4 Instrucţiunea condiţională

4.1 Decizie şi secvenţiere, expresii şi instrucţiuni

Am văzut că *secvenţierea* (prin înşiruirea instrucţiunilor în corpul unei funcţii) şi *decizia* (prin folosirea expresiei condiţionale) sunt, alături de *recursivitate*, elementele de limbaj care ne permit scrierea de programe cu logică şi prelucrări complexe pornind de la expresii şi instrucţiuni elementare.

Remarcăm că există o corespondenţă directă între *expresii* (corespunzând calculelor, inclusiv apelul de funcţie), şi *instrucţiuni*: înainte de instrucţiunea `return` care precizează rezultatul, în funcţii pot apare *instrucţiuni expresie*, de forma *expresie* `;` . Efectul lor e *evaluarea* expresiilor respective, de regulă apeluri de funcţie (`printf`, `putchar`), pentru efectele produse de acestea.

După citirea unui număr natural în baza 10, încercăm să scriem o funcţie de tipărire caracter cu caracter, reproducând astfel funcţionalitatea oferită de `printf`. Structura recursivă a problemei e aceeaşi: se separară ultima cifră, precedată de un număr cu mai puţine cifre (exemplu: $2475 = 247 \cdot 10 + 5$).

$$\text{tipăreşte } n\text{:} \begin{cases} \text{dacă } n < 10 & \text{tipăreşte unica cifră din } n \\ \text{altfel} & \text{tipăreşte } n/10\text{; tipăreşte ultima cifră din } n \end{cases}$$

Încercând să implementăm direct acestă descriere apar următoarele probleme:
– până acum, singurul element de limbaj pentru *decizie* e *expresia* condiţională, care necesită pe ambele ramuri *expresii* de acelaşi tip; formularea noastră însă conţine tipărirea, o acţiune (prelucrare) care se traduce printr-o *instrucţiune*.
– chiar ţinând cont că tipărirea se face de fapt tot cu o expresie (un apel de funcţie fără a folosi rezultatul returnat), ultimul caz conţine *două* tipăriri succesive, iar expresia condiţională permite *o singură* expresie pe fiecare ramură.

Putem găsi o soluţie cu elementele de limbaj descrise până acum: grupăm tipăririle de pe a doua ramură într-o funcţie auxiliară cu valoare întreagă (la fel ca `putchar`), al cărei rezultat e folosit apoi în expresia condiţională.

O astfel de soluţie e însă complicată, şi reflectă asimetria în limbaj între expresii şi instrucţiuni: avem nevoie fie de *secvenţiere* pentru expresii (posibilitatea de a scrie sintactic în succesiune mai multe expresii, evaluate pe rând, cu rezultatul dat de ultima), sau de o structură de *decizie* pentru instrucţiuni.

Unele limbaje, îndeosebi cele funcţionale, nu definesc explicit instrucţiuni, ci doar expresii, având astfel o singură construcţie sintactică pentru secvenţiere şi alta pentru decizie. În C, secvenţierea şi decizia se scriu diferit pentru expresii şi instrucţiuni, şi acestea din urmă sunt folosite predominant, ceea ce dă caracterul *imperativ* al limbajului.

4.2 Instrucţiunea condiţională `if`

Instrucţiunea condiţională, numită şi după cuvântul cheie `if` permite selecţia între execuţia a două instrucţiuni, după valoarea unei expresii numită *condiţie*. Ea are două variante sintactice; în varianta completă, cu două ramuri:

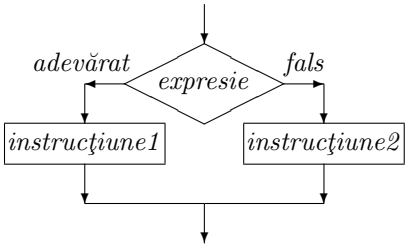
```
if ( expresie )
  instrucţiune1
else
  instrucţiune2
```

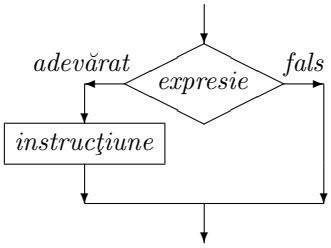
Instrucţiunea evaluează întâi expresia. Dacă aceasta e nenulă (adevărată), se execută prima instrucţiune componentă, altfel (dacă expresia are valoare zero, adică falsă) se execută a doua instrucţiune.

Instrucţiunea are şi o formă scurtă, fără clauza introdusă de `else`:

```
if ( expresie )
  instrucţiune1
```

Ea are acelaşi efect ca şi prima formă cu a doua instrucţiune vidă: dacă expresia are valoare nenulă se execută *instrucţiune*. În ambele forme, după execuţia ramurii selectate, se continuă cu instrucţiunea următoare din program. Fluxul de control pentru cele două forme ale instrucţiunii poate fi reprezentat prin *schema logică* din figură:





Expresia care controlează decizia poate fi orice expresie *scalară*, adică de tip întreg, real, sau adresă (care va fi discutat ulterior).

Sintactic, *parantezele* `( )` fac parte din forma instrucţiunii şi sunt obligatorii, chiar dacă expresia încadrată este simplă. Pe ambele ramuri e permisă câte *o singură* instrucţiune; dacă logica programului impune mai multe prelucrări, acestea trebuie grupate între acolade `{ }` într-o *instrucţiune compusă*. Remarcăm de asemenea că definiţia instrucţiunii `if` nu conţine nici `;` nici acolade, ele pot să apară însă ca parte din instrucţiunile de pe cele două ramuri.

Instrucţiunea `if` e corespondentul pentru instrucţiuni al expresiei condiţionale: cele două au în comun decizia luată pe baza evaluării expresiei, dar diferă conţinutul celor două ramuri: expresii de evaluat (obligatoriu de acelaşi tip, care devine cel al rezultatului), respectiv instrucţiuni de executat.

De exemplu, putem scrie o funcţie pentru afişarea soluţiilor unei ecuaţii de gradul II, în funcţie de valoarea discriminantului:

```
void printsol(double a, double b, double delta)
{
  if (a == 0) printf("Ecuatia nu e de gradul doi!");
  else if (delta >= 0) {
    printf("Solutia 1%f\n", (-b-sqrt(delta))/2/a);
    printf("Solutia 2%f\n", (-b+sqrt(delta))/2/a);
  } else printf("Nu are solutie reala\n");
}
```

În acest exemplu, ramura corespunzătoare valorii "adevărat" a condiţiei conţine o instrucţiune compusă formată din două instrucţiuni, iar ramura "else" conţine o singură instrucţiune.

Scriem ca alt exemplu o funcţie care returnează valoarea (de la 0 la 15) corespunzătoare cifrei hexazecimale date ca argument, sau -1 pentru orice alt caracter. Dacă parametrul e într-adevăr cifră hexazecimală, urmează o a doua decizie, după cum caracterul e cifră sau nu (deci literă); ultimul caz e tratat uniform folosind convertirea la litere mici. Altfel, funcţia returnează -1.

```
#include <ctype.h>

int xdigitvalue(int c)
{
  if (isxdigit(c))      // e cifra hexa ?
    if (isdigit(c)) return c - '0';     // e cifra
    else return tolower(c) - 'a' + 10;  // litera hexa
  else return -1;       // orice altceva
}
```

Acest exemplu arată că deciziile pot fi încuibate, rezultând structuri logice complexe. În acest caz, prima clauză (*instrucţiune1*) a primei instrucţiuni `if` e la rândul ei o instrucţiune condiţională, rezultând în total trei ramuri de execuţie prin funcţie, pe fiecare aflându-se o instrucţiune `return`. Aceasta încheie execuţia funcţiei, chiar dacă în text ar mai fi urmat altă instrucţiune.

Acelaşi efect se obţine scriind corpul funcţiei cu expresia condiţională:

```
  return isxdigit(c) ? isdigit(c) ? c-'0' : tolower(c)-'a' + 10
                     : -1;
```

Comparând expresia şi instrucţiunea condiţională prima observaţie e că, evident, pe cele două ramuri se află într-un caz *expresii* de acelaşi tip, iar în celălalt caz, *instrucţiuni*. În al doilea rând,

operatorul `? :` are *obligatoriu* câte o expresie pentru cele două ramuri, pe când instrucţiunea `if` există şi în forma scurtă fără `else`, care specifică o instrucţiune de executat doar când condiţia e adevărată.

Această diferenţă devine importantă când instrucţiunea încheie o funcţie, ca mai sus. O funcţie care returnează o valoare (are alt tip decât `void`) trebuie să-şi încheie execuţia în toate cazurile cu un rezultat bine definit. Orice cale de execuţie prin funcţie trebuie să ajungă la o instrucţiune `return` *expresie* `;` şi este o eroare dacă, dimpotrivă, se ajunge la acolada de sfârşit `}`, comportamentul în momentul folosirii rezultatului necunoscut al apelului fiind nedefinit. De exemplu, absenţa ultimei clauze `else return -1;` ar fi fost incorectă, lăsând rezultatul nedefinit în cazul unui parametru care nu e cifră hexazecimală.

Compilatorul poate detecta astfel de situaţii şi genera un mesaj de avertisment. Deşi mult mai puţin folosită, expresia condiţională e mai sigură aici: sintaxa ei *obligă* programatorul să specifice valori pe ambele ramuri. Pe de altă parte, combinarea mai multor operatori `? :` devine greu de citit, şi de aceea e preferabilă folosirea instrucţiunii `if`.

Punerea în pagină a funcţiei ilustrează nişte convenţii de structurare pentru citirea mai uşoară a programelor. Codul de pe ramurile deciziei se scrie cu un rând mai jos, *indentat* (de regulă cu două spaţii) spre interior faţă de cuvântul cheie `if` sau `else`. Dacă ramura e o singură instrucţiune, foarte scurtă ca text, se poate plasa mai compact pe acelaşi rând cu `if`, respectiv `else`.

Folosirea deciziilor complexe pune problema asocierii corecte între ramurile acestora. De exemplu, următoarea structură ar putea fi citită în două feluri:

`if (`*expr1*`) if (`*expr2*`)` *instr1* `else` *instr2*

O variantă e să considerăm primul `if` ca formă scurtă, având ca ramură de condiţie adevărată o instrucţiune `if` completă. A doua interpretare e cu primul `if` în formă completă, ramura sa adevărată fiind în formă scurtă, fără `else`.

Standardul prevede prima interpretare: un `else` e întotdeauna *asociat cu cel mai apropiat* `if` permis de sintaxă. Dacă dorim a doua variantă, trebuie să separăm al doilea `if` de ramura `else` care nu-i aparţine, încadrându-l într-un bloc, ca în partea dreaptă a figurii. Asocierea implicită e echivalentă cu cea din stânga figurii. Deşi acoladele nu sunt necesare aici, adăugarea lor clarifică structura şi evită confuziile la modificarea ulterioară a programului.

```
if (expr1) {                    if (expr1) {
  if (expr2) instr1               if (expr2) instr1
  else instr2                   } else instr2
}
```

a) asocierea implicită b) asociere modificată cu `{ }`

Un alt mod de a reţine şi explica această regulă e că se alege asocierea care ar putea fi extinsă, şi astfel echilibrată, cu încă o clauză `else` dacă aceasta ar urma imediat în program. Se observă că acest lucru e posibil cu asocierea din stânga figurii (a celui mai apropiat `if`), dar nu şi cu cea din dreapta.

Indentarea, adesea sugerată de editoarele pentru scrierea de programe, ne ajută la vizualizarea structurii codului, dar nu modifică regulile sintaxei (spaţiile suplimentare sunt ignorate de compilator). În speţă, ea nu poate schimba asocierea `else` – `if` în exemplul de mai sus, şi alinierea instrucţiunilor la acelaşi nivel una sub alta nu poate substitui gruparea lor într-un bloc.

```
if (expr)                       if (expr)
  instr1                          instr1
else            e de fapt       else
  instr2                          instr2
  instr3                        instr3
```

Ca element de programare defensivă se recomandă folosirea instrucţiunilor compuse pe ambele ramuri ale unei decizii, chiar când ele conţin doar câte o instrucţiune, pentru a evidenţia mai bine structura codului şi a evita erori la adăugiri ulterioare. De exemplu, o instrucţiune scrisă după unica instrucţiune dintr-o clauză `else` nu mai aparţine sintactic de aceasta, ci e o instrucţiune independentă şi se execută după instrucţiunea `if`, indiferent de sensul deciziei.

Revenim la problema tipăririi caracter cu caracter a unui număr în baza 10. Instrucţiunea `if` ne permite traducerea directă a definiţiei recursive: apelul recursiv pentru $n/10$ trebuie făcut doar pentru numere de mai multe cifre.

Efectul dorit al funcţiei este tipărirea, ea nu efectuează vreun calcul şi nu trebuie să returneze un rezultat. În consecinţă, ea poate fi declarată cu tipul `void` (tipul vid, fără valori), semnificând că nu returnează o valoare.

```
#include <stdio.h>
void printnat(unsigned n)
{
  if (n > 9) printnat(n / 10);  // ramura recursiva
  putchar('0' + n % 10);        // cazul de baza: ultima cifra
}
```

Execuţia funcţiei se încheie cu ultima instrucţiune din corpul ei, fără a fi necesară instrucţiunea `return`. Ea poate fi folosită însă (în forma `return;` fără expresie, funcţia neavând rezultat) acolo unde se doreşte revenirea din funcţie fără a mai continua până la sfârşitul corpului ei.

4.3 Operatori logici

În expresii condiţionale şi instrucţiunea `if` am folosit până acum expresii simple, mai precis, cu cel mult un operator de comparare: `<=`, `==`, etc. Adesea, problemele necesită combinaţii logice mai complicate. De exemplu:

un an e bisect dacă: se divide cu 4 **şi** **nu** se divide cu 100 **sau** se divide cu 400

După această regulă, 1800 şi 1900 nu au fost ani bisecţi, dar 2000 da.

Putem traduce această definiţie într-o funcţie C după cum urmează:

```
int e_bisect(unsigned an)
{
  return an % 4 == 0 && (!(an % 100 == 0) || an % 400 == 0);
}
```

Atât definiţia dată în cuvinte cât şi funcţia în C conţin trei operatori logici: *negaţia* (nu, `!`), *conjuncţia* (şi, `&&`), şi *disjuncţia* (sau, `||`). Reamintim că în C restul modulo un număr se obţine cu operatorul `%` .

Operatorii logici respectă convenţia prezentată pentru expresii logice în limbajul C: ei produc o valoare întreagă, 1 pentru *adevărat* şi 0 pentru *fals*, ceea ce explică tipul `int` al funcţiei. Ca argumente, ei *acceptă* (ca peste tot în C unde se solicită o condiţie) orice expresii *scalare* (întregi, reali, adrese), o valoare nenulă însemnând *adevărat*, iar una nulă *fals*. Tabelele de adevăr pentru cei trei operatori sunt:

expr	`!` *expr*
0	1
$\neq 0$	0

a) negaţie

	e1 `&&` *e2*	*e2* = 0	*e2* $\neq 0$
e1	0	0	0
	$\neq 0$	0	1

b) conjuncţie

	e1 `||` *e2*	*e2* = 0	*e2* $\neq 0$
e1	0	0	1
	$\neq 0$	1	1

c) disjuncţie

Pe scurt, operatorul `&&` (*ŞI*) produce un rezultat adevărat (1) numai dacă ambii operanzi sunt adevăraţi (nenuli), pe când operatorul `||` (*SAU*) dă ca rezultat 1 atunci când e nenul cel puţin unul din operanzi (inclusiv amândoi).

Precedenţa operatorilor Exemplul dat evidenţiază câteva aspecte privind precedenţa operatorilor logici în raport cu cei deja cunoscuţi. Operatorul unar `!` are precedenţă mai mare decât orice operator binar, necesitând paranteze în expresia `!(an % 100 == 0)`. Operatorii logici binari au precedenţă mai mică decât operatorii relaţionali şi de comparaţie, deci nu e nevoie de paranteze pentru `an % 4 == 0` ca prim operand al lui `&&` şi pentru al doilea operand al lui `||`. Conjuncţia `&&` are precedenţă mai mare decât disjuncţia `||` (de altfel, în circuite logice notaţia foloseşte semnele de înmulţire, respectiv

adunare). Sunt necesare deci paranteze în jurul subexpresiei cu `||` folosită ca operandul al doilea pentru `&&` . Se recomandă folosirea de paranteze pentru subexpresii şi dincolo de minimul necesar, dacă aceasta duce la un cod mai uşor de înţeles.

Condiţia "nu se divide cu 100" se poate scrie direct `an % 100 != 0` fără a mai folosi negaţia. Expresiile `!(`*expr1*`==`*expr2*`)` şi *expr1*`!=`*expr2* sunt echivalente. Cum o expresie e adevărată dacă e nenulă, când se cer expresii logice putem omite comparaţia cu 0, scriind *expr* în loc de *expr*`!=0` şi `!`*expr* în loc de *expr*`==0`. Putem scrie (cu paranteze dictate de precedenţă)

```
!(an % 4) && (an % 100 || !(an % 400))
```

Totuşi, comparaţia explicită cu 0 e adeseori mai clar de citit, şi nu trebuie să sacrificăm claritatea pentru câteva caractere mai puţin de scris. Un compilator bun va genera cod la fel de eficient în ambele cazuri.

Corpul funcţiei date a fost scris cel mai concis cu o expresie condiţională; decizia putea fi făcută şi cu o instrucţiune `if`, iar operatorii logici se folosesc la fel în oricare context:

```
if (an % 4 == 0 && (an % 100 != 0 || an % 400 == 0)) return 1;
else return 0;
```

Evaluarea în scurt-circuit Un aspect foarte important privind operatorii logici binari în limbajul C e ordinea şi modul de evaluare a argumentelor.

În C, *ordinea* de evaluare a operanzilor într-o expresie e nespecificată (poate fi aleasă de implementarea compilatorului). Excepţiile discutate până acum sunt operatorul condiţional `? :` şi cel secvenţial `,` care evaluează întâi *primul* operand, şi aplică toate efectele laterale *înainte* de a-l evalua pe următorul. Ceilalţi operatori binari (aritmetici, relaţionali, de comparaţie) îşi pot evalua operanzii în orice ordine, iar momentul şi ordinea de aplicare a efectelor laterale sunt de asemenea nespecificate (aspect exemplificat în detaliu mai târziu).

Operatorii `&&` şi `||` sunt celelalte cazuri speciale care îsi evaluează strict în ordine operanzii. Mai mult, ei nu evaluează al doilea operand *decât dacă acesta mai poate influenţa valoarea expresiei*. Din tabelul de adevăr pentru `&&`, observăm că dacă primul operand e 0, rezultatul e 0 (fals) *indiferent* de valoarea celui de-al doilea operand. Similar, pentru `||`, dacă primul operand e nenul, rezultatul e 1 (adevărat), independent de valoarea celui de-al doilea. În aceste cazuri, operandul al doilea nu se mai evaluează.

Această regulă, aplicată în C şi alte limbaje inspirate din acesta, se numeşte şi *evaluare în scurt-circuit*. Ea face programele mai eficiente, evitând evaluări inutile. În exemplul anterior, dacă anul nu se divide la 4, primul argument al lui `&&` e fals, şi nu se mai evaluează al doilea: anul sigur nu e bisect.

Mai important, evaluarea în scurt-circuit trebuie luată în calcul cu atenţie pentru a ne asigura că programul e corect. De exemplu, în scrierea

```
if (p != 0 && n % p == 0) // urmeaza o instructiune
```

ne asigurăm că restul lui `n` modulo `p` va fi evaluat doar dacă `p != 0`. Scrierea testului inversând operanzii lui `&&` este potenţial incorectă: riscăm o împărţire la zero înainte ca `p` să fie testat. Deci, operatorii `&&` şi `||` *nu sunt simetrici* în limbajul C, ceea ce nu e evidenţiat în tabelele de adevăr.

Exemplu: inversarea unei linii de text Scriem o funcţie simplă care citeşte de la intrare o linie de text şi îi afişează caracterele în ordine inversă. Funcţia trebuie să se oprească şi dacă sfârşitul intrării (`EOF`) intervine fără să apară caracterul `\n`, de aceea condiţia de apel recursiv e compusă. Dacă linia n-a luat sfârşit, funcţia inversează întâi restul liniei, după care tipăreşte caracterul curent.

```
#include <stdio.h>
void revline(int c)
{
  if (c != EOF && c != '\n') {
    revline(getchar());
    putchar(c);
  }
}

int main(void)
{
  revline(getchar());
  putchar('\n');
  return 0;
}
```