Recursivitate

18 decembrie 2002

Recursivitatea: Noţiuni fundamentale

Recursivitatea e un concept fundamental în matematică şi informatică. Un obiect (o noţiune) e recursiv(ă) dacă e folosit în propria sa definiţie.

Exemplu din matematică: şiruri recurente
– progresie aritmetică: $x_0 = a$, $x_n = x_{n-1} + p$, pentru $n > 0$
– şirul lui Fibonacci: $F_0 = 1$, $F_1 = 1$, $F_n = F_{n-1} + F_{n-2}$ pentru $n > 1$.

Recursivitatea în limbajul C:
– funcţii recursive: o funcţie `f` care se apelează pe sine însuşi (direct) sau indirect, ex. `f` apelează pe `g`, `g` apelează pe `h`, iar `h` pe `f`
– tipuri de date recursive (tip structură cu câmp pointer la acel tip)

```
typedef struct l {
  int info;    /* sau mai multe câmpuri cu diverse date */
  struct l *next;    /* pointer la acelaşi tip */
} list;        /* definim list ca nume de tip pentru struct l */
```

Recursivitate și inducție

O definiție recursivă trebuie să fie *bine formată*
– o noțiune nu se poate defini *doar* în funcție de sine însuși ($x = x$)
– o definiție recursivă se poate folosi *doar* de noțiuni deja definite
NU: $x_n = x_{n+1} - 1$, pentru $n \geq 0$.
$\Rightarrow$ orice șir de apeluri de funcții recursive trebuie să se oprească (nu va genera un calcul infinit)

În general, distingem:
– un *caz de bază* (pentru care noțiunea e definită direct) (ex. $a^0 = 1$)
– un *pas inductiv* (recursivitatea propriu-zisă) (ex. $a^{n+1} = a^n * a$)

Cu principiul inducției, demonstrăm $P(n+1)$ știind $P(i)$ pentru $i \leq n$
La fel, pentru o definiție recursivă, stabilim o *măsură* nenegativă (de ex. *indicele* în șiruri recurente) care descrește pe măsură ce expandăm definiția $\Rightarrow$ ne asigurăm că recursivitatea se va *opri* la cazul de bază

Recursivitate și iterație

Recursivitatea și iterația sunt noțiuni strâns legate.

Orice program recursiv poate fi transformat *mecanic* într-unul care utilizează doar iterația (pe același principiu folosit de compilator la generarea de cod pentru apeluri de funcții, folosind o stivă)

Invers, semantica unui ciclu (cu test inițial) poate fi definită recursiv:

```
while ( cond )
   instrucțiune;
```

```
if ( cond ) {
   instrucțiune;
   while ( cond )
      instrucțiune;
}
```

– în general, codul scris folosind doar iterația e mai eficient
– dar o soluția recursivă e adesea cea mai simplă, elegantă și naturală

Exemplu: funcţia factorial

```
int fact(int n)
{
  if (n == 0) return 1;
  else return n * fact(n-1);
}
```

```
int fact_nonrec(int n)
{
  int p = 1;
  while (n > 0) {
    p = p * n;
    n = n - 1;
  }
  return p;
}
```

$\Rightarrow$ transcriere simplă dintr-o variantă in alta; în cea recursivă, valoarea factorialului se acumulează automat în expresia returnată (în varianta recursivă, e necesară variabila `p`)

Atenţie la condiţiile limită ! `fact` va cicla infinit pentru `n < 0` !

Exemplu: şirul Fibonacci

```
int fib(int n)
{
  if (n <= 1) return n >= 0;
  /* vrem 0 pt. n negativ */
  else return fib(n-1)+fib(n-2);
}
```

```
int fib_nonrec(int n)
{
  int i, *f, res;
  if (n <= 1) return n >= 0;
  f = malloc(n * sizeof(int));
  f[1] = f[0] = 1;
  for (i = 2; i < n; i++)
    f[i] = f[i-1] + f[i-2];
  res = f[n-1] + f[n-2];
  free(f);
  return res;
}
```

Atenţie ! Traducerea directă a formulei rezultă în apeluri redundante ! (se apelează din nou `fib(n-2)` şi în `fib(n-1)`, etc.)
$\Rightarrow$ nr. de apeluri (câte pt. fib(5)?) e exponenţial în `n` (f. ineficient)
Obs. completaţi `fib_nonrec` pt. a funcţiona corect pt. `n <= 1`

Recursivitate: evitarea redundanței

– prin memorarea valorilor intermediare necesare
  – calcul ordonat ca rezultatele intermediare să fie disponibile când sunt necesare (rezolvare de jos în sus, `fib_nonrec`)
  – calculul valorilor după cum devin necesare (de sus în jos)

```
#define MAX   100
int f[MAX] = {1, 1}; /* restul zero */
int fib_memo(int n)  /* doar pentru 0 <= n < MAX */
{
  if (f[n]) return f[n]; /* nenul -> deja calculat */
  else return f[n] = fib_memo(n-1) + fib_memo(n-2);
  /* memorăm în f[n] înainte de a returna valoarea */
}
```

Câte apeluri se efectuează pentru `fib_memo(5)` ?

Economisirea de apeluri pt. cazul terminal

Adesea, cazul de bază e f. simplu (ex. test și returnarea unei valori).
In comparație, costul unui apel de funcție poate fi semnificativ.
⇒ Putem trata cazul de bază fără a mai face un apel suplimentar.

```
#define MAX   100
int f[MAX+1] = {1, 1}; /* restul zero */
int fib_r(int n)  /* pentru n >= 2 */
{
  return f[n]=(f[n-1]?f[n-1]:fib_r(n-1))+(f[n-2]?f[n-2]:fib_r(n-2));
  /* testăm f[k] pt. a decide dacă să apelăm recursiv sau nu */
}
int fib_main(int n) /* se apelează de utilizator */
{
  if (n <= 1) return n >= 0;
  else if (n > MAX) return -1; /* eroare */
  else return fib_r(n);
}
```

Descompunerea în subprobleme

Una din principalele aplicaţii ale recursivităţii: rezolvarea unei probleme prin descompunerea în subprobleme mai mici.

Exemplu: Să se genereze toate şirurile de n cifre binare (în total 2^n)
Definiţie recursivă: $n-1$ cifre binare, urmate de 0 sau 1
(descompunere în două subprobleme mai mici)

```
#define N 10
char s[N+1];
s[N] = '\0';
bitstrings(s+N, N);
```

```
void bitstrings(char *s, int n) {
  if (n == 0) puts(s);
  else { --s;
    *s = '0'; bitstrings(s, n-1);
    *s = '1'; bitstrings(s, n-1);
  } /* completează de la coadă */
}
```

Eliminarea recursivității

Exemplu: `puts`: tipărirea unui șir de caractere, urmat de '\n'
– pentru șirul vid ('\0'), tipărește '\n'
– altfel, tipărește primul caracter, tipărește restul șirului

```
void puts(char *s)
{
  if (*s) {
    putchar(*s);
    puts(s+1);
  } else putchar('\n');
}
```

```
void puts(char *s)
{
  while (*s) {
    putchar(*s);
    s = s+1;
  }
  putchar('\n');
}
```

În cazul recursivității la dreapta (apelul recursiv este ultimul lucru)
– transformăm în buclă, cu aceeași condiție de continuare
– actualizăm variabilele cu valorile argumentelor din apelul recursiv

Eliminarea recursivității

Poate fi necesară rescrierea, pentru a aduce apelul recursiv la sfârșit
Ex. factorialul, cu parametru suplimentar produsul acumulat deja

```
int fact_prod(int n, int p)
{
  if (n > 0) {
    p = p * n;
    return fact_prod(n-1, p);
  } else return p;
}
/* apelat cu fact_prod(n, 1) */
```

```
int fact_nonrec(int n)
{
  int p = 1;
  while (n > 0) {
    p = p * n;
    n = n - 1;
  }
  return p;
}
```

Pentru mai mult de un apel recursiv, e necesară folosirea unei *stive*