Departamentul de Calculatoare și Tehnologia Informației
Universitatea Tehnica din Cluj-Napoca

Limbaje formale si translatoare

Proiect

David Rebeca
Modrea Elena
Grupa: 30233

Cuprins

Capitolul 1

Introducere

Acest proiect propune dezvoltarea unui mini limbaj de programare utilizând Yacc (Yet Another Compiler Compiler) și Lex (Lexical Analyzer Generator). Mini limbajul de programare va permite utilizatorilor să scrie programe simple și să le execute folosind un interpretor specific limbajului. Yacc și Lex sunt două instrumente puternice și populare în domeniul compilatoarelor și al limbajelor de programare, care facilitează generarea analizorului sintactic și a analizorului lexical pentru un limbaj definit de utilizator.

Analizorul lexical (lexer) are rolul de a împărți programul într-o secvență de token-uri (unități lexicale) care vor fi ulterior procesate de analizorul sintactic. În această etapă, se vor defini regulile pentru identificarea și clasificarea token-urilor din codul sursă.

Analizorul sintactic (parser) va analiza secvența de token-uri obținută de la analizorul lexical și va verifica conformitatea acesteia cu regulile gramaticale definite pentru limbajul de programare.

Capitolul 2

Descrierea proiectului

Mini limbajul nostru de programare are o sintaxa asemanatoare cu limbajul de programare C. Am definit pentru acesta 4 tipuri de date, 2 tipuri de instructiuni decizionale, 3 tipuri de instructiuni repetitive si mai multe functii pentru 2 tipuri de date. Utlizatorul are la dispozitie 26 de variabile (o singura litera mica) pe care le poate declara de un anumit tip.

2.1 Tipuri de date

Tipurile de date pe care utilizatorul le poate folosi sunt urmatoarele:

- Tipul int care reprezinta numerele intregi
- Tipul float care reprezinta numerele reale
- Tipul char* care reprezinta sirurile de caractere
- Tipul array care reprezina vectorii de numere intregi

Toate variabilele trebuie sa fie declarate ca fiind de unul din tipurile de date de mai sus pentru a se folosi asignarea de valoarea. De asemenea, daca o variabila a fost declarata ca fiind de un anumit tip, utilizatorul nu poate reinstantia variabila respectiva cu alt tip.

Pentru a afisa variabilele de orice tip, se poate folosi instructiunea print.

2.2 Operatii pe numere intregi si reale

Operatiile care se pot folosi pe variabile de tip int si float sunt urmatoarele:

- Asignare folosind operatorul "="
- Adunare folosind operatorul "+"
- Scadere folosind operatorul "-"
- Inmultire folosind operatorul "*"
- Impartire folosind operatorul "/"

De asemenea, se pot realiza comparari folosind urmatorii operatori:

- "<" - returneaza 1 daca primul operand este mai mic comparativ cu al doilea
- ">" - returneaza 1 daca primul operand este mai mare comparativ cu al doilea
- "<=" - returneaza 1 daca primul operand este mai mic sau egal comparativ cu al doilea
- ">=" - returneaza 1 daca primul operand este mai mare sau egal comparativ cu al doilea
- "==" - returneaza 1 daca operanzii sunt egali
- "!=" - returneaza 1 daca operanzii sunt diferiti

2.3 Instructiuni decizionale

Instructiuniile decizionale suporatate de limbajul nostru de programare sunt urmatoarele:

- Instructiunea if - care poate fi scrisa cu ramura else si care poate contine si alte instructiuni, inclusiv altele de tip if
- Instructiunea switch - care poate avea cat de multe clauze Case doreste utilizatorul (variabila dupa care se poate face switch-ul putand fi de tipul int, float si char*) si poate contine si alte instructiuni in interiorul Case.

2.4 Instructiuni repetitive

Instructiuniile repetitive suporatate de limbajul nostru de programare sunt urmatoarele:

- Instructiunea While
- Instructiunea Do...While
- Instructiunea For

Toate instructiunile repetive pot contine alte instructiuni, inclusv alte instructiuni repetitive, astfel putandu-se scrie programe complexe.

2.5 Functii pe vectori de intregi

Am implementat o serie de functii care se pot apela direct pe vectori de intregi, in anumite cazuri chiar salvandu-se rezultatul in variabila cu care s-a realizat apelul. Functiile care se pot apela pe vectorii de intregi sunt urmatoarele:

- Functia **sort**- Sorteza vectorul dat ca parametru
- Functia **length** - Afiseaza lungimea vectorului
- Functia **search** - Cauta in vectorul dat ca primul parametru intregul dat ca al doilea parametru. Daca intregul este gasit, se afiseaza indexul primei aparitii, in caz contrar, se afiseaza un mesaj.
- Functia **insertFirst** - Insereaza pe prima pozitie in vectorul dat ca primul parametru intregul dat ca al doilea parametru.
- Functia **insertLast** - Insereaza pe ultima pozitie in vectorul dat ca primul parametru intregul dat ca al doilea parametru.
- Functia **delete** - Sterge din vectorul dat ca primul parametru intregul dat ca al doilea parametru. In caz ca intregul nu exista in vector, se va afisa un mesaj.

- Functia **max** - Afiseaza maximul din vector.
- Functia **min** - Afiseaza minimul din vector.

2.6 Functii pe siruri de caractere

Am implementat si pentru siruri de caractere urmatoarele functii pe care utilizatorul le poate apela direct pe variabilele corespunzatoare (doar daca sunt declarate de tipul char*):

- Functia **copy**- Copiaza al doilea sir de caractere (dat ca parametru) in primul.
- Functia **compare** - Compara cele doua siruri de caractere date ca parametru
- Functia **contains** - Cauta cel de-al doilea sir de caractere dat ca parametru in primul sir dat ca parametru
- Functia **toLower** - Schimba toare caracterele din sirul dat ca parametru in litere mici.
- Functia **toUpper** - Schimba toare caracterele din sirul dat ca parametru in litere mari.
- Functia **deleteAll** - Sterge din primul sir toate aparitiile celui de al doilea sir de caractere.
- Functia **append** - Concateneaza cele doua siruri de caractere si pune rezulatul in primul.

Capitolul 3

Decizii de implementare

Proiectul a fost realizat folosind Lex, Yacc si C. Lex reprezinta etapa în care codul sursă este analizat și împărțit într-o secvență de token-uri care urmeaza sa fie preluata de Yacc. Dupa preluare, Yacc genereaza un parser in C. Parserul generat de Yacc analizează codul de intrare conform regulilor de gramatică specificate, identificând elementele sintactice ale limbajului și generând un arbore de parsare.

In Figura 3.1 este o portiune din fisierul lex care are rolul de a recunoaste un vector de intregi de forma {1,2,3,4}. Inputul este preluat, se elimina acoladele si fiecare intreg despartit prin virgula este adaugat intr-un vector de intregi, transmis mai departe fisierlui yacc.

```
[\{]([0-9]+[\,]?)+[\}] {
                       char*s= yytext+1;
                       s[strlen(s)-1]='\0';
                       char*p=strtok(s,",");
                       int* rez=calloc(100,sizeof(int));
                       int k=0;
                       while(p){
                        rez[k]=atoi(p);
                        k++;
                        p=strtok(NULL,",");
                       }
                       yylval.aValue = calloc(100, sizeof(int));
                       yylval.aValue = rez;
                       return ARRAYV;
                       }
```

Figura 3.1: Recunoasterea unui array in Lex

In Yacc, regulile sunt utilizate pentru a defini gramatica limbajului creat de noi. In Figura 3.2 sunt specificate regulile ce stabilesc sintaxa si strctura pentru SWITCH. Sintaxa este urmatoarea:
SWITCH (variable) {
case 1: STATEMENTS;break;
...

default: STATEMENTS;break;

}

```
          | SWITCH  '(' VARIABLE ')' '{' case_statements default_stmt '}'
                    { $$ = opr(SWITCH, 3, id($3), $6, $7);
                    }
          | IF '(' expr ')' statement %prec IFX
                    { $$ = opr(IF, 2, $3, $5); }
          | IF '(' expr ')' statement ELSE statement
                    { $$ = opr(IF, 3, $3, $5, $7); }
          | '{' stmt_list '}' { $$ = $2; }
          ;

stmt_list : statement
          | stmt_list statement   { $$ = opr(';', 2, $1, $2); }
          ;
case_statements : case_statement
                | case_statements case_statement  { $$ = opr(';', 2, $1, $2); }
                ;

case_statement : CASE case_expr ':' stmt_list BREAK ';'  {$$= opr(CASE,2,$2,$4);}

default_stmt : DEFAULT ':' stmt_list BREAK ';' {$$=opr(DEFAULT,1,$3);}
             ;
assignment_stmt : VARIABLE '=' expr { $$ = opr('=', 2, id($1), $3); }
                |
                ;
```

Figura 3.2: Switch statement din Yacc

In Figura 3.3 este prezentata o portiune din functia ex care se afla in fisierul interpreter.y (yacc), si anume portiunea care recunoaste nodul SWITCH si executa in C aceasta instructiune.

```
    case SWITCH:  int ok=0;
                  for(int i=0 ; i<p->opr.op[1]->opr.nops;i++){
                    if(ex(p->opr.op[0]).iVal!=999999){
                     if(ex(p->opr.op[0]).iVal==ex(p->opr.op[1]->opr.op[i]->opr.op[0]).iVal){
                       ex(p->opr.op[1]->opr.op[i]->opr.op[1]);
                       ok=1;
                     }}
                     else if(ex(p->opr.op[0]).fVal!=999999.0) {
                            if(ex(p->opr.op[0]).fVal==ex(p->opr.op[1]->opr.op[i]->opr.op[0]).fVal){
                               ex(p->opr.op[1]->opr.op[i]->opr.op[1]);
                               ok=1;
                     }
                     }
                     else {
                      if(strcmp(ex(p->opr.op[0]).cVal,ex(p->opr.op[1]->opr.op[i]->opr.op[0]).cVal)==0){
                               ex(p->opr.op[1]->opr.op[i]->opr.op[1]);
                               ok=1;
                      }
                    }
                  }
                  if(ok==0){
                    ex(p->opr.op[2]->opr.op[0]);
                  }
                   return type;
```

Figura 3.3: Switch in functia ex din Yacc

Capitolul 4

Exemple de rulare

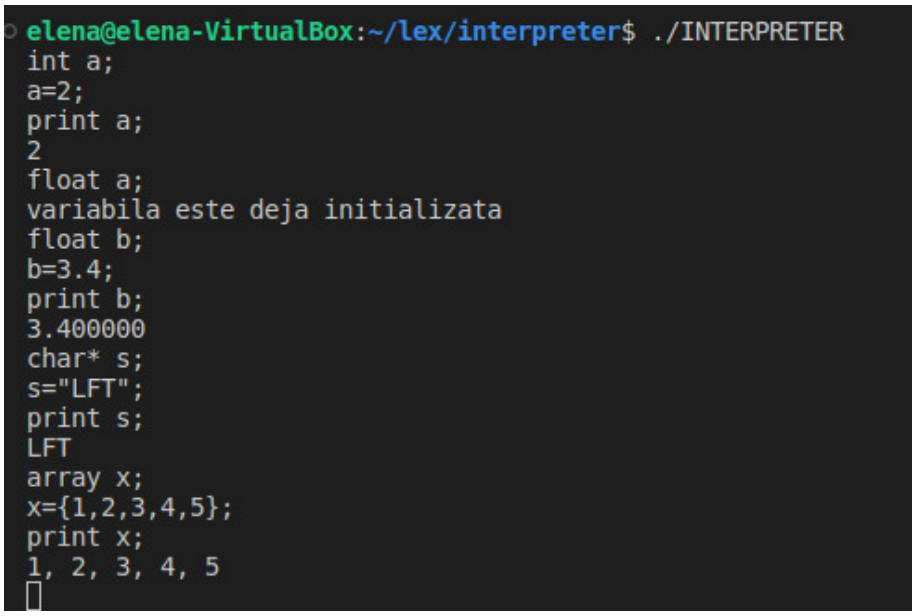

Figura 4.1: Declararea variabilelor si asignarea valorilor

```
int a;
a=2;
int b;
b=3;
int c;
c=2;
int d;
d=a<b;
print d;
1
d=a>b;
print d;
0
d=a==b;
print d;
0
d=a!=b;
print d;
1
d=a<=b;
print d;
1
d=a>=b;
print d;
0
d=a==c;
print d;
1
float e;
e=3.5;
float f;
f=4.6;
d=e<f;
print d;
1
d=a<f;
operanzii nu sunt de acelasi tip
```

Figura 4.2: Folosirea operatoriilor de comparare

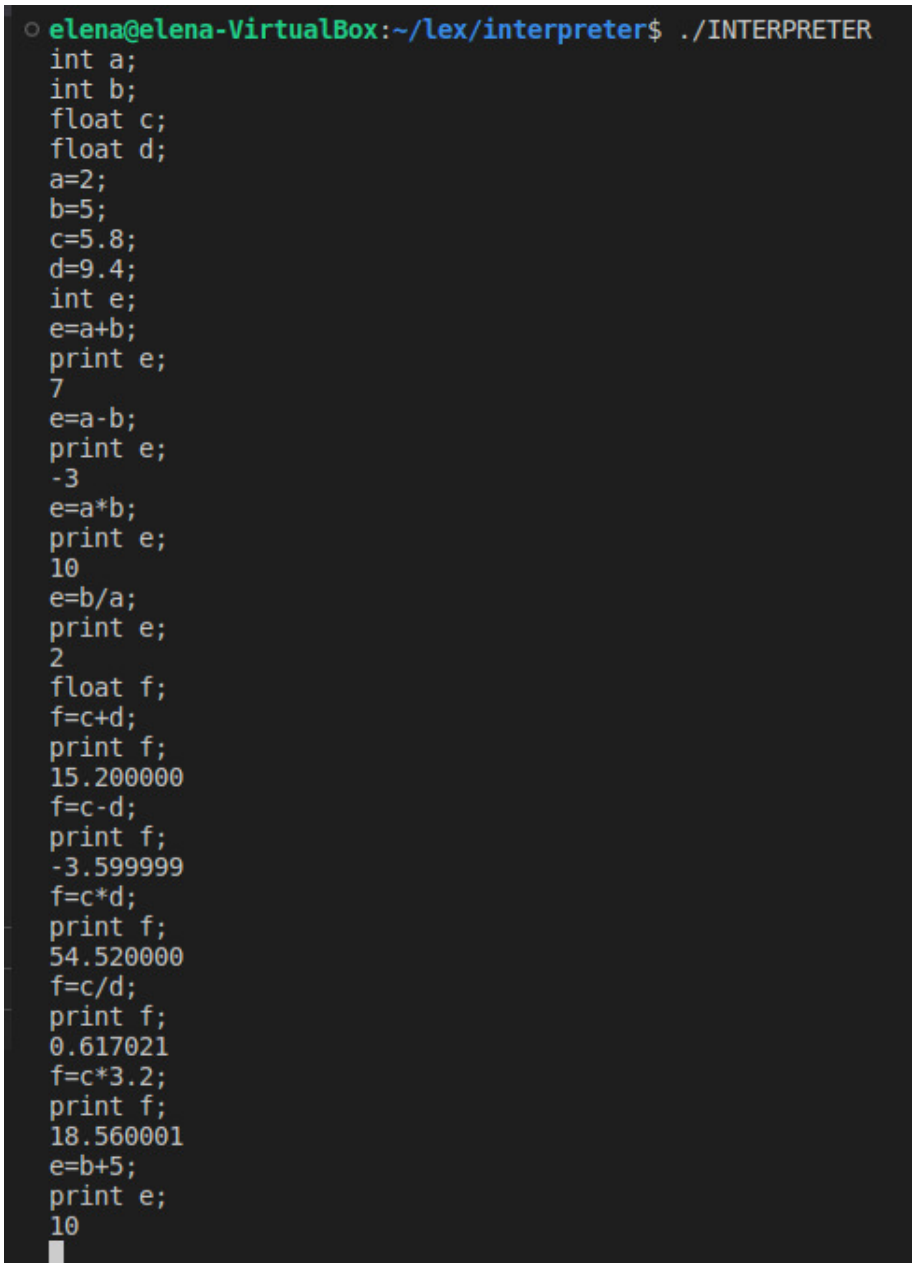

Figura 4.3: Folosirea operatiilor aritmetice pe numere intregi si reale

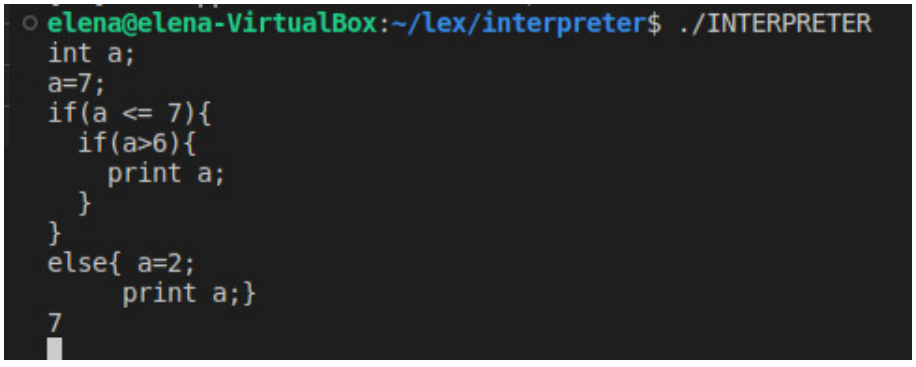

Figura 4.4: Instructiunea if

```
elena@elena-VirtualBox:~/lex/interpreter$ ./INTERPRETER
int a;
a=7;
switch(a){
 case 1: print 1; break;
 case 2: print 2; break;
 case 7: a=3; print a; break;
 default: print 4; break;
}
3
switch(a){
 case 1: print 1; break;
 case 2: print 2; break;
 case 7: a=3; print a; break;
 default: print 4; break;
}
4
```

Figura 4.5: Instructiunea switch

```
elena@elena-VirtualBox:~/lex/interpreter$ ./INTERPRETER
int a;
a=0;
int b;
b=23;
int c;
c=45;
while(a<10){
 b=b+1;
 while(c>30){
   c=c+-2;
 }
a=a+1;
}
print a;
10
print b;
33
print c;
29
```

Figura 4.6: Instructiunea while

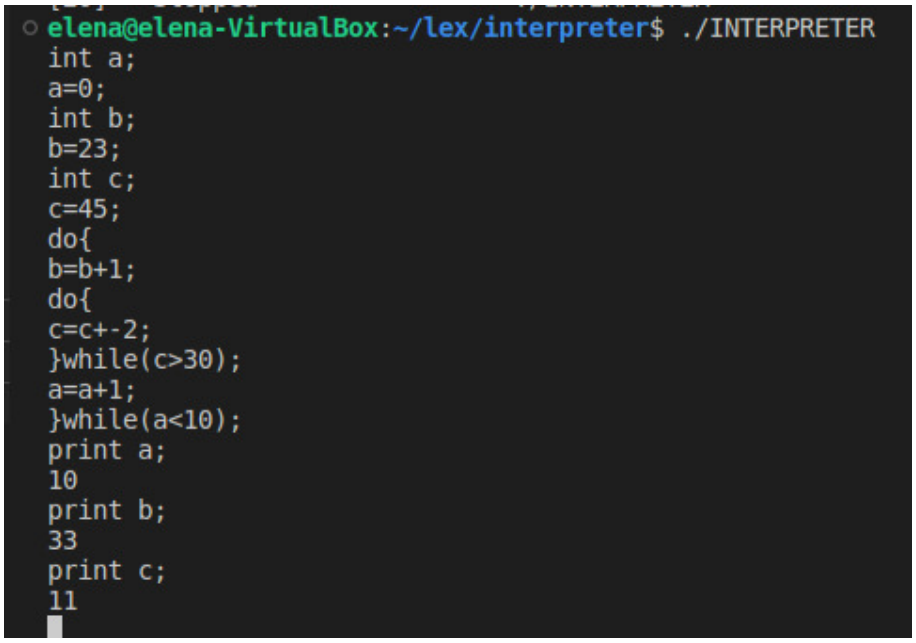

Figura 4.7: Instructiunea do...while

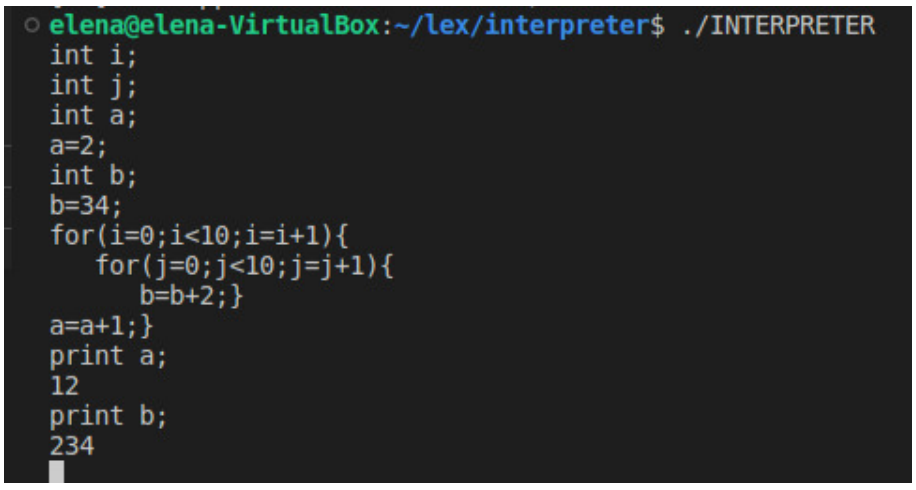

Figura 4.8: Instructiunea for

```
elena@elena-VirtualBox:~/lex/interpreter$ ./INTERPRETER
array a;
a={4,1,7,2,5};
sort(a);
print a;
1, 2, 4, 5, 7
length(a);
lungimea array-ului este: 5
search(a,4);
indexul unde se afla elementul cautat este: 2
insertFirst(a,11);
print a;
11, 1, 2, 4, 5, 7
insertLast(a,3);
print a;
11, 1, 2, 4, 5, 7, 3
delete(a,11);
print a;
1, 2, 4, 5, 7, 3
delete(a,11);
elementul nu a fost gasit
max(a);
maximul este: 7
min(a);
minimul este: 1
int b;
b=2;
delete(b,2);
nu se poate face delete decat pe array
```

Figura 4.9: Functii pe vectori de intregi

```
elena@elena-VirtualBox:~/lex/interpreter$ ./INTERPRETER
char*s;
s="Rebeca";
compare(s,"Elena");
primul sir este mai mare decat al doilea
compare(s,"Rebeca");
sirurile sunt egale
copy(s,"Elena");
print s;
Elena
append(s," si Rebeca");
print s;
Elena si Rebeca
toUpper(s);
print s;
ELENA SI REBECA
toLower(s);
print s;
elena si rebeca
deleteAll(s,"e");
print s;
lna si rbca
```

Figura 4.10: Functii pe siruri de caractere

Capitolul 5

Link catre cod

Codul proiectului poate fi regasit la:
https://github.com/elenamodrea/ProiectLFT.git.