TEOREME de programare

Clara Ionescu

În numerele precedente ale revistei am prezentat primele două "teoreme" care ofereau modele pentru prelucrarea unui șir de date, respectiv pentru stabilirea unei proprietăți a unui element dintr-un șir sau a unei proprietăți globale a șirului. Aceste modele sunt utile deoarece respectarea lor garantează corectitudinea subalgoritmului respectiv și conferă un plus de claritate codului.

Selecția

Rezolvările problemelor **P5-P8**, prezentate în contextul teoremei de programare pe care am numit-o ***Decizie***, verifică o anumită proprietate într-un șir de date și comunică prezența sau absența ei. Analizând mai atent rezolvările, constatăm că am putea fructifica subalgoritmul, afișând în plus informații cum ar fi poziția pe care se află elementul cu proprietatea cerută. Fie în continuare problemele:

P9. Cunoscând denumirea unei luni, să se afișeze numărul ei de ordine!
P10. Să se determine cel mai mic divizor diferit de 1 al unui număr natural, strict mai mare decât 1!
P11. Cunoscând onomasticele (zilele prenumelor) dintr-un calendar, să se afișeze data la care își serbează ziua o persoană având un prenume dat!

Evident, din nou ne punem întrebarea, ce anume au în comun enunțurile de mai sus? Datele de intrare formează un șir, iar ceea ce se mai dă, este - cu siguranță - un element din șirul datelor.

Rezultă, în primul rând, că de data aceasta nu trebuie să verificăm apartenența acestei date la șirul dat, pentru că enunțul garantează acest lucru. În **P9** se dă denumirea unei *luni*, iar în **P11**, un *prenume*. Dar ce putem spune despre această "garanție" în cazul problemei **P10**? Aici șirul datelor care trebuie prelucrate (divizorii) nu trebuie citit, deoarece poate fi generat. În plus, știm că orice număr natural mai mare decât 1 (deci și numerele prime) are cel puțin doi divizori: 1 și numărul însuși. Deci, dacă numărul *n* nu se divide cu nici un număr mai mic decât *n*, dar este mai mare sau egal cu 2, totuși are un divizor diferit de 1, și anume chiar pe *n*.

Vom prezenta în continuare câteva detalii privind declarațiile din programul care va conține subalgoritmul care rezolvă o asemenea problemă:

```
n: întreg          {numărul elementelor șirului de prelucrat}
x: tablou(1..n):   tip element{elementele șirului dat}
nr_ord: întreg                                  {rezultatul}

Subalgoritm Selecția(n,x,nr_ord):
  nr_ord := 1
  Cât timp nu P(x[nr_ord]) execută
    nr_ord := nr_ord + 1
  Sfârșit cât timp
Sfârșit subalgoritm
```

Nu am verificat dacă valoarea contorului a depășit dimensiunea șirului dat, tocmai datorită faptului că enunțul garantează existența în șir a valorii căutate. Elementul cu proprietatea cerută va fi găsit cel târziu atunci când prelucrăm ultimul element.

Vom aplica această rezolvare generală pentru problemele enunțate la începutul articolului, dând implementările în limbajul *Pascal*.

Problema **P9** cere numărul de ordine al unei luni date.

```
procedure P9(n:Byte; luni:Tluni; cuv: string;
                                        var nr:Byte);
{avem definit tipul Tluni = array[1..12] of string}
begin
  nr:=1;
  while luni[nr]<>cuv do              { avansăm în șir }
    Inc(nr);          { din while am ieșit în momentul }
end;                              { în care am găsit luna dată }
```

P10. Această problemă cere determinarea celui mai mic divizor, diferit de 1, al unui număr natural. Vom prezenta o posibilă implementare în care se folosește o funcție, și nu o procedură.

Evident, și în cazul problemei precedente am fi putut lucra cu o funcție.

```
function divizor(n: Word): Word;
var i:Word;
begin
  i:=2;                          {primul divizor "permis"}
  while n mod i <> 0 do
    Inc(i);                      {următorul divizor posibil}
  divizor:=i
end;
```

Acum, ne putem întreba dacă este sau nu cazul să "generăm" valori pentru divizorii "posibili" care sunt mai mari decât rădăcina pătrată a numărului dat? Evident, ***nu***! Rezultă că în condiția din **while**, de data aceasta este indicat să adăugăm și subexpresia `(i ≤ rad)`, unde valoarea `rad` trebuie calculată înaintea ciclului, pentru a nu cere evaluarea ei de fiecare dată când acesta se reia. Va trebui să fim atenți pentru cazul în care numărul *n* este prim. Dacă nu a fost găsit nici un divizor mai mic decât rădăcina pătrată a lui *n*, atunci *n* este număr prim, deci rezultatul va fi *n*.

P11. Vom determina data din an (luna și ziua) la care o persoană, având prenumele cunoscut, își va serba ziua numelui. Calendarul constituie șirul *x* și presupunem că acesta cuprinde *n* articole. Articolele vor avea structura prezentată mai jos:

```
type onomastica=record
                  prenume:string;
                  luna,ziua:Byte
                end;
     calendar=array[1..MaxN] of onomastica;
var x:calendar; n:Byte;
    pren:string; zi:onomastica;

procedure P11(n:Byte;x:calendar;pren:string;
                              var zi:onomastica);
var i:Byte;
begin
  i:=1;
  while x[i].prenume<>pren do Inc(i);
  zi:=x[i]
end;
```

În această implementare nu am putut respecta fidel modelul dat pentru selecție, deoarece a apărut o diferență: nu numărul de ordine al unui element din șirul dat ne-a interesat, ci câmpurile acestuia. Cu toate că ar fi trebuit să păstrăm doar valorile cîmpurilor care diferă de cel a cărui valoare o cunoșteam în prealabil, în procedură, am păstrat în parametrul de ieșire `zi` întregul element de tip **record**, pentru a nu lungi lista parametrilor.

Se observă că acest model de rezolvare conduce la găsirea primului element, având proprietatea cerută. Dacă ne interesează ultimul astfel de element vom parcurge șirul în sens invers, adică începând cu ultimul și "avansând" spre primul.

Căutarea (secvențială)

Problema **P6** a fost rezolvată folosind modelul oferit de ***Decizie***. Aceasta era, de fapt, o simplă căutare (un cuvânt dat este sau nu denumirea unei luni din an?). Acolo era suficient să răspundem la întrebări de genul "*există*?". Revenim, tratând tipul de problemă în care cele două cerințe din ultimele două clase de probleme se combină, respectiv, după aflarea răspunsului la întrebarea "*există*?", afișăm, de exemplu, poziția pe care se află elementul căutat. Cu alte cuvinte, în această clasă intră probleme care nu garantează existența elementului căutat în șirul dat.

P12. Se consideră valorile intrărilor și ieșirilor sumelor de bani care reprezintă încasările și plățile efectuate de către o firmă pe parcursul unei luni. În cazul în care există o zi cu pierderi, să se afișeze această zi! (Deci, se cere ziua în care plățile au fost mai mari decît încasările.)

Evident, modelul de rezolvare se bazează pe modelele prezentate până acum.

```
Subalgoritm Căutare(n,x,găsit,nr_ord):
  i:=1
  Cât timp (i<=n) şi nu (P(x[i]) execută i:=i+1
  Sfârşit cât timp
  găsit:=i<=n
  Dacă găsit atunci nr_ord:=i
Sfârşit subalgoritm
```

Se observă că soluția pornește cu modelul ***Deciziei*** care se completează cu determinarea rezultatului conform ***Selecției***.

În rezolvarea problemei **P12**, presupunem că `int` și `ies` sunt două tablouri având numărul de elemente egal cu numărul zilelor din luna prelucrată, deci în variabila `ziua` vom păstra indicele primei zile din lună în care firma a avut pierderi, dacă există o asemenea zi. Evident, înainte de afișare, în unitatea de program apelantă vom testa valoarea variabilei `găsit`, transmisă ca parametru de ieșire (prin referință):

```
procedure P12(n:Byte; int,ies:tablou;
           var gasit:Boolean; var ziua:Byte);
var i:Byte;
begin
  i:=1;
  while (i<=n) and (int[i]>=ies[i]) do
    Inc(i);
  gasit:=i<=n;
  if gasit then ziua:=i
end;
```

Din nou, dacă ne interesează ultima zi cu pierderi, vom parcurge tablourile în sens invers.

În următorul articol vom vedea cum procedăm dacă dorim să determinăm toate elementele unui șir dat, având o proprietate dată.