenzi, numite instructiuni, sa vorbeasca pe ,,limba" lui. Instructiunile sunt grupate in programe (numite executabile sau binare deoarece contin caractere intr-o ordine care nu este inteligibila omului), ce pot fi lansate in executie de cate ori este nevoie.

La inceput programele erau scrise direct in cod masina (cod direct executabil) insa acest mod de scriere avea dezavantaje majore : programele erau foarte greu de scris (pentru a face o simpla adunare un programator trebuia sa dea codul operatiei de adunare si apoi adresele celor doi operanzi), iar de depanat aproape ca nici nu incapea vorba daca un program depasea 50 - 100 de instructiuni.

Urmatorul pas l-au reprezentat limbajele de asamblare (exemple de asambloare: masm, tasm, nasm, as, gas), care au facut viata programatorilor putin mai frumoasa si mai usoara (o adunare in limbaj de asamblare se scrie sub forma : add ax,3 , adica aduna 3 la registrul ax). Avantajul limbajelor de asamblare este viteza mare si accesul direct la resursele calculatorului, marele lor dezavantaj este insa portabilitatea (un program portabil este un program care poate fi compilat/executat pe mai multe platforme (diferite tipuri de calculatoare: IBM-PC, Apple, DEC, SPARC, Alpha etc. si/sau sisteme de operare diferite: Windows, Unix, Linux, MacOS, BeOS, FreeBSD etc.)).

Dupa limbajelor de asamblare au aparut limbajelor de nivel inalt, limbaje structurate care ulterior au inglogat si conceptul de programare orientata-obirect (Ada, Pascal, C/C++) . Acestea sunt mult mai apropiate de limbajul uman (au o sintaxa ce contine cuvine din limba engleza (majoritatea, al meu nu !) si expresiile matematice se pot scrie simplu (o adunare se scrie sub forma : a =b+3).

Pana acum am vorbit insa de compilatoare (programe care transforma un program sursa (inteligibil oamenilor) in program obiect (ce poate fi inteles de calculator)). Diferenta dintre un compilator si un interpretor este aceea ca interpretorul nu face decat sa interpreteze (execute) instructiunile imediat ce le ,,citeste", acesta nu mai face nici o traducere in cod masina (programele scrise pentru interpretor se numesc scripturi sau scenarii).

Interpretoarele au aparut cam in acelasi timp cu limbajele de nivel inalt:Pascal, C dar nu au fost atat de populare deoarece un script ruleaza mai incet decat un program executabil si pe vremea aceea viteza procesorului era foarte mica .

Dezavantajul ca scripturile ruleaza cu 20 30 % mai incet decat programele compilate echivalente, este compensat prin faptul ca pentru a testa modificarile facute

nu mai e novoie de o noua compilare. Timpul de compilare pentru un proiect mare poate dura foarte mult, kernelul de linux are un timp de compilare de ordinul zecilor de minute, insa pentru asemenea proiecte nu se pune problema interpretarii lor.

Neexistand compilarea, depanarea programelor (gasirea erorilor in program) este mult mai usoara si mai putin consumatoare de timp, timp de care toata lumea cam duce lipsa in secolul XXI . Exemple de limbaje-interpretor : Java , Perl, Python, Tcl/Tk si multe altele.

Motivatia

Ceea ce m-a motivat a fost curiozitatea, vroiam sa vad daca sunt in stare sa fac asa ceva, si apoi m-am gandit ca ar fi mult mai usor tuturor celor care ar dor sa invete sa programeze daca ar avea un limbaj mai apropiat de vorbirea curenta (nu toti vorbesc engleza, mai ales cei din clasele mici).De aceea am hotarat sa distribui programul sub licenta GNU GPL (GNU General Public Licence) versiunea 2 si nu alta versiune ulterioara sau anterioara , pentru ca oricine doreste, sa poata invata din sursele programului .

Descriere

Programul IPC (Interpretor de pseudo-cod) este scris in limbajul C. Are o structura modulara, usor de modificat. Sintaxa este asemanatoar pseudo-codului, pentru a fi usor de inteles. Are un numar de 18 cuvinte cheie si anume : "caracter", "intreg", "real", "sub", "sfsub", "daca", "altfel", "sfdaca", "pentru", "sfpentru", "ctimp", "sfctimp", "repeta", "pana", "citeste", "scrie".

!!! Este sensibil la tipul caracterelor (case-sensitive), adica "intreg" si "Intreg" sunt doua constructii diferite pentru interpretor.

Comentariile : sunt pe o singura linie, incep de la secventa // si se termina la sfarsitul liniei .

Ex.

```
// ce scriu aici este un comentariu
scrie " un text"; // acesta este un alt comentariu
```

Tipurile predefinite de date sunt : caracter, intreg, real; nu se pot defini alte tipuri de date decat cele existente, si sunt memorate intern ca numere reale in simpla precizie.

Declararea variabilelor :

Se poate face oricand pe parcursul programului, mai putin in interiorul structurii decizionale (daca - altfel - sfdaca), si in interiorul ciclurilor (pentru - sfpentru , repeta - pana si ctimp sfctimp), si are se face asemanator ca in C, tipul de date este numele variabilelor separate prin virgula si declaratia se termina prin punct-virgula. Variabilele pot fi initializate la declarare.

Ex.

```
intreg a,b = 3;  // se declara doi intregi
caracter c;      // o variabila de tipul caracter
real r;          // o alta varabila de tipul real
```

Afisarea si citirea variabilelor:

Se face cu ajutorul instructiunilor si citeste.

scrie trebuie sa fie urmata de cel putin un text incadrat intre ghilimele duble sau de numele unei variabile. Acestea se pot repeta in orice ordine , separate prin virgula (,) . Declaratia se termina cu caracterul punct-virgula (;) .

In textul de afisat se pot folosi secventele speciale de caractere : ·

- \n care va face trecerea la o linie noua
- \t tabulare (TAB)
- \r face trecerea la inceputul liniei
- \" scrie apostroful dublu
- \\ scrie caracterul salsh (\ )

citeste accepta unul sau mai multe nume de variabile separate prin virgula (,), si trebuie urmata de caracterul punct-virgula (;).

Ex.

`scrie` "mesaj 1" , nume_variabila_1 , "mesaj 2", nume_variabila_2 , ... ;
`citeste` nume_variabila_1 , nume_variabila_2 , ... ;

Structura coditionala:

Am implementat structura de decizie astfel :

```
daca ( <expresie> )<bloc_instructiuni_adevarat>
[ altfel <bloc_instructiuni_fals> ]
sfdaca
```

Daca in urma evluarii, expresia <expresie> are o valoare diferita de 0 (este adevarata) atunci se executa <bloc_instructiuni_adevarat> si se trece la urmatoarea instructiune dupa sfdaca; daca <expresie> are valoarea 0 (este falsa) atunci se executa <bloc_instructiuni_fals> (ramura altfel), daca aceasta exista. Parantezele din jurul <expresie> sunt obligatorii .

Ex.

```
daca ( gama == 4 )
     scrie "\n expresia este adevarata" ;
altfel
     scrie "\n gama are valoarea : ",gama ;
sfdaca
```

Structuri repetitive:

Structuri repetitive sunt :

- `pentru - sfpentru` -> structura repetitiva cu contor
- `repeta - pana` -> structura repetitiva cu testul conditiei la final
- `ctimp - sfctimp` -> structura repetitiva cu testul conditiei la inceput

Structura repetitiva cu contor are forma :

```
pentru ( expresie_1 ; expresie_2 ; expresie_3 )
     <bloc_instructiuni>
sfpentru
```

Expresie_1 trebuie sa fie numele unei variabile care va fi variabila contor si care poate fi initializata o singura data, la inceputul ciclului .

Daca expresie_2 este adevarata atunci se intra in ciclu, altfel nu.

Expresie_3 reprezint pasul cu care se face cresterea, trebuie sa fie o atribuire in care apare variabila in expresie_1 .

Structura repetitiva cu testul conditiei la final are forma :

```
repeta
     <bloc_instructiuni>
pana ( <expresie> )
```

Acest ciclu va determina executarea cel putin o data a blocului de instructiuni
<bloc_instructiuni>. Repetarea se va face pana cand <expresie> va avea valoarea falsa.

Structura repetitiva cu testul conditiei la inceput are forma :

```
ctimp ( <expresie> )
     <bloc_instructiuni>
sfctimp
```

Se evalueaza expresia <expresie> si daca aceasta este adevarata se intra in ciclu si se executa <bloc_instructiuni> pana cand valoarea expresiei devine falsa .

Subrutine:

Subrutinele nu sunt inca implementate.

!!! **Programul se termina obligatoriu cu `.' !!!**

Acesta este un program scris in limbajul IPC :

```
//
// program de test pentru atestat
//

intreg i , j , divizori  , numar2  , N ;

// este implementat algoritmul pentru aflarea numerelor prime
// mai mici sau egale cu N citit de la tastatura

scrie "\n\tAcest program afiseaza primele N numere prime, dati N:";
citeste N ;

pentru (i=1; i< N ; i= i+1)
       divizori = 0 ;
       pentru (j = 1 ; j < i ; j = j+1 )
            daca ( i % j == 0 )
               divizori = divizori + 1 ;
            sfdaca
       sfpentru
       daca ( divizori == 0 )
         // scrie "numarul ",i," are ","1 divizori";
        scrie "\nnumarl 1 este prim";
       altfel
          // scrie "\nnumarul ",i," are ", divizori," divizori";
          daca ( divizori == 2 )
             scrie "\nnumarul ",i," este prim";
          sfdaca
       sfdaca
sfpentru

intreg numar1 = 0 ;
// cicleaza pana cand primeste un numar negativ;
repeta
    scrie "\ndati un numar negativ :";
    citeste numar1;
pana ( numar1 < 0 )

// acesta cicleaza pana cand numarul citit anterior
// ajunge la o noua valoare citita tot de la tastatura
scrie "\nintroduceti un numar";
citeste numar2;

ctimp ( numar1 != numar2 )
      scrie "\n",numar1;
      daca ( numar1 < numar2 )
         numar1 = numar1++;
      altfel
        numar1 = numar1--;
      sfdaca
sfctimp
.  // marcheaza sfarsitul programului
```

Modul de realizare

Programul face „traducerea” codului folosind translatarea recursiv-descendenta, (recursiv-descent parsing). Expresiile sunt „traduse” folosind algoritmul numit „precedence climbing” (escaladarea nivelelor de precedenta).

Structurile si tipurile de date folosite sunt :

```
struct t_atom
{
  char nume[MARIME_IDENTIFICATOR];
  int cod;
}
typedef struct_t atom ATOM;
struct s_identificator
{
  char nume[MARIME_IDENTIFICATOR];
  int tip;
  float valoare_f;
}
typedef struct s_identificator variabila_t;
struct _arbore
{
  char nume[MARIME_IDENTIFICATOR];
  int tip;
  float valoare;
  struct *vecin_st,*vecin_dr;
}
typedef struct _arbore arbore;
```

Tipul de date ATOM este folosit la impartirea fluxului de caractere in atomi lexicali, si retine numele si codul asociat numelui.

Tipul variabila_t este folosit penru a retine date despre variabilele declarate in script. Retine numele, tipul variabilei (intreg, caracter, real) cat si valoarea acesteia.

Tipul de date arbore este folosit in traducatorul de expresii. In urma algoritmului „precedence climbing” rezultand un arbore care este apoi parcurs, rezultand valoarea expresiei.

Variabilele globale, declarate in fisierul main.c sunt :

long marime_fisier = 0; care retine marimea fisierului sursa

long pozitie_in_fisier = 0; este indicativul de pozitie in fisier

long numar_de_linii = 1; retine numarul de linii

long numar_de_variabile = 0; numarul de variabile declarate

char* sursa; referinta catre un sir de caractere ce repreinta fisierul citit in memorie

int sfarsit_de_fisier = 0; e folosita pentru a semnala sfarsitul de fisier

ATOM atom,atom_vechi; retine atomii lexicali

arbore *tree; folosit in rezolvarea expresiilor

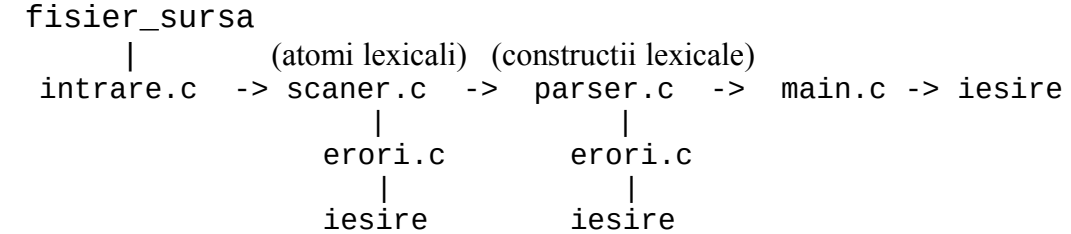
Structura programului si algoritmul:

Fisiserele sursa sunt: erori.c, intrare.c, scaner.c, parser.c, main.c.Fisierele header corespunzatoare sunt erori.h, intrare.h , scaner.h, parser.h si baza.h.

Algoritmul de baza este urmatorul :

Se citeste fisierul in memorie (variabila sursa), si se imparte in atomi lexicali cu ajutorul subrutinelor din scanner-ul lexical (scaner.c). Analiza lexicala este recursiv-descendenta , fiecare atom lexical este transmis mai departe parserului, care verifica daca declaratia e corecta din punct de vedere sintactic.

Programul e structurat in felul urmator :



Subrutinele scanerului lexical sunt :

- get_name() care verifica si intoarce in variabila atom urmatorul identificator .
- get_number() care intoarce in atom, urmatorul numar .
- get_operator() care returneaza urmatorul operator din sursa .
- get_atom() intoarce urmatorul atom lexical cat si codul acestuia .
- sari_spatiu() sare peste spatiile albe si peste comentarii .

Pentru a accesa sirul de caracter, get_name, get_number, get_operator si sari_spatiu se folosesc de rutinele definite in intrare.c : next_char(), curent_char() si view_next_char().

Fiecare apel la get_atom() intoarce atomul lexical urmator si se poate scrie un parser predictiv.

main.c

```
/*
 *  ipc - interpretor/compilator pentru C
 *  copyright (c) 2004 stan ioan-eugen
 *
 *  e-mail: ieugen222@mailsurf.com
 *  tel   : 0244/528 096
 *  adresa: str. aleea arnauti nr.5,bl.69,sc.A,ap.8
 *  Ploiesti,Prahova,Romania
 *     Acest program e gratuit ; il puteti redistribui si / sau modi
 * fica sub conditiile si termenii Licentei Publice Generale GNU
 * (GNU General Public License) versiunea 2 sau  orice alta versiune
 * ulterioara.
 *     Acest program e distribuit in speranta ca va fi folositor,
 * dar e distribuit FARA NICI O GARANTIE, implicita sau explicita.
 *     Cititi GNUGPL, General Public License pentru detalii .
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

#include "include/intrare.h"
#include "include/scaner.h"
#include "include/parser.h"
#include "include/baza.h"


long marime_fisier = 0;
long pozitie_in_fisier = 0;
long numar_de_linii = 1;
long numar_de_variabile = 0;
char* sursa;
int sfarsit_de_fisier = 0;
ATOM atom,atom_vechi;
arbore *tree;
// definit in parser.h; sa fac alocare dinamica
variabila_t variabile[NUMAR_MAX_DE_VARIABILE];

/*    LEXIC =  macro care se expandeaza la un sir de siruri de
  caractere  si care reprezinta cuvintele cheie ale limbajului
*/
LEXIC ;

int main (int argc, char *argv[])
{
  FILE *fin;
  long i = 0;
  if (argc == 1)
    {
      printf ("Mod de utilizare : %s file_name\n",argv[0]);
      exit (0);
    }
  if ((fin = fopen (argv[1], "r")) == NULL)
    {
```

```
      fprintf (stderr, "\nEroare la deschiderea fisierului %s\n",argv
[1]);
      exit (1);
    }
  marime_fisier = 0;
  fseek (fin, 0, SEEK_END);  // calculam marimea fisierului
  marime_fisier = ftell (fin);  // daca e vid inseamna ca a aparut o
eroare
  if (marime_fisier == 0)
    {
      fprintf (stderr, "\nFisier vid sau eroare de citire\n");
      exit (1);
    }
  fseek (fin, 0, SEEK_SET);
  if ((sursa = (char *) malloc (marime_fisier * sizeof (char))) ==
NULL)
    {
      fprintf(stderr,"\nEroare la alocarea de memorie pentru incarcarea
fisierului\n");
      exit (1);
    }
  i = 0;
  while (!feof (fin))    // citim fisierul in sursa
      sursa[i++] = fgetc (fin);
  sursa[marime_fisier] = '\0';
  pozitie_in_fisier = 0;

  main_parse();
  return 0;
}
```

erori.c

```
#include <stdio.h>
#include <stdlib.h>
#include "include/baza.h"
#include "include/erori.h"
#include "include/intrare.h"

extern long pozitie_in_fisier;
extern long numar_de_linii;
extern long marime_fisier;
extern ATOM atom_vechi;
extern ATOM atom;

void eroare (const char *mesaj_eroare)
{
  fprintf(stderr,"\n Eroare :(linia: %li, caracterul %li/%li ):dupa `%
s` , %s (%s)\n",numar_de_linii , pozitie_in_fisier, marime_fisier,
atom_vechi.nume, mesaj_eroare , atom.nume);
  exit (1);
}
```

intrare.c

```
#include <stdio.h>
#include <stdlib.h>
```

```
#include "include/baza.h"
#include "include/intrare.h"
#include "include/erori.h"

extern long pozitie_in_fisier;
extern long numar_de_linii;
extern long marime_fisier;
extern char* sursa;
extern int sfarsit_de_fisier;

char next_char ()
{
  if (sursa[pozitie_in_fisier] == '\n') numar_de_linii++;
  if ( pozitie_in_fisier == marime_fisier ) sfarsit_de_fisier = 1;
  pozitie_in_fisier++;
  return sursa[pozitie_in_fisier-1];
}
char curent_char ()
{
  if (pozitie_in_fisier == 0) return sursa[pozitie_in_fisier];
  else return sursa [pozitie_in_fisier-1];
}
char view_next_char ()
{
  return sursa[pozitie_in_fisier];
}
```

scaner.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include "include/baza.h"
#include "include/intrare.h"
#include "include/erori.h"
#include "include/scaner.h"

extern long pozitie_in_fisier;
extern long marime_fisier;
extern long numar_de_linii;
extern ATOM atom;
extern ATOM atom_vechi;
extern char* sursa;

void muta()
{
  strcpy(atom_vechi.nume,atom.nume);
  atom_vechi.cod = atom.cod;
}
/* sare peste spatii si comentarii stil C++ */
void sari_spatiu()
{
  char c = curent_char();
  do
    {
```

```
      while ( isspace (c) ) c = next_char();
      if ( c == '/' )
      {
        c = view_next_char();
        if ( c == '/' )
          {
            while ( c!='\n') c = next_char();
            c = next_char();
          }
        else break;
      }
    }while( ( isspace(c) || curent_char() == '/')
         && pozitie_in_fisier <= marime_fisier);
}
int isoperator (char c)
{
  return ( c==','|| c=='|' || c=='6'|| c=='^' || c=='='||
      c== '!'||     c=='<' || c== '>' || c== '+' || c== '-'
      || c=='%'   || c=='/' || c=='*'  || c== '~' || c=='('
      || c==')' || c=='[' || c== ']'|| c=='.' || ';' || c=='\''
      || c=='"');
}
void get_name()
{
  int i;
  muta();
  if ( isalpha( curent_char() ) == 0 )
      eroare ("nume de identificator asteptat");
  atom.cod = COD_NUME;
  i = 0;
  do
    {
      atom.nume[i] = curent_char();
      next_char();
      i++;
    }
  while ( isalnum(curent_char())  && i < MARIME_IDENTIFICATOR);
  atom.nume[i]='\0';
  if ( i == MARIME_IDENTIFICATOR && isalnum(curent_char()) )
      eroare ("nume de identificator prea lung");
}
void get_number()
{
  int i;
  muta();
  if ( isxdigit ( curent_char() ) == 0)
      eroare ("numar asteptat");
  atom.cod = COD_NUMAR;
  i = 0;
  do
    {
      atom.nume [i] = curent_char();
      next_char();
      i++;
    }
  while ( isxdigit(curent_char()) && i < MARIME_IDENTIFICATOR);
```

```
  atom.nume[i]='\0';
  if ( i == MARIME_IDENTIFICATOR && isxdigit(curent_char()) )
      eroare("numar prea mare (lung al dracu)");
}
void get_operator()
{
  muta();
  if ( isoperator(curent_char()) == FALSE )
      eroare("operator asteptat");
  atom.nume[0] = curent_char();
  atom.nume[1] = '\0';
  atom.nume[2] = '\0';
  atom.cod = COD_OPERATOR;
  switch (atom.nume[0])
    {
    case '+':
      // daca urmatorul caracter e '+' -> '++'
      if ( view_next_char() == '+')
      atom.nume[1] = next_char();
      break;
    case '-':
      // daca e '-' -> '--'
      if (  view_next_char() ==  '-' )
      atom.nume[1] = next_char();
      break;
    case '>':
      // daca e '>' -> '>>' , daca e '=' -> '>='
      if ( view_next_char() == '>' || view_next_char() == '=')
      atom.nume[1] = next_char();
      break;
    case '<':
      // daca urm. e '<' -> '<<', daca e '=' -> '<='
      if ( view_next_char() == '<' || view_next_char() == '=')
      atom.nume[1] = next_char();
      break;
    case '=':
      // daca e '=' -> '=='
      if ( view_next_char() == '=')
      atom.nume[1] = next_char();
      break;
    case '!':
      // deca e '=' -> '!='
      if ( view_next_char() == '=')
      atom.nume[1] = next_char();
      break;
    case '&':
      // daca e '&' -> '&&'
      if ( view_next_char() == '&')
      atom.nume[1] = next_char();
      break;
    case '|':
      // daca e '|' -> '||'
      if ( view_next_char() == '|')
      atom.nume[1] = next_char();
      break;
    }
```

```
  next_char();
}
void get_atom()
{
  sari_spatiu();
  if ( isalpha( curent_char()) )
    get_name();
  else  if ( isdigit( curent_char()) )
    get_number();
  else  if ( isoperator(curent_char()) )
    get_operator();
  else eroare("caracter neasteptat");
  atom.cod = scan_cod();
}
void get_it(const char *valoare)
{
  get_atom();
  if ( strcmp(atom.nume,valoare) != 0)
      eroare("valoarea asteptata nu a fost gasita");
}
int scan_cod()
{
  int i = 0;
  char **tmp;
  char *lexic[NUMAR_DE_CUVINTE_CHEIE]=
{"caracter","intreg","real","sub","sfsub","daca","altfel","sfdaca","pen
tru","sfpentru","ctimp","sfctimp","repeta","pana","citeste","scrie"};

  tmp = lexic;
  i = 0;
  if (atom.cod == COD_NUME)
    {
      while ( *tmp != NULL )
      {
        if ( strcmp(atom.nume,*tmp) == 0 ) return i+1;
        tmp++;
        i++;
      }
      return COD_NUME;
    }
  if (atom.cod == COD_OPERATOR)
    {
      if ( atom.nume[1]=='\0') //operatorul are un sg. caracter ?
      {
        switch (atom.nume[0])
          {
          case '+':
          case '-':
          case '/':
          case '*':
          case ',':
          case ';':
          case '=':
          case '!':
          case '\'':
          case '"':
```

```
          case '|':
          case '&':
          case '^':
          case '<':
          case '>':
          case '%':
          case '~':
          case '(':
          case ')':
          case '[':
          case ']':
          case '.':
            return atom.nume[0];
            break;
          default:
            eroare("cod atom eronat");
            break;
          }
      }
      else
      {
        // avem un operator dublu
        if ( atom.nume[0] == '+' ) return OP_INCREMENTARE;
        if ( atom.nume[0] == '-' ) return OP_DECREMENTARE;
        if ( atom.nume[0] == '<' )
          {
            if ( atom.nume[1] == '<'  ) return OP_DEPLASARE_ST;
            if ( atom.nume[1] == '='  ) return OP_MAI_MIC_EGAL;
          }
        if ( atom.nume[0] == '>' )
          {
            if ( atom.nume[1] == '>' ) return OP_DEPLASARE_DR;
            if ( atom.nume[1] == '=' ) return OP_MAI_MARE_EGAL;
          }
        if ( atom.nume[0] == '&' ) return OP_SI_LOGIC;
        if ( atom.nume[0] == '|' ) return OP_SAU_LOGIC;
        if ( atom.nume[0] == '=' ) return OP_EGALITATE;
        if ( atom.nume[0] == '!' ) return OP_DIFERIT;
      }
    }
  return COD_NUMAR;
}
```

parser.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "include/scaner.h"
#include "include/intrare.h"
#include "include/erori.h"
#include "include/parser.h"
#include "include/baza.h"
```

```
extern long numar_de_linii;
extern long marime_fisier;
extern long pozitie_in_fisier;
extern long numar_de_variabile;
extern ATOM atom;
extern ATOM atom_vechi;
extern char* sursa;
extern int sfarsit_de_fisier;
// definit in parser.h; sa fac alocare dinamica
extern variabila_t variabile[NUMAR_MAX_DE_VARIABILE];
arbore nod;  // arbore pentru rezolvarea expresiilor
long paranteze_deschise = 0;

void main_parse()
{
  get_atom();
  do
    {
      switch(atom.cod)
      {
      case COD_CITESTE:
        do_citeste();
        break;
      case COD_SCRIE:
        do_scrie();
        break;
      case COD_DACA:
        do_daca();
        break;
      case COD_PENTRU:
        do_pentru();
        break;
      case COD_REPETA:
        do_repeta();
        break;
      case COD_CTIMP:
        do_ctimp();
        break;
      case COD_SUB:
        do_subrutina();
        break;
      case COD_CARACTER:
        declarare_var(TIP_CARACTER);
        break;
      case COD_INTREG:
        declarare_var(TIP_INTREG);
        break;
      case COD_REAL:
        declarare_var(TIP_REAL);
        break;
      case '.':
        sfarsit_de_fisier = 1; // s-a ajuns la sfarsitul programului
        break;
      case COD_NUME: // atribuire (sau, cea mai incolo apel de functie)
        // daca e funcie => apel functie
        // daca nu => atribuire
```

```
        do_atribuire();
        break;
      default:
        if (sfarsit_de_fisier == 1)
          printf("\n s-a atins sfarsitul de fisier\n");
        else
          {
            printf("\n caracterul %li din %
li\n",pozitie_in_fisier,marime_fisier);
            eroare("caracter/identificator nepermis");
          }
         break;
      }
    }
  while ( sfarsit_de_fisier == FALSE );
}

int cauta_identificator(const char* nume_var )
{
  int i = numar_de_variabile;
  while ( i > 0 )
    {
      if ( strcmp(nume_var,variabile[i].nume) == 0 ) return i;
      i--;
    }
  return -1;
}

void adauga_identificator(const char* nume_var,int tip_var,float
val_var)
{
  int i = ++numar_de_variabile;
  if ( numar_de_variabile <= NUMAR_MAX_DE_VARIABILE )
    {
      strcpy(variabile [i].nume,nume_var);
      variabile [i].tip = tip_var;
      variabile [i].valoare_f = val_var;
    }
  else eroare("s-a atins numarul maxim de variabile");
}

void atribuie_identificator(const char* nume_var,float val_var)
{
  int i = cauta_identificator(nume_var);
  if ( i > 0 )
    // identificatorul exista
    variabile [i].valoare_f = val_var;
  else
    eroare("identificatorul nu a fost gasit");
}

void declarare_var(int tip_xxx)
{
  //atom.nume == "intreg" sau "caracter" sau "real"
  do
    {
```

```
      get_atom();
      if (atom.cod != COD_NUME)
      eroare("nume de identificator asteptat");
      if ( cauta_identificator(atom.nume) < 0 )
      // nu exista deci il putem adaugam
      adauga_identificator(atom.nume,tip_xxx,0.0);
      get_atom();

      switch (atom.cod)
      {
      case '=':
        get_atom();
        // asteptam o constanta sau o variabila deja definita
        if ( atom.cod == COD_NUMAR )
          {
            // daca e constanta facem atriguirea
            variabile[numar_de_variabile].valoare_f = atof(atom.nume);
          }
        else
          if ( atom.cod == COD_NUME )
            {
            // daca e nume de variabila cautam ...
            int indice;
            if ( (indice=cauta_identificator(atom.nume))>0)
              {
                // daca am gasit-o luam valoarea
                variabile[numar_de_variabile].valoare_f =
                  variabile[indice].valoare_f;
              }
            // daca nu iesim
            else eroare("variabila nedefinita");
            }
          else eroare("eroare de sintaxa,constanta asteptata");
        get_atom();
        if ( !(atom.cod == ',' || atom.cod == ';') )
          eroare ("',' sau ';' astepati");
        break;
      case ',':
      case ';':
        break;
      default:
        eroare ("eroare de sintaxa in declararea de variabile");
      }
    }
  while(atom.cod == ',');
  if ( atom.cod != ';' ) eroare ("lipseste ';'");
  get_atom();
}
void do_scrie()
{
  int indice = 0;
  do
    {
      get_atom();
      if (atom.cod == '"' )
      {
```

```
        while (curent_char() != '"' )
          {
            if ( curent_char() == '\\' )
            {
              switch (view_next_char())
                {
                case '\\':
                  next_char(); printf("\\");
                  break;
                case 'n':
                  next_char(); printf("\n");
                  break;
                case 't':
                  next_char(); printf("\t");
                  break;
                case 'r':
                  next_char(); printf("\r");
                  break;
                }
            }
            else printf("%c",curent_char());
            next_char();
          }
        next_char();// mancam apostroful de sfarsit
      }
      if (atom.cod == COD_NUME)
      {
        // trebuie sa afisez valoarea variabilei
        indice = cauta_identificator(atom.nume);
        if ( indice == 0)
            eroare("variabila nedefinita");
        else
          {
            switch (variabile[indice].tip)
            {
            case TIP_CARACTER:
              printf("%c",(char)variabile[indice].valoare_f);
              break;
            case TIP_INTREG:
              printf("%i",(int)variabile[indice].valoare_f);
              break;
            case TIP_REAL:
              printf("%f",(float)variabile[indice].valoare_f);
              break;
            default :
              eroare(" tip nedefinit (o prosrie)");
            }

          }
      }
      get_atom(); // trebuie sa manance virgula
    }
  while (atom.cod == ',' );
  if ( atom.cod != ';' ) eroare ("lipseste ';'");
  get_atom();
}
```

```
void do_citeste()
{
  int indice = -1;
  char var_caracter;
  int var_intreg;
  float var_real;

  do
    {
      get_atom();
      if ( atom.cod != COD_NUME )
        eroare("nume de identificator asteptat");
      else
      {
        indice = cauta_identificator(atom.nume);
        if ( indice > 0 )
          {
            // variabila exista
            switch (variabile[indice].tip)
            {
            case TIP_CARACTER:
              scanf("%c",&var_caracter);
              variabile[indice].valoare_f = var_caracter;
              break;
            case TIP_INTREG:
              scanf("%i",&var_intreg);
              variabile[indice].valoare_f = var_intreg;
              break;
            case TIP_REAL:
              scanf("%f",&var_real);
              variabile[indice].valoare_f = var_real;
              break;
            default:
              eroare ("tip nedefinit");
              break;
            }
          }
      }
      get_atom();
    }
  while (atom.cod == ',' );
  if (atom.cod != ';' )
    eroare("lipseste `;'");
  get_atom();
}

// *****************************************
//  incepe rutina pentru daca
//
// *****************************************
void do_daca(void)
{
  float valoare_conditie = 0;

  get_atom();
  if ( atom.cod != '(' ) eroare("lipseste `('");
```

```
  valoare_conditie = rezolva_exp();
  if ( atom.cod != ')' ) eroare("lipseste `)'");
  else get_atom();

  if ( valoare_conditie == 1 )
    {
      bloc_instructiuni();
      if ( atom.cod == COD_ALTFEL )
      {
        get_atom();
        sari_bloc();
      }
    }
  else
    {
      sari_bloc();
      if ( atom.cod == COD_ALTFEL )
      get_atom();
      bloc_instructiuni();
    }
  if ( atom.cod != COD_SFDACA )
    eroare("declaratie incorecta, lipseste `sfdaca'");
  get_atom();
}
void bloc_instructiuni()
{
  int sfarsit_bloc = 0;
  do
    {
      switch (atom.cod)
      {
      case COD_SCRIE:
        do_scrie();
        break;
      case COD_CITESTE:
        do_citeste();
        break;
      case COD_DACA:
        do_daca();
        break;
      case COD_PENTRU:
        do_pentru();
        break;
      case COD_REPETA:
        do_repeta();
        break;
      case COD_CTIMP:
        do_ctimp();
        break;
      case COD_SUB:
        do_subrutina();
        break;
      case COD_NUME:
        // vedem daca e functie / atribuire
        //deocamdata doar atribuire
        do_atribuire();
```

```
        break;
      default:
        // inseamna ca s-a atins un cod pentru care trebuie sa iesim
        sfarsit_bloc = 1;
        break;
      } // sf_switch
    }while ( ! sfarsit_bloc );
}
void sari_bloc()
{
  int sfarsit_bloc = 0 ;
   do
    {
      switch(atom.cod)
      {
      case COD_SCRIE:
        sari_scrie();
        break;
      case COD_CITESTE:
        sari_citeste();
        break;
      case COD_DACA:
        sari_daca();
        break;
      case COD_PENTRU:
        sari_pentru();
        break;
      case COD_REPETA:
        sari_repeta();
        break;
      case COD_CTIMP:
        sari_ctimp();
        break;
      case COD_SUB:
        sari_subrutina();
        break;
      case COD_NUME:
        // vedem daca e functie / atribuire
        //deocamdata doar atribuire
        sari_atribuire();
        break;
      default:
        // inseamna ca trebuie sa iesim
        sfarsit_bloc = 1;
        break;
      }
    } while ( !sfarsit_bloc );
}
void sari_scrie()
{
  int indice = 0;
  do
    {
      get_atom();
      if (atom.cod == '"' )
      {
```

```
        while (curent_char() != '"' )
          {
            if ( curent_char() == '\\' )
            {
              switch (view_next_char())
                {
                case '\\':
                  next_char();
                  break;
                case 'n':
                  next_char();
                  break;
                case 't':
                  next_char();
                  break;
                case 'r':
                  next_char();
                  break;
                }
            }
            next_char();
          }
        next_char();// mancam apostroful de sfarsit
      }
      if (atom.cod == COD_NUME)
      {
        // trebuie sa afisez valoarea variabilei
        indice = cauta_identificator(atom.nume);
        if ( indice == 0)
            eroare("variabila nedefinita");
        else
          {
            switch (variabile[indice].tip)
            {
            case TIP_CARACTER:
            case TIP_INTREG:
            case TIP_REAL:
              break;
            default :
              eroare(" tip nedefinit (o prosrie)");
              break;
            }
          }
      }
      get_atom(); // trebuie sa manance virgula
    }
  while (atom.cod == ',' );
  if ( atom.cod != ';' )
    eroare ("lipseste ';'");
  get_atom();
}
void sari_citeste()
{
  int indice = -1;
  do
    {
```

```
      get_atom();
      if ( atom.cod != COD_NUME )
        eroare("nume de identificator asteptat");
      else
      {
        indice = cauta_identificator(atom.nume);
        if ( indice > 0 )
          {
            // variabila exista
            switch (variabile[indice].tip)
            {
            case TIP_CARACTER:
            case TIP_INTREG:
            case TIP_REAL:
              break;
            default:
              eroare ("tip nedefinit");
              break;
            }
          }
      }
      get_atom();
    }
  while (atom.cod == ',' );
  if (atom.cod != ';' )
    eroare("lipseste `;'");
  get_atom();
}
void sari_atribuire()
{
  int indice ;
  float tmp;
  indice = cauta_identificator(atom.nume);

  if ( indice >0 )
    {
      get_atom();
      if ( atom.cod != '=' )
      eroare("lipseste operatorul de atribuire `='");
      else
      {
        get_atom();
        tmp = rezolva_exp();
        if ( atom.cod != ';' )
          eroare("lipseste `;'");
        get_atom();
      }
    }
  else
    eroare("variabila nedefinita");
}
void sari_daca()
{
  float valoare_conditie = 0;
  get_atom();
  if ( atom.cod != '(' ) eroare("lipseste `('");
```

```
  valoare_conditie = rezolva_exp();
  if ( atom.cod != ')' ) eroare("lipseste `)'");
  else get_atom();
  if ( valoare_conditie == 1 )
    {
      sari_bloc();
      if ( atom.cod == COD_ALTFEL )
      {
        get_atom();
        sari_bloc();
      }
    }
  else
    {
      sari_bloc();
      if ( atom.cod == COD_ALTFEL )
      {
        get_atom();
        sari_bloc();
      }
    }
  if ( atom.cod != COD_SFDACA )
    eroare("lipseste `sfdaca', declaratie incorecta");
  get_atom();
}
// *******************************************
//  incepe parserul de expresii
//
// *******************************************

float rezolva_exp()
{
  arbore *tree = NULL;
  if ( atom.cod == '(' )
    {
      paranteze_deschise = 1;
      get_atom();
    }
  if ( atom.cod == ';' || atom.cod == ')' )
      eroare("expresia nu exista");
  tree = expresie(0);
  if ( tree == NULL ) eroare("expresia nu exista");
  if ( atom.cod == ')' ) paranteze_deschise --;
  return rezultat(tree);
}
arbore *mknod(ATOM atm,arbore *t1,arbore *t2)
{
  arbore *t;
  t = (arbore *) malloc (sizeof (arbore) );
  t->tip = atm.cod;
  strcpy(t->nume,atm.nume);
  t->vecin_st = t1;
  t->vecin_dr = t2;
  return t;
}
```

```
arbore *mkfrunza(ATOM atm)
{
  arbore *t;
  int indice = 0 ;
  t = (arbore *) malloc (sizeof (arbore) );
  t->tip = atm.cod;
  strcpy(t->nume,atm.nume);
  if ( atm.cod == COD_NUMAR ) t->valoare = atoi(atm.nume);
  else
    if ( atm.cod == COD_NUME)
      {
      indice = cauta_identificator(atom.nume);
      if ( indice > 0 ) t->valoare = variabile[indice].valoare_f;
      }
  t->vecin_st = NULL;
  t->vecin_dr = NULL;
  return t;
}

int precedenta(int codu)
{
  // aflam precedenta operatorului
  switch (codu)
    {
    case OP_INCREMENTARE:
    case OP_DECREMENTARE:
    case '~':
    case '!':
      return NUMAR_NIVELE_PRECEDENTA - 4;
      break;
    case '*':
    case '/':
    case '%':
      return NUMAR_NIVELE_PRECEDENTA - 3;
      break;
    case '+':
    case '-':
      return NUMAR_NIVELE_PRECEDENTA - 4;
      break;
    case '<':
    case OP_MAI_MIC_EGAL:
    case '>':
    case OP_MAI_MARE_EGAL:
    case OP_EGALITATE:
    case OP_DIFERIT:
      return NUMAR_NIVELE_PRECEDENTA - 6;
      break;
    case '&':
      return NUMAR_NIVELE_PRECEDENTA - 7;
      break;
    case '^':
      return NUMAR_NIVELE_PRECEDENTA - 8;
      break;
    case '|':
      return NUMAR_NIVELE_PRECEDENTA - 9;
      break;
```

```
    case OP_SI_LOGIC:
      return NUMAR_NIVELE_PRECEDENTA - 10;
      break;
    case OP_SAU_LOGIC:
      return NUMAR_NIVELE_PRECEDENTA - 11;
      break;
      // trebuie sa modific nivelele 12 pt amandoua ?
    case '=':
      return NUMAR_NIVELE_PRECEDENTA - 12;
      break;
    case ',':
      return NUMAR_NIVELE_PRECEDENTA - 13;
      break;
    default:
      return -1;
      break;
    }
  return -1;
}
int e_op_binar(int codu)
{
  // aflam daca operatorul e binar
  switch (codu)
    {
    case '~':
    case '!':
    case OP_INCREMENTARE:
    case OP_DECREMENTARE:
      return OPERATOR_UNAR;
      break;
    case '+':
    case '-':
    case '*':
    case '/':
    case '%':
    case ',':
    case '|':
    case '&':
    case '^':
    case '=':
    case '<':
    case '>':
      return OPERATOR_BINAR; break;
    default: return OPERATOR_NECUNOSCUT; break;
    }
  return OPERATOR_NECUNOSCUT;
}

int asociativ(int codu)
{
  // aflam asociativitatea stnga/dreapta
  if (e_op_binar(codu) == OPERATOR_UNAR&& e_op_binar (atom_vechi.cod))
    return ASOCIATIV_LA_DREAPTA;
  else  return ASOCIATIV_LA_STANGA ;
}
```

```
arbore *expresie(int nivel_precedenta)
{
  arbore *tmp0 = NULL ,*tmp1 = NULL;
  ATOM atom_tmp;
  tmp0 = termen();
  if (atom.cod == ';' ) return tmp0;
  get_atom();
  if ( atom.cod == '(' ) paranteze_deschise++;
  else
    if ( atom.cod == ')' )
      {
      paranteze_deschise--;
      if (paranteze_deschise == 0 )  return tmp0;
      }
  while ( e_op_binar(atom.cod) && precedenta(atom.cod) >=
nivel_precedenta  )
    {
      strcpy(atom_tmp.nume,atom.nume);
      atom_tmp.cod = atom.cod;
      get_atom();
      if ( asociativ (atom_tmp.cod) == ASOCIATIV_LA_DREAPTA )
      tmp1 = expresie( precedenta (atom_tmp.cod) );
      else
      if (asociativ (atom_tmp.cod)  == ASOCIATIV_LA_STANGA )
        tmp1 = expresie( precedenta (atom_tmp.cod) + 1 );
      tmp0 = mknod( atom_tmp, tmp0, tmp1 );
    }
  return tmp0;
}

arbore *termen()
{
  arbore *tmp;
  ATOM atom_tmp;
  if (atom.cod == COD_NUME ) return mkfrunza(atom);
  else
    if ( atom.cod == COD_NUMAR ) return mkfrunza(atom);
    else
      if ( e_op_binar(atom.cod) == OPERATOR_UNAR )
      {
        strcpy(atom_tmp.nume,atom.nume);
        atom_tmp.cod = atom.cod ;
        get_atom();
        tmp = expresie(precedenta (atom_tmp.cod) );
        return mknod(atom_tmp,tmp,NULL);
      }
      else
      if ( atom.cod == '(' )
        {
          get_atom(); // mancam '('
          paranteze_deschise++;
          tmp = expresie(0);
          if ( atom.cod != ')' ) eroare("in expresie, lipseste ')'");
          else
            {
             paranteze_deschise --;
```

```
            return tmp;
            }
        }
  return NULL;
}

float rezultat(arbore *t)
{
  float valoare_st = 0,valoare_dr = 0 ;
  if ( t->vecin_st == NULL && t->vecin_dr == NULL ) return t->valoare;
  if ( t->vecin_st != NULL && t->vecin_dr == NULL )
    {
      valoare_st = t->vecin_st->valoare;
      switch (t->tip)
      {
      case OP_INCREMENTARE:
        valoare_st ++;
        break;
      case OP_DECREMENTARE:
        valoare_st --;
        break;
      case '!':
        valoare_st = ! ((int) valoare_st);
        break;
      case '~':
        valoare_dr = ~((int)valoare_st);
        break;
      }
      return valoare_st;
    }

  if ( t->vecin_st->tip == COD_NUMAR || t->vecin_st->tip == COD_NUME )
    valoare_st = t->vecin_st->valoare;
  else valoare_st = rezultat(t->vecin_st);
  if ( t->vecin_dr->tip == COD_NUMAR || t->vecin_dr->tip == COD_NUME )
    valoare_dr = t->vecin_dr->valoare;
  else valoare_dr = rezultat(t->vecin_dr);
  switch(t->tip)
    {
    case '+':
      return valoare_st + valoare_dr;
      break;
    case '-':
      return valoare_st - valoare_dr;
      break;
    case '*':
      return valoare_st * valoare_dr;
      break;
    case '/':
      if ( valoare_dr == 0 )
      eroare("diviziune cu 0 (/)");
      else return valoare_st / valoare_dr;
      break;
    case '%':
      if ( valoare_dr == 0 )
      eroare("diviziune cu 0 (%)");
```

```
      return (int)valoare_st % (int)valoare_dr;
      break;
    case '<':
      return valoare_st < valoare_dr;
      break;
    case '>':
      return valoare_st > valoare_dr;
      break;
    case OP_MAI_MIC_EGAL:
      return valoare_st <= valoare_dr;
      break;
    case OP_MAI_MARE_EGAL:
      return valoare_st >= valoare_dr;
      break;
    case OP_EGALITATE:
      return valoare_st == valoare_dr;
      break;
    case OP_DIFERIT:
      return valoare_st != valoare_dr;
      break;
    case ',':
      return valoare_st;
      break;
    case '&':
      return (int)valoare_st & (int)valoare_dr;
      break;
    case '|':
      return (int)valoare_st | (int)valoare_dr;
      break;
    case '^':
      return (int)valoare_st ^ (int)valoare_dr;
      break;
    case OP_SI_LOGIC:
      return valoare_st && valoare_dr;
      break;
    case OP_SAU_LOGIC:
      return valoare_st || valoare_dr;
      break;
    case '~':
      return ~((int)valoare_st);
      break;
    case '!':
      return !((int)valoare_st);
      break;
    default:
      eroare("operator invalid");
      break;
    }
  return valoare_st + valoare_dr;
}

void do_atribuire(void)
{
  int indice ;
  indice = cauta_identificator(atom.nume);
```

```
  if ( indice >0 )
    {
      get_atom();
      if ( atom.cod != '=' )
      eroare("lipseste `=', pentru atribuire");
      else
      {
        get_atom();
        variabile[indice].valoare_f = rezolva_exp();
        if ( atom.cod != ';' )
          eroare("lipseste `;'");
        get_atom();
      }
    }
  else
    eroare("variabila nedefinita");
}

void do_repeta(void)
{
  float valoare_conditie = 1;
  long pozitie_inceput,numar_linii;
  ATOM atom_tmp;
  get_atom(); // mancam `repeta'
  pozitie_inceput = pozitie_in_fisier;
  atom_tmp.cod = atom.cod;
  strcpy(atom_tmp.nume,atom.nume);
  numar_linii = numar_de_linii;
  do
    {
      strcpy(atom.nume,atom_tmp.nume);
      atom.cod = atom_tmp.cod;
      pozitie_in_fisier = pozitie_inceput;
      numar_de_linii = numar_linii;
      bloc_instructiuni();
      if (atom.cod != COD_PANA ) eroare("lipseste `pana'");
      get_atom();
      if ( atom.cod != '(' ) eroare ("lipseste `('");
      get_atom();
      valoare_conditie = rezolva_exp();
      if ( atom.cod != ')' ) eroare("lipseste `)'");
      get_atom();
    }
  while (!valoare_conditie );
}

void sari_repeta(void)
{
  float valoare_conditie;
  get_atom(); // mancam `repeta'
  sari_bloc();
  if (atom.cod != COD_PANA ) eroare("lipseste `pana'");
  get_atom();
  if ( atom.cod != '(' ) eroare ("lipseste `('");
  get_atom();
  valoare_conditie = rezolva_exp();
```

```
  if ( atom.cod != ')' ) eroare("lipseste `)'");
  get_atom();
}
void do_ctimp(void)
{
  float valoare_conditie;
  long numar_linii,pozitie_inceput;
  ATOM atom_tmp;
  get_atom();  // mancam `ctimp'
  numar_linii = numar_de_linii;
  pozitie_inceput = pozitie_in_fisier;
  atom_tmp.cod = atom.cod;
  strcpy(atom_tmp.nume,atom.nume);
  if (atom.cod != '(' ) eroare ("lipseste `('");
  get_atom();
  valoare_conditie = rezolva_exp();
  if ( atom.cod != ')' ) eroare("lipseste `)'");
  get_atom();
  do
    {
      if ( valoare_conditie != 0 )
      {
        bloc_instructiuni();
        if ( atom.cod != COD_SFCTIMP ) eroare("lipseste `sfctimp'");
        numar_de_linii = numar_linii;
        pozitie_in_fisier = pozitie_inceput;
        strcpy(atom.nume,atom_tmp.nume);
        atom.cod = atom_tmp.cod;
        get_atom();
        valoare_conditie = rezolva_exp();
        get_atom();
      }
      else sari_bloc();
    }
  while (valoare_conditie != 0 );
  sari_bloc();
  if ( atom.cod != COD_SFCTIMP ) eroare("lipseste `sfctimp'");
  get_atom();
}
void sari_ctimp(void)
{
  float valoare_conditie;
  get_atom();  // mancam `ctimp'
  if (atom.cod != '(' ) eroare ("lipseste `('");
  get_atom();
  valoare_conditie = rezolva_exp();
  if ( atom.cod != ')' ) eroare("lipseste `)'");
  get_atom();
  sari_bloc();
  if ( atom.cod != COD_SFCTIMP ) eroare("lipseste `sfctimp'");
  get_atom();  //mancam sfctimp
}
void do_pentru(void)
{
  float valoare_conditie,tmp;
  long numar_linii,pozitie_inceput;
```

```
  int indice_contor,indice2;
  ATOM atom_tmp;
  get_atom();  // mancam `pentru'
  if (atom.cod != '(' ) eroare ("lipseste `('");
  get_atom(); // asta trebuie sa fie numele contorului
  indice_contor = cauta_identificator(atom.nume);
  if ( indice_contor <= 0 ) eroare("variabila nedefinita");
  get_atom();
  if ( atom.cod == '=' ) et_atom();  // mancam `='
  else if (atom.cod != ';' )
      eroare("initializarea contorului incorecta");
  variabile[indice_contor].valoare_f = rezolva_exp();
  if ( atom.cod != ';' ) eroare("lipseste `;'");
  get_atom();
  // salvam pozitia si variabilele
  numar_linii = numar_de_linii;
  pozitie_inceput = pozitie_in_fisier;
  atom_tmp.cod = atom.cod;
  strcpy(atom_tmp.nume,atom.nume);
  valoare_conditie = rezolva_exp();
  if ( atom.cod != ';' ) eroare("lipseste `;'");
  get_atom();  // urmeaza incrementarea contorului
  indice2 = cauta_identificator(atom.nume);
  if ( indice2 != indice_contor ) eroare("lipseste variabila contor");
  get_atom();
  if ( atom.cod != '=' ) eroare("lipseste `=', atribuire asteptata");
  get_atom();
  tmp = rezolva_exp();
  if ( atom.cod != ')' ) eroare("lipseste `)'");
  get_atom();
  do
    {
      if ( valoare_conditie != 0 )
      {
        bloc_instructiuni();
        if ( atom.cod != COD_SFPENTRU ) eroare("lipseste `sfpentru'");
        numar_de_linii = numar_linii;
        pozitie_in_fisier = pozitie_inceput;
        strcpy(atom.nume,atom_tmp.nume);
        atom.cod = atom_tmp.cod;
        valoare_conditie = rezolva_exp();
        if ( atom.cod != ';' ) eroare("lipseste `;'");
        get_atom();  // urmeaza incrementarea contorului
        indice2 = cauta_identificator(atom.nume);
        if ( indice2 != indice_contor )
             eroare("lipseste variabila contor");
        get_atom();
        if ( atom.cod != '=' )
            eroare("lipseste `=', atribuire asteptata");
        get_atom();
        variabile[indice_contor].valoare_f = rezolva_exp();
        if ( atom.cod != ')' ) eroare("lipseste `)'");
        get_atom();
      }
      else sari_bloc();
    }
```

```
  while (valoare_conditie != 0 );
  sari_bloc();
  if ( atom.cod != COD_SFPENTRU ) eroare("`sfpentru' asteptat");
  get_atom();
}
void sari_pentru()
{
  float valoare_conditie;
  int indice_contor,indice2;
  get_atom();  // mancam `pentru'
  if (atom.cod != '(' ) eroare ("lipseste `('");
  get_atom(); // asta trebuie sa fie numele contorului
  indice_contor = cauta_identificator(atom.nume);
  if ( indice_contor <= 0 ) eroare("variabila nedefinita");
  get_atom();
  if ( atom.cod == '=' ) get_atom();
  else if (atom.cod != ';' )
      eroare("initializarea contorului incorecta");
  valoare_conditie = rezolva_exp();
  if ( atom.cod != ';' ) eroare("lipseste `;'");
  get_atom();
  valoare_conditie = rezolva_exp();
  if ( atom.cod != ';' ) eroare("lipseste `;'");
  get_atom();  // urmeaza incrementarea contorului
  indice2 = cauta_identificator(atom.nume);
  if ( indice2 != indice_contor ) eroare("lipseste variabila contor");
  get_atom();
  if ( atom.cod != '=' ) eroare("lipseste `=', atribuire asteptata");
  get_atom();
  valoare_conditie = rezolva_exp();
  if ( atom.cod != ')' ) eroare("lipseste `)'");
  get_atom();
  sari_bloc();  // sarim peste instructiuni
  if ( atom.cod != COD_SFPENTRU ) eroare("`sfpentru' asteptat");
  get_atom();
}
void do_subrutina(void)
{
  // ramane de implementat
  eroare ("din pacate nu am avut timp sa implementez subrutinele");
}
void sari_subrutina(void)
{
  // si asta ramane de implementat
  eroare ("din pacate nu am avut timp sa implementez subrutinele");
}
```

Bibliografie :

- Jack Crenshaw - „ Let's build a compiler”
- Michelle Donalie - „Writing a compiler”
- Irina Athanasiu - Limbaje formale si translatoare (note de curs)