

# Limbaje Formale si Translatoare

## Proiect

Eusebiu-Ioan Florean  
Răzvan-Vasile Bumbu

May 2023

## Contents

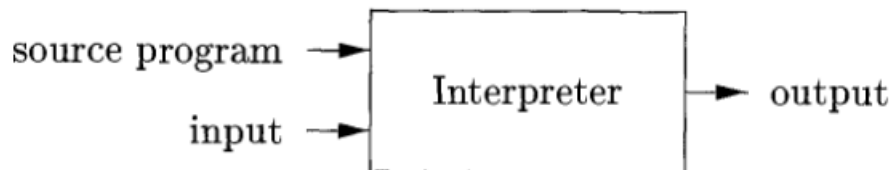
1	Introducere	3
2	Descrierea proiectului	3
3	Decizii de implementare	6
4	Exemple de rulare	9
5	Link proiect	12

## 1 Introducere

În cadrul acestui proiect am realizat un mini-interpreter pentru un limbaj de programare propriu, care este capabil de asemenea să execute și să afișeze rezultatul codului introdus ca și input.

Interpreterul este în tehnică software (de programare), un program special prin care se citește, se analizează și se execută (interpretează pe loc) comenzile și instrucțiunile dintr-un program sursă - și anume pas cu pas, una după alta, fără o compilare anterioară. Eventualele erori de programare din programul-sursă devin evidente abia în momentul când interpreterul încearcă să-l execute și evident se blochează.

Interpretarea unui program sursă durează mai mult decât executarea programului (același) compilat. Aceasta se explică prin aceea că instrucțiunile unui program compilat sunt direct executate, în timp ce interpreterul citește și analizează mai întâi instrucțiunile, după care le poate executa.



*Cun funcționează un interpreter*

## 2 Descrierea proiectului

Limbajul propriu de programare are la bază 3 tipuri de variabile regăsite în majoritatea limbajelor de programare imperative:

- Numere întregi - **int**
- Numere reale - **float**
- Siruri de caractere - **string**

De asemenea, pe langa variabile, limbajul contine si o serie de instructiuni:

- Instructiuni de selectie
  - **if**
  - **switch**
- Instructiuni de ciclare
  - **while**
  - **do while (repeat until)**
  - **for**
- Instructiuni de operatii aritmetice pe numere intregi si numere reale
  - **+**
  - **-**
  - **\***
  - **/**
  - **=**
- Instructiuni de operatii logice pe numere intregi si numere reale
  - **<**
  - **>**
  - **<=**
  - **>=**
  - **==**
  - **!=**
- Instructiune de afisare output
  - **print**

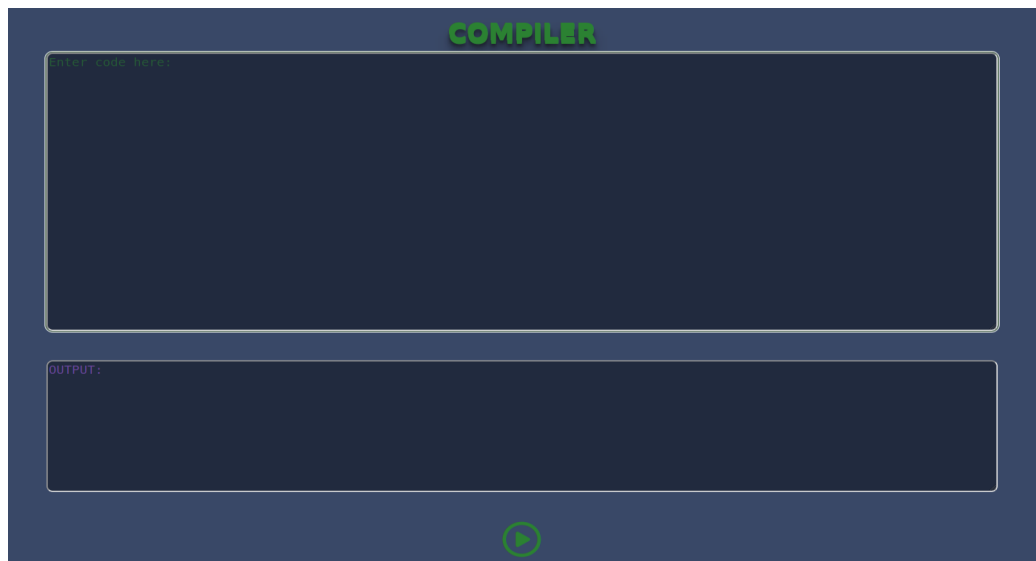
In plus, am decis sa implementam in cadrul proiectului doua functionalitati diferite fata de limbajele de programare clasice.

O prima functionalitate este aceea a introducerii unei instructiuni care permite utilizatorului sa joace jocul Tic Tac Toe (X si O).

Cea de-a doua functionalitate adaugata, este una gandita pentru utilizarea interpretorului, si anume interfata web in care utilizatorul poate introduce codul sursa, dupa care, prin apasarea butonului pentru rulare, acesta va primi output-ul programului.

## OBSERVATII

- Variabilele pot fi declarate doar utilizand literele mici ale alfabetului englez (**a..z**)
- Declararea si instantierea variabilelor se va face exclusiv de forma:  
**tip nume;**  
**nume = valoare;**
- Instantierea numerelor reale se va face folosind simbolul **punct(.)** intre partea intreaga si partea fractionara.
- Instantierea sirurilor de caractere se va face folosind simbolurile " " .
- Sirurile de caractere nu trebuie sa contina o singura litera sau cuvinte rezervate pentru nume de variabile sau instructiuni si de asemenea nu pot sa contina spatii.
- Instructiunea **print** poate afisa doar valoarea unei singure variabile, fara alte mesaje.
- Instructiunea **if** trebuie sa contina cel mult o ramura **else**.
- Instructiunea **switch** trebuie sa contina doua ramuri **case** si o ramura **default**.
- Finalul codului se va marca prin simbolul **punct(.)**.
- Exemple pentru folosirea corecta a instructiunilor poate fi vazuta in sectiunea **4. Exemple de rulare**.



*Interfata grafica a aplicatiei*

### 3 Decizii de implementare

Acest proiect a fost dezvoltat cu ajutorul mai multor limbaje de programare:

- **lex**
- **yacc**
- **C**
- **nodemon**
- **HTML/CSS/JAVA SCRIPT**

Prin lex are loc parsarea textului(codul sursa) spre yacc, generand tokeni prin expresiile regulate definite in fisierul "interpreter.l". Un exemplu de parsare, reprezentat prin imaginea de mai jos, este recunoasterea comenzilor de tipul "put on x y". In momentul in care va fi introdusa de catre utilizator o comanda analog acesteia, parserul o va valida, fiind vorba despre o comanda din jocul "tic tac toe", va seta in yylval.x pozitia liniei, in yylval.y pozitia coloanei dupa care va returna tokenul PUT care va fi interpretat ulterior de catre yacc.

```

"put on "[1-3][ ][1-3]
{
    char*s = yytext;
    yyval.ticValue.x = s[7] - '0';
    yyval.ticValue.y = s[9] - '0';
    return PUT;
}

```

*parser.l code-example*

In cadrul fisierului **interpreter.y** sunt definite functiile in limbajul C care vor fi apelate in momentul validarii tokenilor. Semnatura tuturor acestor functii, impreuna cu declaratiile de noi tipuri sunt definite in fisierul "types.h" care este importat pe urma in yacc.

Acest fisier contine marea majoritate a logicii, care consta in construirea arborelui de parsare, adaugandu-se cate un nou nod cu fiecare nou token parsat. Cand scanarea fisierului s-a terminat, adica exista un arbore complet, va fi apelata functia `execute()` care parcurge intregul arbore evaluand valoarea fiecarui nod.

Pentru printarea valorii unei variabile prin comanda "**print**" se acceseaza memoria in cadrul careia se pastreaza pentru fiecare variabila de la 'a' la 'z' tipul si valoarea. Se verifica **tipul** variabilei, dupa care are loc afisarea **valorii** acesteia prin utilizarea variabile aux de tipul data care reprezinta valoarea functiei apelate recursiv.

```

case PRINT :
    aux = execute(node->opr.op[0]);
    if (aux.intValue != INT_MAX) {
        printf("%d\n", execute(node->opr.op[0]).intValue);
        return data;
    }
    if (aux.floatValue != INT_MAX) {
        printf("%f\n", execute(node->opr.op[0]).floatValue);
        return data;
    }
    if (aux.stringValue != NULL) {
        printf("%s\n", execute(node->opr.op[0]).stringValue);
        return data;
    }
    return data;

```

*print code-example*

Pentru recunoasterea structurilor de tip **if** are loc crearea unui nod in 2 cazuri:

- **if-then:** nod cu 2 copii: expresia si propozitia(then)
- **if-then-else:** nod cu 3 copii: expresia si cele 2 propozitii (then si else)

O data creat acest nod, va fi recunoscut in functia `execute` unde se va distinge daca este un **if** cu sau fara ramura **else** in functie de

numarul de copii ai nodului radacina, apelandu-se pentru fiecare caz codul C corespunzator.

```

| IF '(' expr ')' statement %prec IFX      { $$ = opr(IF, 2, $3, $5); }
| IF '(' expr ')' statement ELSE statement { $$ = opr(IF, 3, $3, $5, $7); }
case IF :
    if (node->opr.op[0]->opr.oper == '<' || node->opr.op[0]->opr.oper == '>' ||
        node->opr.op[0]->opr.oper == EQ || node->opr.op[0]->opr.oper == NE ||
        node->opr.op[0]->opr.oper == GE || node->opr.op[0]->opr.oper == LE) {
        if (execute(node->opr.op[0]).intValue)
            execute(node->opr.op[1]);
        else if (node->opr.nops > 2)
            execute(node->opr.op[2]);
        }
    return data;

```

*if-then-else code-example*

Scheletul interfetei grafice a fost realizat in **HTML**, stilizat cu ajutorul **CSS**, iar pentru dinamicitatea elementelor grafice s-a folosit **Bootstrap**.

Partea de client a aplicatiei care se ocupa cu functionalitatii paginii web este dezvoltata in **Java Script**, care utilizeaza **web-sockets** pentru comunicarea cu server-ul **nodemon**. In momentul in care utilizatorul actioneaza butonul de rulare al programului, este preluat textul introdus de acesta in casuta de input si este transmis spre server(**I**). Acesta citeste mesajul reprezentand programul scris de utilizator, il executa cu ajutorul fisierul yacc si lex, dupa care trimite ca raspuns clientului output-ul (**II**), fiind preluat de catre client care va actualiza interfata grafica (**III**).

```

runButton.addEventListener('click', function(event) {
I   websocket.send(inputArea.value);
});

ws.on('message', (data) => {
    var content = data.toString() + "\n" + "." + "\n";
    try {
        const data = fs.writeFileSync('file.in', content)
        //file written successfully
    } catch (err) {
        console.error(err);
    }
    var exec = require('child_process').exec;

    exec('make -C ../lex_yacc', function (error, stdout, stderr) {
        if (error) {
            console.error('Error running make:', error);
            return;
        }
        console.log('Make completed successfully.');

        exec('../lex_yacc/program.out < file.in', function (error, stdout, stderr) {
            if (error) {
                console.error('Error running a.out:', error);
                return;
            }
            console.log('Program execution completed.');
            ws.send(stdout.toString());
        });
    });

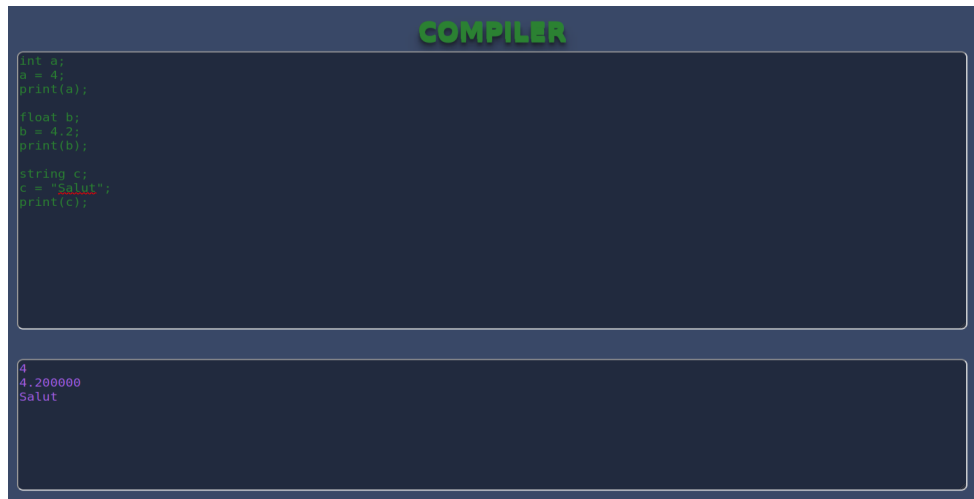
    websocket.addEventListener('message', ((data) => {
III   outputArea.value = data;
    }));
});

```

*client server code-example*



## 4 Exemple de rulare



The screenshot shows a compiler window with a dark blue background. At the top, the word "COMPILER" is written in green. Below it, a text area contains the following C++ code:

```
int a;  
a = 4;  
print(a);  
  
float b;  
b = 4.2;  
print(b);  
  
string c;  
c = "Salut";  
print(c);
```

Below the code area, a separate box displays the output of the program:

```
4  
4.200000  
Salut
```

*Asignarea variabilelor*



The screenshot shows a compiler window with a dark blue background. At the top, the word "COMPILER" is written in green. Below it, a text area contains the following C++ code:

```
int a;  
int b;  
string c;  
a = 2;  
b = 3;  
if (a > b)  
    c = "mare";  
else  
    c = "mic";  
print(c);
```

Below the code area, a separate box displays the output of the program:

```
mic
```

*Instructiunea if*

COMPILER

```
int a;  
a = 3;  
int b;  
b = 6;  
string c;  
c = "success";  
switch(a)  
{  
    case 1: print(a);  
    case 2: print(b);  
    default: print(c);  
}endSwitch
```

success

*Instructiunea **switch***

COMPILER

```
int a;  
a = 0;  
while (a < 4) {  
    print(a);  
    a = a + 1;  
}  
.
```

0  
1  
2  
3

*Instructiunea **while***

```
COMPILER

float a;
a = 1.2;
repeat {
    print(a);
    a = a+1.0;
} until ( a < 5.5);

1.200000
2.200000
3.200000
4.200000
5.200000
```

*Instructiunea repeat until*

```
COMPILER

float a;
int b;
b = 1;
for a = 1.2; a < 5.1; a = a + 1.0; {
    print(b);
    b = b + 2;
}

1
3
5
7
```

*Instructiunea for*

```
COMPILER

tic tac toe
put on 1 1
put on 1 2
put on 2 1
put on 2 2
put on 3 1
.

0|X| |
0|X| |
-----
0| | |
-----
0 won!
```

*Instructiunea Tic Tac Toe*

```
COMPILER

int a;
a = 0;
while (a < 20)
{
    printf(
        "%d\n",
        a * a * 1);
}

float b;
if (a == 20)
{
    b = 3.4;
}
else {
    b = 8.5;
}
printf(
    "%f\n",
    b);

switch(a)
{
    case 1: c = "abc";
    case 2: c = "def";
    default: c = "ghi";
}

repeat {
    a = a + 1;
    printf(
        "%d\n",
        a);
} until(a < 4);

0
1
3.400000
def
def
```

*Mai multe instructiuni combinate*

## 5 Link proiect

Proiectul poate fi regasit aici.