**Evaluarea expresiilor aritmetice**

Problema evaluării unei expresii aritmetice este des întâlnită în concursurile de programare. Totodară este și cu o mare aplicabilitate practică, dacă ne gândim că vom provoca lansarea la un moment dat a unui astfel de algoritm de fiecare dată când scriem o expresie în limbaj (de exemplu o atribuire). Atunci când codul scris de noi se va analiza și transforma, unul dintre pașii care se fac este evaluarea expresiilor care apar.

În acest material voi prezenta un algoritm care se bazează pe recursivitatea indirectă.

Să enunțăm mai întâi problema (o vom face utilizând constrângerile problemei din arhiva educațională infoarena, https://infoarena.ro/problema/evaluare).

Se dă o expresie aritmetică alcătuită din:

- semnele `+`, `-`, `*`, `/` (reprezentând operațiile matematice de adunare, scăderem înmulțire respectiv cât al împărțirii întregi);
- paranteze rotunde;
- secvențe de cifre, cu semnificația de constante întregi.

Expresia nu conține spații sau alte caractere. Operatorii aritmetici prezenți sunt toți binari, cel de înmulțire și cel de cât au prioritate mai mare decât cel de adunare și cel de scădere, iar în caz de priorități egale operatorii se aplică de la stânga la dreapta. Expresia se dă corectă (parantezată corect, operatorii se aplică în mod corect etc).
Se cere să determinăm valoarea întreagă care se obține în urma evaluării.

*Exemplu:*
Pentru expresia `(10+(1+1)*13)/8+2*5/3-5` valoarea de după evaluare ar fi `2`.

Operațiile s-ar aplica astfel:
Se realizează calculul din cel mai adânc set de paranteze, adică `1+1` cu rezultatul `2`.
Se înmulțește apoi mai întâi `2` cu `13` și apoi se adună `10`.
Se determină câtul împărțirii lui `36` la `8` care este `4`.
Independent de calculele anterioare se poate realiza la un moment dat și calculul `2*5/3`, iar în final ar fi de operat `4+3-5` cu rezultatul final `2`.

Iată mai departe o strategie de a “ataca” problema.
Avem așadar de evaluat o *expresie*.
Pentru asta identificăm mai întâi operatorii de prioritatea cea mai mică (la noi ei sunt `+` și `-` și ne vom referi la ambii deodată prin șirul `+-`).
Pentru expresia de evaluat ne interesează acei operatori de acest tip care nu sunt incluși în niciun set de paranteze.
Privind lucrurile astfel, considerăm că acești operatori separă expresia în așa numiți *termeni*.

Adică ne imaginăm lucrurile așa:

$$\texttt{expresie} = \texttt{termen}_1 \texttt{ +- termen}_2 \texttt{ +- … +- termen}_n$$

Pentru expresia din exemplul nostru avem

| expresia | | | | |
|---|---|---|---|---|
| (10+(1+1)*13)/8 | + | 2*5/3 | - | 5 |
| $\texttt{termen}_1$ | | $\texttt{termen}_2$ | | $\texttt{termen}_3$ |

Acum analizăm structura unui *termen*. Pentru acest lucru ne gândim la urmuătorul grup de operatori în funcție de prioritate. În cazul nostru ei sunt `*` și `/` și vom nota grupul `*/`.
Dacă pentru un *termen* dat considerăm acești operatori (aceia din afara unor paranteze din interiorul termenului) observăm că un *termen* se poate scrie astfel:

$$\texttt{termen} = \texttt{factor}_1 \texttt{ */ factor}_2 \texttt{ */ … */ factor}_m$$

Observăm o similitudine între modului de descompunere a *expresiei* în *termeni* și modul de descompunere a unui *termen* în *factori*.
În exemplul nostru descompunerile sunt:

Pentru termen1:

| termen | | |
|---|---|---|
| (10+(1+1)*13) | / | 8 |
| $\texttt{factor}_1$ | | $\texttt{factor}_2$ |

Pentru termen2

| termen | | | | |
|---|---|---|---|---|
| 2 | * | 5 | / | 3 |
| $\texttt{factor}_1$ | | $\texttt{factor}_2$ | | $\texttt{factor}_3$ |

În fine, să analizăm care este structura unui *factor*. Dacă am mai fi avut operatori cu alt nivel de prioritate, mai mare, am fi descompus un *factor* în elemente mai mici așa cum am procedat la *expresie* și la *termen*. Dar noi nu mai avem alt nivel de operatori.
Observăm din cele două tabele anterioare că un *factor* poate fi:

- Un șir de cifre (o constantă, în exemplele noastre `8`, `2`, `5`, `3, 13, 10, 8, 1`)
- O altă *expresie* pusă între paranteze.

Adică:

`factor = constantă`
sau
`factor = (expresie)`

Acum în definirea unui *factor* nu mai facem apel la elemente noi, însă ne referim la unul definit anterior, adică *expresie*.
Să recapitulăm pentru o mai bună înțelegere:

- Intervin în analiza noastră noțiunile: *expresie*, *termen*, *factor*, *constantă* și ele sunt definite astfel:

| |
|---|
| `expresie = termen`$_1$ `+- termen`$_2$ `+- … +- termen`$_n$ |
| `termen = factor`$_1$ `*/ factor`$_2$ `*/ … */ factor`$_m$ |
| `factor = constantă`<br>sau<br>`factor = (expresie)` |
| `constantă = secvență de cifre` |

Pentru a implementa într-un program cele analizate prezentăm următoarea variantă.
Scriem câte o funcție pentru a trata fiecare tip de element.
Observăm că singura funcție care nu face apel la alta este cea corespunzătoare constantei.
În rest, fiecare dintre celelalte face apel la altă funcție: *expresie* apelează pe *termen*, *termen* apelează pe *factor*, *factor* apelează pe constantă dar și pe *expresie*.
Apare astfel o dependență circulară între *expresie*, *termen* și *factor*. Niciuna nu se apelează pe ea însăși dar fiecare poate ajunge să se autoapeleze indirect prin intermediul celorlalte.

Noi vom implementa în limbaj cele de mai sus.
Mai întâi să stabilim semnăturile funcțiilor.
Fiecare dintre ele ar trebui să returneze un număr ce reprezintă rezultatul evaluării termenului de care ea se ocupă.

Și acum o idee foarte importantă: în fiecare funcție vom avansa cu un indice caracter cu caracter și dependent de context decidem în ce fel de element suntem. La niciuna dintre funcții să nu nu vom folosi parametri dar ne asigurăm ca toate să fie atente la acest indice (care va fi deci global). Tot global va fi șirul care stochează datele de intrare.
Alt lucru foarte important de lămurit înainte de a începe implementarea este dat de ordinea în care definim funcțiile. Știm că în C/C++ (dar și în alte limbaje) un identificator trebuie anunțat în codînainte de locul primei folosiri. Având însă dependență circulară, indiferent cu care dintre cele trei vom începe nu avem cum să respectăm regula. Pentru aceste situații (și nu numai) vom folosi facilitatea limbajului de a anunța mai întâi antetele (prototipurile) funcțiilor și a reveni ulterior cu implementarea completă. Anunțarea antetului presupune scrierea unei linii unde indicăm tipului de date returnat de funcție, numele funcției, parametrii (sunt suficiente tipurile lor în ordinea corectă) și care se termină cu “punct și virgulă”.
Astfel, structura programului nostru va fi:

```
#include <iostream>
using namespace std;

char s[100001];
int i;

int expresie();
int termen();
int factor();
int constanta();

int main () {
    cin>>s;
    i = 0;
    cout<<expresie();
}

int expresie () {
    /// Implementarea completa
}
int termen () {
    /// Implementarea completa
}
int factor () {
    /// Implementarea completa
}
int constanta () {
    /// Implementarea completa
}
```

Pentru claritate am aplicat și pentru funcția constanta cele discutate despre ordinea declarării. Acest lucru nu era obligatoriu, de exemplu pe constanta o puteam declara și implementa înainte de *factor*.

Trecem acum la implementarea fiecărei funcții, să începem cu *expresie*.
Vom considera că `i` se află pe primul caracter al *expresiei* și pentru a păstra integritatea algoritmului va trebui să ne asigurăm că la fiecare apel al altei funcții `i` va fi mutat corespunzător de codul executat anterior pentru a fi pe primul caracter de tratat în acel apel.
Pentru început `i` va fi `0` (poziția primului caracter din șir deci și primul din expresia principală).
Ne amintim structura unei expresii, adică $termen_1$ +- $termen_2$ +- … +- $termen_n$.
Astfel mai întâi vom apela functia *termen* și ne gandim ca aceasta ar trebui sa returneze valoarea primului *termen* al *expresiei* dar totodată va muta și pe `i` încât acesta să treacă de toate caracterele acelui *termen*. Astfel, avem două variante: fie `i` ajunge pe un `+` sau `-` (semn pentru noi că expresia la care tocmai evaluăm are mai mult de un *termen*) fie s-a terminat expresia.
Așa că, dacă decidem să apelăm iar *termen* (`i` ajunsese în dreptul unui caracter `+` sau `-`) trebuie mai întâi să mărim pe `i` (dacă apelăm iar *termen* trebuie să fim cu `i` pe primul caracter din *termen*, deci trebuie să sărim peste `+-`).
În cadrul funcției *expresie* vom folosi o variabilă locală în care obținem rezultatul final al *expresiei*, pe care il vom și returna la final. Notăm cu `r` această variabilă. Inițial `r` va lua valoarea primului *termen*, apoi, dacă mai sunt *termeni*, așa cum spuneam, apelăm iarăși funcția *termen* și acumulăm rezultatul la `r` (în funcție de semnul de dinaintea termenului). Aici vom folosi o structură repetitivă și la fiecare pas acumulăm la `r` câte un *termen*.
Având în minte cele explicate mai sus, funcția *expresie* ar fi:

```
int expresie() {
   /**
    apel al functiei care ne va extrage primul termen al
    acestei expresii
   **/
   int r = termen();
   while (s[i] == '+' || s[i] == '-') {
       if (s[i] == '+') {
           i++; /// sar peste plus
           r += termen();
       } else {
           I++; /// sar peste minus
           r -= termen();
       }
   }
   return r;
}
```

Atragem atenția asupra următorului detaliu de implementare. Observăm că `i++` este scris pe ambele ramuri ale lui `if` și astfel de situații ne fac să mutăm pe `i` în afara `if`-ului. Atenție că în

acest caz `i` apare și în condiția de la `if` și ar trebui să modificăm și acolo dacă mutăm incrementarea înainte (de exemplu ar trebui ca în condiție să ne referim la `s[i-1]`). Scopul pentru care am scris ca mai sus este unul didactic. Evident că odată înțeles bine algoritmul fiecare programator îl poate adapta conform stilului său.

Așa cum am prezentat și la analiza inițială, funcția *termen* va avea o sutructură simulară cu *expresie*: pune mai întâi în `r` primul *factor* al termenului apoi acumulează în mod repetat la el pe fiecare dintre ceilalți. Practic, vom putea face și o implementare mai abstractă: copy paste de la *expresie*; înlocuim *expresie* cu *termen*; înlocuim *termen* cu *factor*; înlocuim operatorii `+-` cu `*/`.

```
int termen() {
   int r = factor();
   while (s[i] == '*' || s[i] == '/') {
       if (s[i] == '*') {
           i++;
           r *= factor();
       } else {
           i++;
           r /= factor();
       }
   }
   return r;
}
```

La implementarea funcției *factor* vom avea așadar de tratat cele două cazuri: “un *factor* este fie un șir constant fie o altă *expresie* plasată între paranteze”.

```
int factor() {
    int r;
    if (s[i] == '(') {
        i++; /// sar peste paranteza deschisa
        r = expresie();
        i++; /// sar peste paranteza inchisa
    } else {
        r = constanta();
    }
    return r;
}
```

Dacă în funcțiile *expresie* și *termen* am făcut salturi ale lui `i` pese oparatori, în acestă funcție sărim cu `i` peste paranteze iar în funcția constantă vom trata și salturile lui `i` peste cifre. Atragem atenția asupra unui alt detaliu de implementare. Noi punem rezultatele de pe fiecare ramură în `r` și îl returnăm pe acesta la final. Ar părea că putem pune return direct pe ramuri.

Acest lucru ar fi greșit la prima ramură pentru că nu s-ar mai executa `i++` după apelul *expresie*, adică nu am mai sări peste paranteza închisă. Evident că s-ar strica programul.

Pentru funcția constanta nu avem decât să iterăm cu `i` pe toată secvența de cifre consecutive și să construim numărul în baza `10` corespunzător. Atenție să nu uitați că trebuie să trecem de la simbolul de caracter cifră la valoarea numerică.

```
int constanta() {
    int r = 0;
    while (s[i] >= '0' && s[i]<='9') {
        r = r + s[i] - '0';
        i++;
    }
    return r;
}
```

Timpul de calcul este liniar. Ordinul depinde de lungimea expresiei și de adâncimea parantezării (de exemplu la un moment dat i poate sta pe loc și se tot fac apeluri dintr-o funcție în alta dar acest număr depinde de adâncimea parantezării). Adâncimea parantezării însă este de ordinul lugimii șirului (în cazul cel mai defavorabil jumătate din lungime).

## Probleme rezolvate.

### Expresii min-max (infoarena)

Considerați o expresie care conține numere naturale, paranteze, și operatorii binari `M` si `m`. `M` este operatorul de maxim și `m` este operatorul de minim. Astfel, rezultatul operației `A m B` este valoarea minimă dintre `A` si `B`, iar rezultatul operației `A M B` este valoarea maximă dintre `A` și `B`. De exemplu, rezultatul `2m7` este `2`, iar rezultatul `9M8` este `9`. Cei doi operatori au aceeași prioritate. Asta înseamnă că dacă nu sunt paranteze, vor fi evaluati de la stanga la dreapta. De exemplu, rezultatul `1M22m13m789` este `13` iar pentru `(1m1m1M1M1m1M1M1m1M0)m1M1` este `1`.

*Soluție*
Problema este “cu un nivel” mai simplă decât cea prezentată la explicarea algoritmului. Adică avem operatori de un singur nivel de prioritate (`m` și `M` se aplică de la stânga la dreapta dacă apar unul după altul).
Rămâne doar să fim atenți la combinarea rezultatului curent cu al următorului element de expresie, acum realizându-se un minim sau un maxim.

*Sursa*

```
#include <fstream>
using namespace std;
char s[100100];
```

```
int i;
int expresie();
int factor();

int expresie () {
    int r = factor();
    while (s[i] == 'm' || s[i] == 'M') {
        if (s[i] == 'm') {
            i++;
            r= min(r, factor());
        } else {
            i++;
            r= max(r,factor());
        }
    }
    return r;
}


int factor() {
    int r;
    if (s[i] == '(') {
        i++; // salt peste paranteza deschisa
        r = expresie();
        i++; // salt peste paranteza inchisa
    } else {
        // sigur este o valoare
        r = 0;
        while (s[i] >= '0' && s[i] <= '9') {
            r = r * 10 + s[i]-'0';
            i++;
        }
    }
    return r;

}

ifstream fin ("emm.in");
ofstream fout("emm.out");

int main () {
    fin>>s;
    i = 0; // pozitia primului caracter
    fout<<expresie();

    return 0;
}
```

Am realizat o implementare mai directă, renunțând la scrierea separată a unei funcții de tratare a valorii constante. În rest este același cod, cu aceleași principii și notații ca la algoritmul prezentat inițial.

**Bool (Infoarena)**

Enunțul pe scurt:
Haralambie a primit la școală o temă destul de dificilă. El trebuie sa evalueze o expresie logică. Această expresie conține variabile (litere mari ale alfabetului englez de la `A` la `Z`), constante (`TRUE` si `FALSE`), paranteze rotunde `(` și `)`, si operatorii logici `NOT`, `AND` si `OR`. `NOT` are prioritatea cea mai mare, `OR` are prioritatea cea mai mica. Inițial toate variabilele au valoarea `FALSE`. Lui Haralambie îi place să evalueze expresii, dar variabilele își mai schimbă uneori valoarea și expresia trebuie reevaluată. Speriat din această cauză, a decis sa apeleze la ajutorul vostru!
Inițial toate variabilele au valoarea `FALSE`. Se cere evaluarea expresiei de mai multe ori, la fiecare pas indicându-se o variabilă care își schimbă valoarea (se complementează), și după fiecare astfel de pas trebuie determinat iarăși rezultatul expresiei.
De exemplul pentru testul:

```
A AND ((B OR NOT C) OR ((TRUE)))
4
ABCA
```

Se va afișa: `1110`.
Adică mai întâi se evaluează pentru `A=TRUE`, `B=FALSE`, `C=FALSE`, `D=FALSE`, apoi se complementează și `B`, adică a doua evaluare se face pentru: `A=TRUE`, `B=TRUE`, `C=FALSE`, `D=FALSE`, a treia oară se complementează `C`, deci evaluarea se va face pentru `A=TRUE`, `B=TRUE`, `C=TRUE`, `D=FALSE`, în fine se complementează iarăși `A`, deci revine la valoarea `FALSE` și ultima evaluare se face pentru: `A=FALSE`, `B=TRUE`, `C=TRUE`, `D=FALSE`.
Pentru început vom parsa șirul și îl vom transforma în unul echivalent dar mai prietenos pentru a ne putea concentra ulterior doar pe partea de evaluare de expresie. Elementele mai speciale care apar aici sunt: prezența variabilelor, prezența spațiilor, operatori și constante cu nume de lungimi diferite, apariția unui operator unar.
Iară mai jos deciziile noastre la implementarea ce va fi prezentată la finalul descrierii:

- Parcurgem caracter cu caracter șirul dat `s` și construim prin concatenare un alt șir `t` astfel: spațiile întâlnite în `s` le ignorăm, adică nu punem în `t` nimic în contul lor; când întâlnim în `s` secvența `TRUE` adăugăm în `t` caracterul `1`; pentru secvența `FALSE` în `s` adăugăm în `t` caracterul `0`; analog, adăugăm în `t` un `&` în loc de `AND`, un `|` în loc de `OR`, iar în loc de `NOT` adăugăm `1^` (acesta este un mic truc pentru de a transforma operatorul unar în unul binar, adică în loc de `not x` vom avea `1 xor x`, cu același efect).
- Semnele adăugate în `t` la descrierea din paragraful anterior au fost alese așa pentru claritatea codului, pentru a fi în concordanță cu operația din `C` corespunzătoare, dar

evident puteam alege și alte caractere pentru ele, semnificația este dată de modul în care le tratăm în cod.

- Pentru variabile păstrăm aceleași simboluri, literele mari ale alfabetului. La implementare vom ține un vector de frecvență cu câte o poziție pentru fiecare variabilă și în care notăm `0` sau `1` după cum variabila corespunzătoare este la un moment dat `TRUE` sau `FALSE`.
- Odată obținut șirul `t` prin transformare după regulile de mai sus, avem acum de evaluat o expresie fără spații, cu trei tipuri de priorități de data asta (`XOR` este acum cel mai prioritar pentru că este simularea operatorului unar, apoi `AND` și apoi `OR`). Operanzii efectivi din expresie sunt fie constantele `0` și `1` fie valorile variabilelor (pe care le obținem din vectorul de frecvență).
- Soluția prezentată mai jos declanșează algortimul de evaluare pentru fiecare modificare de variabilă.

```
#include <fstream>

using namespace std;

ifstream fin ("bool.in");
ofstream fout("bool.out");

char s[1005], t[1005];
int i, k, n;
int F[130];
char c;

int expresieOr();
int expresieAnd();
int expresieXor();
int factor();

/// functia pentru tratarea separarii prin operatorul
/// de prioritate minima
int expresieOr() {
    int r = expresieAnd();
    while (t[i] == '|') {
        i++;
        r = (r | expresieAnd());
    }
    return r;
}

int expresieAnd() {
    int r = expresieXor();
    while (t[i] == '&') {
        i++;
        r = (r & expresieXor());
    }
    return r;
```

```
}

int expresieXor() {
    int r = factor();
    while (t[i] == '^') {
        i++;
        r = (r^factor());
    }
    return r;
}

int factor() {
    int r;
    /*
     Factorii pot fi de trei tupuri:
         (expresie)
         constanta 0 sau 1
         o variabila a carei valoare o extragem din vectorul f
    */
    if (t[i] == '(') {
        i++;
        r = expresieOr();
        i++;
    } else {
        if (t[i] =='0') {
            i++;
            return 0;
        }
        if (t[i] == '1') {
            i++;
            return 1;
        }
        return F[ t[i++] ];
    }
    return r;
}

int main () {
    fin.get(s, 1002); /// citire sir care poate contine si spatii
    fin.get();

    /// parsarea lui s pentru si obtinerea sirului simplificat t
    for (i=0;s[i]!=0;i++) {
        if (s[i] == ' ')
            continue;
        if (s[i] == '(' || s[i] == ')') {
            t[k++] = s[i];
            continue;
        }
        if (s[i] == 'T' && s[i+1] == 'R') {
            t[k++] = '1';
```

```
            i+=3;
            continue;
        }
        if (s[i]== 'F' && s[i+1] == 'A') {
            t[k++] = '0';
            i+=4;
            continue;
        }
        if (s[i] == 'O' && s[i+1] == 'R') {
            t[k++] = '|';
            i++;
            continue;
        }
        if (s[i]=='A' && s[i+1] == 'N') {
            t[k++] = '&';
            i+=2;
            continue;
        }
        if (s[i] == 'N' && s[i+1] == 'O') {
            t[k++] = '1';
            t[k++] = '^';
            i+=2;
            continue;
        }
        t[k++] = s[i];
    }
    fin>>n;
    for (int j=1;j<=n;j++) {
        fin>>c;
        //F[c] = !F[c];
        F[c] = 1 - F[c]; /// se trece din 1 în 0 si din 0 in 1
        i = 0;
        fout<<expresieOr();
    }
    return 0;
}
```

**Perle (infoarena)**

Enunțul integral se găsește aici: https://www.infoarena.ro/problema/perle
Pe scurt, se dau variabilele `A`, `B`, `C` și constantele `1`, `2`, `3` precum și următoarele reguli de transformare:

`A` se poate înlocui cu un `1` sau cu un `2` sau cu un `3`
`B` se poate înlocui cu `2B` sau cu `1A3AC`
`C` se poate înlocui cu `2` sau cu `3BC` sau cu `12A`

Apoi se dă un șir format din valori 1, 2, 3 și trebuie să decidem dacă îl putem genera alegând o singură variabilă de început (dintre A, B, C) și aplicând transformări dintre cele descrise, asupra ei și a celorlalte variabile care se pot obține pe parcurs.
De exemplu șirul 2 2 2 nu se poate genera. Pentru primul 2 trebuie să alegem la început fie un A (care apoi să se transforme în 2) fie un B care apoi să se transforme în 2B.
Dacă alegem A, obținem un 2 și atât, deci nu mai putem genera celelalte valori 2. Dacă alegem 2B acoperim primul 2 și apoi al doilea și al treilea 2 îi obținem transformând B rămas în 2B.
Problema este că după ce obținem cei trei de 2 ne mai rămâne și un B care și el trebuie înlocuit cu ceva, deci nu ne putem limita la doar trei valori 2.
Un alt exemplu este șirul 2 1 1 3 2 1 2 3 pe care vom arăta cum l-am putea genera.

Pornim cu B

```
2 1 1 3 2 1 2 3
B                 îl transformăm pe B în 2B și obținem

2 1 1 3 2 1 2 3
2 B               acest B îl înlocuim cu 1A3AC și obținem

2 1 1 3 2 1 2 3
2 1 A 3 A C       Primul A îl înlocuim cu 1, al doilea a îl înlocuim cu 2 iar pe C îl înlocuim cu
12A și obținem

2 1 1 3 2 1 2 3
2 1 1 3 2 1 2 A   Acum ne rămâne să înlocuim pe A cu 3 și am terminat.
```

O analiză atentă a relațiilor ne arată că la un moment dat pe șirul de generat putem face o singură decizie. Vom scrie câte o funcție recursivă pentru fiecare variabilă și în ea plimbăm un indice global pe șirul de generat. Fiecare funcție va returna o valoare logică în funcție de reușita sau nu a generării dintr-un apel lansat din variabila corespunzătoare funcției.

```
#include <fstream>

using namespace std;

int v[10010];
int n, t, i, A(), B(), C(), T;

int A() {
    if (i==n+1)
        return 0;

    if (v[i] == 1 || v[i] == 2 || v[i] == 3) {
        i++;
        return 1;
```

```
    } else
        return 0;
}

int B() {
    if (i==n+1)
        return 0;
    if (v[i] == 2) {
        i++;
        return B();
    } else
        if (v[i] == 1) {
            i++;
            int aux = A();
            if (aux == 0)
                return 0;
            if (v[i] != 3)
                return 0;
            i++;
            aux = A();
            if (aux == 0)
                return 0;
            return C();
        } else
            return 0;
}

int C() { /// da rezultat 1 daca C poate genera restul de
          /// sir incepand cu valoarea curenta a lui i
    if (i==n+1)
        return 0;

    if (v[i] == 2) {
        i++;
        return 1;
    } else
        if (v[i] == 1) {
            i++;
            if (v[i] != 2) {
                i++;
                return 0;
            } else {
                i++;
                return A();
            }
        } else {
            i++;
            int aux = B();
            if (aux == 0)
                return 0;
            return C();
```

```
        }
}

int main (){
    ifstream fin ("perle.in");
    ofstream fout("perle.out");

    for (fin>>T; T--; ) {
        fin>>n;
        for (i=1;i<=n;i++)
            fin>>v[i];

        i = 1;
        int rez = A();
        if (rez == 1 && i == n+1) {
            fout<<"1\n";
            continue;
        }
        i = 1;
        rez = B();
        if (rez == 1 && i == n+1) {
            fout<<"1\n";
            continue;
        }
        i = 1;
        rez = C();
        if (rez == 1 && i == n+1) {
            fout<<"1\n";
            continue;
        }
        fout<<"0\n";
    }
    return 0;
}
```

O observație asupra implementării de mai sus: declararea prototipurilor funcțiilor se poate face și “toate pe același rând“ precum variabilele.

**Evaluare2 (infoarena)**

Enunțul complet se găsește aici: https://www.infoarena.ro/problema/evaluare2
Pe scurt, se cere să se evalueze o expresie care are operații pe `4` niveluri: adunare, cât al împărțirii, ridicare la putere și “oglinditul operandului”, ultimul obținându-se prin operaror unar. Special este faptul că operatorul uar și cel de ridicare la putere se aplică de la dreapta la stânga (dacă apar succesiv).
Pentru a trata aplicarea de la dreapta ne-am folosit de o funcție recursivă auxiliară (aceasta face calculele efective după autoapel și astfel aplicăm operatorii în ordine inversă).
Aceste detalii de implementare se pot observa în sursă, mai ales că aceasta este scrisă pe template-ul prezentat la problemele anterioare.

```
#include <fstream>
#include <iostream>
#define LONG long long
using namespace std;

char s[200010], t[200010];
int i;

LONG putere(LONG a, LONG b) {
    LONG r = 1;
    for (int i=1;i<=b;i++)
        r = r*a;
    return r;
}

LONG oglindit(LONG x) {
    LONG r = 0;
    while (x) {
        r = r*10 + x%10;
        x = x/10;
    }
    return r;
}

LONG expresie();
LONG termenPlus();
LONG termenCat();
LONG termenExponent();
LONG recExponent();

LONG expresie() {
    LONG r = termenPlus();
    while (s[i] == '+') {
        i++;
        r += termenPlus();
    }
    return r;
}

LONG termenPlus() {
    LONG r = termenCat();
    while (s[i] == '/') {
        i++;
        r /= termenCat();
    }
    return r;
}

LONG recExponent() {
```

```
    if (s[i] != '^')
        return 1;
    else {
        i++;
        LONG a = termenExponent();
        LONG b = recExponent();
        return putere(a, b);
    }
}

LONG termenCat() {
    LONG a = termenExponent();
    LONG b = recExponent();
    return putere(a, b);
}

LONG termenExponent() {
    if (s[i] == '!') {
        i++;
        return oglindit( termenExponent() );
    }

    LONG r = 0;

    if (s[i] == '(') {
        i++;
        r = expresie();
        i++;
    } else {
        while (s[i]>='0' && s[i]<='9') {
            r = r * 10 + s[i]-'0';
            i++;
        }
    }
    return r;
}

int main () {
    ifstream fin ("evaluare2.in");
    ofstream fout("evaluare2.out");
    fin>>s;
    fout<<expresie();
    return 0;
}
```