Evaluarea expresiilor aritmetice

January 23, 2023

Prin expresie intelegem o insiruire de operanzi legati intre ei prin operatori aritmetici(uzual `+,-,*,/`, carora li se pot adauga diversi operatori nestandard definiti in textul problemei) si eventual, paranteze. Expresia este data sub forma unui sir de caractere iar scopul este de a determina rezultatul expresiei, respectand ordinea efectuarii operatiilor(prioritatea operatorilor). Sunt descrisi o serie de algoritmi pentru rezolvarea acestei probleme. Dintre acestia vom considera 3 variante de abordare:

1. algoritm bazat pe tehnica Divide et Impera
2. algoritm bazat pe recursivitatea indirecta
3. algoritmul lui Dijkstra cu stiva

1 Algoritm folosind Divide et Impera

Tehnica DI presupune impartirea problemei date in probleme mai mici, de acelasi tip si disjuncte, care se rezolva independent iar solutiile lor se combina pentru rezolvarea problemei initiale. Procesul se aplica recursiv pana cand instanta problemei de rezolvat este atomica(nu mai poate fi impartita si admite rezolvare directa).

In acest caz, vom considera expresia ca fiind o compunere de subexpresii, intentionand sa determinam de fiecare data ultima operatie care se va executa, aceasta reprezentant punctul prin care vom realiza impartirea problemei. Se disting 4 situatii, care trebuiesc considerate in aceasta ordine:

1. exista cel putin un operand `+,-` aflat in afara oricarei paranteze. In acest caz, cel mai din dreapta astfel de operator va fi ultimul executat
2. exista cel putin un operand `*,/` aflat in afara oricarei paranteze. In acest caz, cel mai din dreapta astfel de operator va fi ultimul executat

3. expresia incepe si se termina cu `(,)`, ceea ce inseamna ca parantezele sunt inutile, le eliminam si trecem la evaluarea expresiei interne

4. expresia este un numar

O posibila abordare a acestui scenarui este sa consideram un subprogram eval, ce are ca parametri doi indici `st,dr` reprezentand capatul stang si captul drept al expresiei curente de evaluat, si care urmareste, in ordine, cele 4 scenarii descrise mai sus. Astfel, vom cauta mai intai cel mai din dreapta operator `+,-`. Fie p pozitia la care l-am gasit(presupunem p=-1, daca nu gasim). Daca p nu este -1, adica, am gasit operatorul, reapelam eval pentru subexpresiile determinate de indicii `st,p-1` si `p+1,dr`, urmand ca rezultatele subexpresiilor sa fie combinate corespunzator cu operatorul aflat pe pozitia p. In cazul in care p=-1, repetam procedura, cautand de aceasta data `*,/`. Daca nu gasim, inseamna ca expresia nu contine niciun semn aflat in afara unei paranteze.
Daca totusi cotine paranteze, atunci este de tipul `(E)`, caz in care parantezele sunt inutile, si reevaluam expresia pentru indicii `st+1,dr-1`. In final, daca nu ne-am aflat nici in al 3-lea caz, expresia este in sine un numar, si il returnam ca atare. Se pune problema sa verificam daca un operator gasit este sau nu in afara oricarei paranteze. Pentru aceasta, o buna metoda este sa contorizam fiecare paranteza inchisa cu +1, si fiecare paranteza deschisa cu -1(asta pentru ca parcurgem de la dreapta catre stanga). Astfel, un operator va fi in afara oricarei paranteze daca si numai daca atunci cand il intalnim, contorul de paranteze este 0, adica s-au inchist tot atatea paranteze cate s-au si deschis.

O posibila implementare a agloritmului este :

```
#include <iostream>
#include <fstream>
#include <string.h>

using namespace std;

char s[100005];

int cautas(int st, int dr, char s1, char s2)
{
    int nr=0;
    for(int i=dr;i>=st;i--)
    {
        if(s[i]==')')
```

```
            nr++;
        if(s[i]=='(')
            nr--;
        if(nr==0 && (s1==s[i] || s2==s[i]))
            return i;
    }
     return -1;
}

int num(int st, int dr)
{
    int numar=0;
    for(int i=st;i<=dr;i++)
        numar=numar*10+(s[i]-'0');
    return numar;
}

int solve(int st, int dr)
{
    int poz=cautas(st,dr,'+','-');
    if(poz!=-1)
    {
        int e1=solve(st,poz-1);
        int e2=solve(poz+1,dr);
        if(s[poz]=='+')
            return e1+e2;
        return e1-e2;
    }
    poz=cautas(st,dr,'*','-/');
    if(poz!=-1)
    {
        int e1=solve(st,poz-1);
        int e2=solve(poz+1,dr);
        if(s[poz]=='*')
            return e1*e2;
        return e1/e2;
    }
    if(s[st]=='(' && s[dr]==')')
        return solve(st+1,dr-1);
    return num(st,dr);
}
```

```
int main()
{
    ifstream fin("evaluare.in");
    ofstream fout("evaluare.out");
    fin.getline(s,100005);
    int n=strlen(s);
    fout<<solve(0,n-1);
    return 0;
}
```

In aceasta implementare functia eval are comportamentul descris deja. Functia cautas primeste ca parametri st,dr cu semnificatiile stabilite, si doua variabile de tip char, `s1,s2`, reprezentand operatorii cautati. Aceasta parcurge exspresia de la dreapta catre stanga, numara parantezele, si returneaza pozitia celui mai din dreapta operator care nu se afla in paranteza, sau -1 daca nu gaseste astfel de operator.
Functia num este apelata cand expresia delimitata de indicii st,dr este un numar si returneaza valoarea acestui numar.

2 Algoritm folosind recursivitatea indirecta

Recursivitatea indirecta este o tehnica in care un set de functii recursive se apeleaza intre ele. Acest lucru impune descrierea antetelor functiilor inainte de a descrie corpul acestora, pentru a asigura intervizibilitatea. Consideram ca fiecare expresie este impartita, si aici in componente, astfel:

1. termeni ai unei sume, spearati intre ei prin operatori `+,-`
2. factori ai unui produs, spearati intre ei prin operatori `*,/`
3. subexpresii de tipul `(E)` sau valori numerice.

Deoarece subexpresiile pot fi formate la randul lor din termeni sau factori este evidenta necesitatea revenirii la aceste cazuri, si implicit a recursivitatii indirecte. Fiecare termen al unei adunari poate fi compus la randul sau din mai multi factori inmultiti, iar fiecare factor al unei inmultiri poate reprezenta la randul sau o subexpresie sau un numar. Astfel, vom avea 3 subprograme, unul care se ocupa de factori, unul care se ocupa de termeni, si unul care se ocupa de expresia in ansamblul sau si implicit de subexpresii.

O posibila implementare este:

```
#include <cstdio>
#include <cstring>
#include <cctype>
#define Lmax 100005

using namespace std;

FILE *fin=freopen("evaluare.in","r",stdin);
FILE *fout=freopen("evaluare.out","w",stdout);

char s[Lmax];
int i=0;

int num();
int mult();

int solve()
{
    long t=mult();
    while(s[i]=='+' || s[i]=='-')
    {
        if(s[i]=='+')
        {
            i++;
            t+=mult();
        }
        else
        {
            i++;
            t-=mult();
        }
    }
    return t;
}

int mult()
{
    long t=num();
    while(s[i]=='*' || s[i]=='/')
    {
        if(s[i]=='*')
```

```
        {
            i++;
            t*=num();
        }
        else
        {
            i++;
            t/=num();
        }
    }
    return t;
}

int num()
{
    int t=0;
    if(s[i]=='(')
    {
        i++;
        t=solve();
        i++;
    }
    else
        while(isdigit(s[i]))
        {
            t=t*10+s[i++]-'0';
        }
    return t;
}

void citire()
{
    fgets(s,Lmax,stdin);
}

int main()
{
    citire();
    printf("%d\n",solve());
    return 0;
}
```

In aceasta implementare avem expresia memorata sub forma sirului de caractere s, declarat global, iar variabila `i` reprezinta indicele la care ne aflam in expresie la momentul respectiv. Functia `solve()` evalueaza fiecare termen al unei adunari, acesta fiind alcatuit la randul sau din factori ai unui produs, atfel cauta primul factor. Fiecare factor va fi evaluat de functia `mult()`, care la randul sau imparte factorul in subexpresii, de care se ocupa functia `num()`. Aceasta intalneste doua situatii: subexpresia continuta intre paranteze, care se elimina, dar trebuie considerata o noua expresie, caz in care recurvisitatea indirecta revine la `solve()`, sau este un numar, pe care il calculeaza si returneaza.

3 Forme poloneze

O formă poloneză a unei expresii aritmetice este o notaţie matematică compacta, în care prin modificarea ordinii între operanzi şi operatori se obţin expresii echivalente, dar care nu conţin paranteze. Computaţional acestea au o serie de avantaje, cum ar fi evaluarea cu algoritm liniar, citirea şi prelucrarea uşoară.
Să considerăm expresia $a + b - c$, în care operanzi sunt variabilele a, b, c şi operatorii sunt $+, -$. Forma uzuală a expresiei nu este o notaţie poloneză, dar poate fi considerată **forma infixată** a expresiei, pentru că operatorul se află poziţionat între operanzi.
Vor exista atfel două forme poloneze:

- **forma poloneză prefixată** - în care operatorul se află înaintea operanzilor

$$-+abc$$

- **forma poloneză postfixată** - în care operatorul se află după operanzi

$$ab + c-$$

Pornind de la o expresie în formă infixată putem obţine formele poloneze cu un algoritm ce utilizează o stivă, sau folosind arbori.

3.1 Forma poloneză prefixată

În forma prefixată operatorul se afla înaintea operanzilor, deci $a+b$ va deveni $+ab$. Pentru expresii mai complicate putem considera că orice expresie mai mare poate fi văzută ca o operaţie între doua expresii mai mici şi folosim notaţia $E = E_1 op E_2$, unde E va fi expresia originală, E_1, E_2 expresiile ce o compun şi *op* va fi ultimul operator care se execută în expresia originală, adică:

1. cel mai din dreapta semn $+-$ care nu se află într-o paranteză
2. cel mai din dreapta semn $*/$ care nu se află într-o paranteză
3. cea mai din dreapta operaţie de ordinul 3 care nu se află într-o paranteză

Dacă nu exxista niciun semn în afara oricărei parenteze, atunci expresia noastra va fi de forma (E), deci eliminăm parantezele şi continuăm. Pentru a implementare avem la dispozitie tehnica divide et impera descrisa in prima sectiune, complexitatea pe care o obţinem fiind una pătratrică. Să urmărim cum funţioneaza pe un exemplu . Fie expresia $2+3*(5-(4/2-1)) \uparrow (8/(1+1))$:

Expresie	Operator
$\underline{2+3*(5-(4/2-1)) \uparrow (8/(1+1))}$	$+$
$+2\ \underline{3*(5-(4/2-1)) \uparrow (8/(1+1))}$	*
$+2*3\ \underline{(5-(4/2-1)) \uparrow (8/(1+1))}$	$\uparrow$
$+2*3 \uparrow \underline{(5-(4/2-1))}\ \underline{(8/(1+1))}$	nu există, eliminăm paranteze
$+2*3 \uparrow \underline{5-(4/2-1)}\ \underline{8/(1+1)}$	-, eliminăm paranteze
$+2*3 \uparrow -5\ \underline{4/2-1}\ \underline{8/(1+1)}$	-
$+2*3 \uparrow -5-\ \underline{4/2}\ 1\ \underline{8/(1+1)}$	/
$+2*3 \uparrow -5-/4\ 2\ 1\ \underline{8/(1+1)}$	/, eliminăm paranteze
$+2*3 \uparrow -5-/4\ 2\ 1/8\ \underline{1+1}$	$+$
$+2*3 \uparrow -5-/4\ 2\ 1/8+1\ 1$	

La fiecare pas, am marcat ca fiind subliniată porţiunea de expresie care nu se află în forma poloneză, deci mai trebuie prelucrată.

3.2 Forma poloneză postfixată

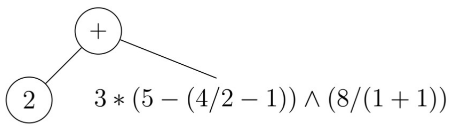
În mod complet analog se obţine forma poloneză prefixată, punând de această dată operatorul la sfârşit. Pentru expresia de mai sus vom obţine:

$$2\ 3\ 5\ 4\ 2/1--8\ 1\ 1+/^{*}+$$

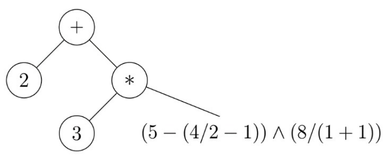
4 Arborele unei expresii aritmetice

O altă metodă utilă de a converti o expresie matematică infixată în formă poloneză este folosind arborele de expresie. Un arbore de expresie este un arbore binar în care fiecare nod, cu exceptia frunzelor, conţine câte un operator, iar frunzele conţin operanzi, cu semnificaţia că operatorul din nod va fi aplicat având ca operanzi conţinutul din subarborele drept şi pe cel din subarborele stâng. Mecanismul va fi asemănător cu cel de la obţinerea directă formelor poloneze, căutând la fiecare pas cel mai din dreapta semn de prioritate minimă, pe care în punem în rădăcină, iar cu operanzii săi continuăm recursiv pe cei doi subarbori. În acest caz, după construcţia aborelui, parcurgerea sa in preordine va furniza forma prefixată a expresiei, iar parcurgerea în postordine forma postfixată. Să urmărim de exemplu expresia $2 + 3 * (5 - (4/2 - 1)) \uparrow (8/(1 + 1))$

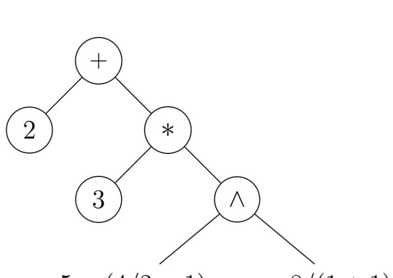
Cel mai din dreapta semn de prioritate minimă este $+$, deci se va pune în rădăcina arboirelui. Operanzii săi sunt 2 care va deveni fiu stâng, şi $3 * (5 - (4/2 - 1)) \uparrow (8/(1 + 1))$ care va deveni fiu drept.



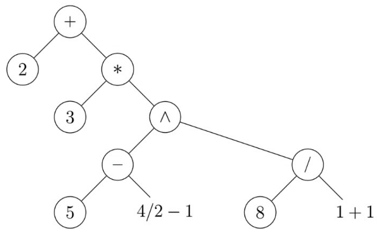
Continuăm pe subarborele drept unde vom pune în rădăcină $*$, 3 pe subarborele stâng şi restul expresiei pe dreeapta.

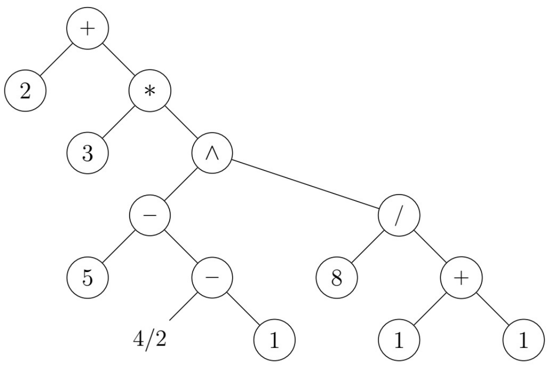


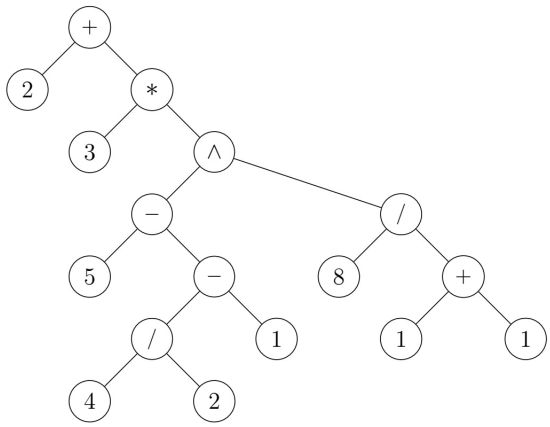
Continuăm pe subarborele drept unde vom pune în rădăcină $\uparrow$, $5 - (4/2 - 1)$ pe subarborele stâng şi $8/(1 + 1)$ pe dreapta.



Continuând procedeul obţinem, succesiv arborii:







5 Evaluarea unei expresii aritmetice în formă poloneză

Dacă se consideră o expresie matematică în formă poloneză prefixată care trbuie evaluată, atunci fiecare grup consecutiv de elemente de tipul operator operand operand va fi inlocuit cu valoarea sa, procedeul repetându-se până când expresia conţine o singura valoare, care va reprezenta rezultatul.

Observaţie 1 *Pentru orice expresie în formă poloneză numă rul de operatori va fi cu unul mai puţin decât numărul de operanzi, proprietate care trebuie să se păstreze în orice subşir de lungime impară al expresiei.*

Expresie
- + / + 3 * 3 5 + - 4 - 7 4 + * 1 2 * 1 3 * + ↑ 3 2 ↑ 2 2 / 8 - 3 1 5
- + / + 3 15 + - 4 3 + 2 3 * + 9 4 / 8 2 5
- + / 18 + 1 5 * 13 4 5
- + / 18 6 52 5
- + 3 52 5
- 55 5
50

Procedeul este analog pentru expresiile în formă postfixată. În practică, procedeul descris va folosi o stivă, în care elementele se introduc succesiv,expresia în formă prefixată fiin parcursă invers. Fiecare operand este introdus în stivă, iar la întâlnirea unui operator, cei mai din vârf doi operanzi se extrag din stivă, se calculează şi se introduce rezultatul înapoi în stivă. Dacă la un pas nu există suficienţi operanzi în stivă, înseamnă că expresia iniţială a fost greşită.

6 Algoritmul lui Dijkstra pentru evaluarea expresiilor

Conversia unei expresii în forma poloneză cu ajutorul arborelui de expresie conduce la o complexitate finală pătratică, ceea ce nu este avantajos. Un algoritm liniar pentru evaluarea expresiilor, dar care furnizează totodată şi forma poloneză a fost conceput de către matematicianul olandez Edsger Dijkstra. Acest algoritm are la bază doua stive, una pentru operatori şi una pentru operanzi, iar expresia se parcurge liniar. La fiecare pas vom analiza câte un element al expresiei. Dacă este operand va fi pus în stiva de operanzi. Dacă este operator vom avea una din următoarele două situaţii: daca stiva de operanzi este goală, sau operandul curent are prioritate mai mare decât cel din vârful stivei de operanzi, va fi introdus ; altfel, eliminăm din stiva de operanzi elemente până cand aceasta este goala, sau operandul din vârf va avea prioritate mai mică decât cel curent. Elementele eliminate din stiva de operanzi vor fi introduse în aceeaşi ordine în stiva de operatori, care va furniza rezultatul. Parantezele deschise vor fi introduse in stiva de operanzi, iar la intalnirea uneia inchise se va extrage din stiva de operanzi tot conţinutul, pânľa întâlnirea primei paranteze.

Expresie	St Operatori	St operanzi
2 + 3 * 5 - 7 + 8 - 2 * 3	stiva vida	stiva vida
+ 3 * 5 - 7 + 8 - 2 * 3	stiva vida	2
3 * 5 - 7 + 8 - 2 * 3	+	2
* 5 - 7 + 8 - 2 * 3	+	2 3
5 - 7 + 8 - 2 * 3	+ *	2 3
- 7 + 8 - 2 * 3	+ *	2 3 5
7 + 8 - 2 * 3	-	2 3 5 * +
+ 8 - 2 * 3	-	2 3 5 * + 7
8 - 2 * 3	+	2 3 5 * + 7 -
- 2 * 3	+	2 3 5 * + 7 - 8
2 * 3	-	2 3 5 * + 7 - 8 +
* 3	-	2 3 5 * + 7 - 8 + 2
3	- *	2 3 5 * + 7 - 8 + 2
expresie vida	- *	2 3 5 * + 7 - 8 + 2 3
expresie vida	stiva vida	2 3 5 * + 7 - 8 + 2 3 * -

Expresie	St Operatori	St operanzi
3 + 4 * 2 / (1 - 5) ↑ 2	stiva vida	stiva vida
+ 4 * 2 / (1 - 5) ↑ 2	stiva vida	3
4 * 2 / (1 - 5) ↑ 2	+	3
* 2 / (1 - 5) ↑ 2	+	3 4
2 / (1 - 5) ↑ 2	+ *	3 4
/ (1 - 5) ↑ 2	+ *	3 4 2
(1 - 5) ↑ 2	+ /	3 4 2 *
1 - 5) ↑ 2	+ / (	3 4 2 *
- 5) ↑ 2	+ / (	3 4 2 * 1
5) ↑ 2	+ / (-	3 4 2 * 1
) ↑ 2	+ / (-	3 4 2 * 1 5
↑ 2	+ /	3 4 2 * 1 5 -
2	+ / ↑	3 4 2 * 1 5 -
expresie vida	+ / ↑	3 4 2 * 1 5 - 2
expresie vida	stiva vida	3 4 2 * 1 5 - 2 ↑ / +

Mai jos găsiţi algoritmul lui Dijkstra cu stiva pentru evaluarea expresiilor.

```
#include <iostream>
#include <fstream>
#include <stack>
#include <cstring>

using namespace std;

char s[100005];
int p=0;

stack <char> op;
stack <int> nr;

int eval(int x, int y, char semn)
{
    switch (semn)
    {
        case '+': return x+y;
        case '-': return x-y;
        case '*': return x*y;
        case '/': return x/y;
    }
}

int nextnum()
{
    int numar=0;
    while(s[p]>='0' && s[p]<='9')
        numar=numar*10+(s[p++]-'0');
    return numar;
}

int prioritate(char s1, char s2)
{
    if(s1=='(')
        return 1;
    if((s1=='-' || s1=='+') && (s2=='*' || s2=='/'))
        return 1;
    return 0;
}
```

```
int main()
{
    ifstream fin("evaluare.in");
    ofstream fout("evaluare.out");
    fin.getline(s,100005);
    int n=strlen(s);
    while(p<n)
    {
        switch (s[p])
        {
        case '(':{op.push(s[p++]);break;}
        case')':{
            while(op.top()!='(')
            {
                int v2=nr.top();
                nr.pop();
                int v1=nr.top();
                nr.pop();
                char semn=op.top();
                op.pop();
                nr.push(eval(v1,v2,semn));
            }
            op.pop();
            p++;
            break;
            }
        case'+':
        case'-':
        case'*':
        case'/':{
            if(op.empty() || prioritate(op.top(),s[p])==1)
            {
                op.push(s[p++]);
            }
            else
            {
                do{
                    int v2=nr.top();
                    nr.pop();
                    int v1=nr.top();
                    nr.pop();
```

```
                    char semn=op.top();
                    op.pop();
                    nr.push(eval(v1,v2,semn));
                }while(!op.empty() && prioritate(op.top(),s[p])==0);
                op.push(s[p++]);

            }
            break;
        }
        default :{
            nr.push(nextnum());
        }
        }

    }
    while(!op.empty())
    {
         int v2=nr.top();
                    nr.pop();
                    int v1=nr.top();
                    nr.pop();
                    char semn=op.top();
                    op.pop();
                    nr.push(eval(v1,v2,semn));
    }
    fout<<nr.top();
    return 0;
}
```