CURS 1

Introducere în programarea declarativă. Recursivitate

Cuprins

Bibliografie 1
1. Programare şi limbaje de programare 1
2. Recursivitate 2
   2.1 Exemple recursivitate 3

Bibliografie

Capitolul 1, Czibula, G., Pop, H.F., *Elemente avansate de programare în Lisp şi Prolog. Aplicaţii în Inteligenţa Artificială.*, Ed. Albastră, Cluj-Napoca, 2012

1. Programare şi limbaje de programare

➢ Limbaje

- **Procedurale (imperative) – limbaje de nivel înalt**
  - Fortran, Cobol, Algol, Pascal, C,...
  - program – secvenţă de instrucţiuni
  - instrucţiunea de atribuire, structuri de control – pentru controlul execuţiei secvenţiale, ramificării şi ciclării.
  - rolul programatorului – **"ce" şi "cum"**
    1. să descrie **CE** e de calculat
    2. să organizeze calculul **CUM**
    3. să organizeze gestionarea memoriei
  - !!! se susţine că instrucţiunea de atribuire este periculoasă în limbajele de nivel înalt, aşa cum instrucţiunea GO TO a fost considerată periculoasă pentru programarea structurată în anii '68.

- **Declarative (descriptive, aplicative) – limbaje de nivel foarte înalt**
  - se bazează pe expresii
  - expresive, uşor de înţeles (au o bază simplă), extensibile

- programele pot fi văzute ca descrieri care declară informaţii despre valori, mai degrabă decât instrucţiuni pentru determinarea valorilor sau efectelor.
- renunță la instrucțiuni
    1. protejează utilizatorii de la a face prea multe erori
    2. sunt generate din principii matematice - analiza, proiectarea, specificarea, implementarea, abstractizarea şi raţionarea (deducţii ale consecinţelor şi proprietăţilor) devin activităţi din ce în ce mai formale.
- rolul programatorului - "**ce**" (nu "**cum**")
- două clase de limbaje declarative
    1. **limbajele funcționale** (de exemplu Lisp, ML, Scheme, Haskell, Erlang)
        - se focalizează pe valori ale datelor descrise prin expresii (construite prin aplicări ale funcţiilor şi definiţii de funcţii), cu evaluare automată a expresiilor
    2. **limbaje logice** (de exemplu Prolog, Datalog, Parlog), care se focalizează pe aserţiuni logice care descriu relaţiile dintre valorile datelor şi derivări automate de răspunsuri la întrebări, plecând de la aceste aserţiuni.
- aplicații în Inteligența Artificială – demonstrarea automată, procesarea limbajului natural și înțelegerea vorbirii, sisteme expert, învățare automată, agenți, etc.

- Limbaje multiparadigmă: **F#, Python, Scala** (imperativ, funcțional, orientat obiect)
- Interacțiuni între limbajele declarative și cele imperative – limbaje declarative care oferă interfețe cu limbaje imperative (ex C, Java): SWI-Prolog, GNUProlog, etc.
- **Logtalk** – integrează logica și OOP.
- Programare logică în Python
    - **Karen**
    - **SymPy** – bibliotecă pentru calcul simbolic

2. Recursivitate

- mecanism general de elaborare a programelor.
- recursivitatea a apărut din necesităţi practice (transcrierea directă a formulelor matematice recursive; vezi funcţia lui Ackermann)
- recursivitatea este acel mecanism prin care un subprogram (funcţie, procedură) se autoapelează.
    - două tipuri de recursivitate: **directă** sau **indirectă**.
- **!!! Rezultat**

- orice funcție calculabilă poate fi exprimată (deci și programată) în termeni de funcții recursive
- două lucruri de considerat în descrierea unui algoritm recursiv: **regula recursivă** și **condiția de ieşire din recursivitate**.
- **avantaj** al recursivității: text sursă extrem de scurt și foarte clar.
- **dezavantaj** al recursivității: umplerea segmentului de stivă în cazul în care numărul apelurilor recursive, respectiv ai parametrilor formali și locali ai subprogramelor recursive este mare.
  - în limbajele declarative există mecanisme specifice de optimizare a recursivității (vezi mecanismul recursivității de coadă în Prolog).

2.1 Exemple recursivitate

Notații

- o listă este o secvență de elemente ($l_1 l_2 \dots l_n$)
- lista vidă (cu 0 elemente) o notăm cu $\emptyset$
- prin $\oplus$ notăm operația care adaugă un element în listă

1. Să se creeze lista (1,2,3,...n)

a) direct recursiv

$$creareLista(n) = \begin{cases} \emptyset & daca\ n = 0 \\ creareLista(n-1) \oplus n & altfel \end{cases}$$

b) folosind o funcție auxiliară recursivă pentru crearea sublistei (*i*, *i*+1,..., n)

// crearea listei formată din elementele *i*, *i*+1,…, *n*

Model matematic recursiv

$$creare(i,n) = \begin{cases} \phi & daca\ i > n \\ i \oplus creare(i+1,n) & altfel \end{cases}$$

// crearea listei formată din elementele 1, 2,…, *n*

$$creareLista(n) = creare(1,n)$$

Pseudocod

Alegerea representării: reprezentare simplu înlănțuită, cu alocare dinamică a nodurilor.

<u>NodLSI</u>

e: TElement //infomația utilă a nodului
urm: ↑NodLSI //adresa la care e memorat următorul nod

<u>LSI</u>

prim: ↑NodLSI //adresa primului nod din listă

Funcția creeazaNodLSI(*e*)
*{**pre**: e:* TElement*}*
*{**post**: se returnează un* ↑NodLSI *conținând e ca informație utilă}*
{se alocă un spațiu de memorare pentru un NodLSI *}*
{*p*: ↑NodLSI}
aloca(*p*)
[*p*].e ← e
[*p*].urm ← NIL
{rezultatul returnat de funcție}
creeazaNodLSI ← *p*
SfFuncție

Funcția creare(*i*, *n*)
*{**post**: se returnează un* ↑NodLSI*, pointer spre capul listei înlănțuite formate }*
{ din elementele i, i+1,..., n }
Dacă *i* > *n* **atunci**
creare ← NIL
altfel
{ se alocă un spațiu de memorare pentru un NodLSI *având }*
{ informația utilă e}
q ← **creeazaNodLSI**(*i*)
{ se creează legătura între nodul q și capul listei înlănțuite }
{ formate din elementele i+1,..., n }
[*q*].urm ← **creare**(*i*+1, *n*)
creare ← *q*
SfDacă
SfFuncție

Funcția creareLista(*n*)
*{**post**: se returnează un* ↑NodLSI*, pointer spre capul listei înlănțuite formate }*
{ din elementele 1, 2,..., n }
creareLista ← **creare**(1, *n*)
SfFuncție

2. **Dându-se un număr natural *n*, să se calculeze suma 1+2+3+…+*n*.**

a) direct recursiv

$$suma(n) = \begin{cases} 0 & daca\ n = 0 \\ n + suma(n-1) & altfel \end{cases}$$

b) folosind o funcție auxiliară recursivă pentru calculul sumei $i+(i+1)+\ldots+n$

$$suma\_aux(n,i) = \begin{cases} 0 & daca\ i > n \\ i + suma(n, i+1) & altfel \end{cases}$$

$$suma(n) = suma\_aux(n, 0)$$

3. **Să se construiască lista obținută prin adăugarea unui element la sfârşitul unei liste.**

// construirea listei $(l_1, l_2, \ldots, l_n, e)$

$$adaug(e, l_1 l_2 \ldots l_n) = \begin{cases} (e) & daca\quad l\ e\ vida \\ l_1 \oplus adaug(e, l_2 \ldots l_n) & altfel \end{cases}$$

4. **Să se verifice apariția unui element în listă.**

$$apare(E, l_1 l_2 \ldots l_n) = \begin{cases} fals & daca\ \ l\ e\ vida \\ adevarat & daca\quad l_1 = E \\ apare(E, l_2 \ldots l_n) & altfel \end{cases}$$

5. **Să se numere de câte ori apare un element în listă.**

$$nrap(E, l_1 l_2 \ldots l_n) = \begin{cases} 0 & daca\ \ l\ e\ vida \\ 1 + nrap(E, l_2 \ldots l_n) & daca\quad l_1 = E \\ nrap(E, l_2 \ldots l_n) & altfel \end{cases}$$

6. **Să se verifice dacă o listă numerică este mulțime.**

$$eMultime(l_1 l_2 \ldots l_n) = \begin{cases} adevarat & daca\ \ l\ e\ vida \\ fals & daca\ \ l_1 \in (l_2 \ldots l_n) \\ eMultime(l_2 \ldots l_n) & altfel \end{cases}$$

7. Să se construiască lista obținută prin transformarea unei liste numerice în mulțime.

$$multime(l_1 l_2 \ldots l_n) = \begin{cases} \phi & daca\ \ l\ e\ vida \\ multime(l_2 \ldots l_n) & daca\ \ l_1 \in (l_2 \ldots l_n) \\ l_1 \oplus multime(l_2 \ldots l_n) & altfel \end{cases}$$

8. Să se returneze inversa unei liste.

a) direct recursiv

$$invers(l_1 l_2 \ldots l_n) = \begin{cases} \phi & daca\ \ l\ e\ vida \\ invers(l_2 \ldots l_n) \oplus l_1 & altfel \end{cases}$$

b) folosind o variabilă colectoare

L	**Col**
(1, 2, 3)	∅
(2, 3)	(1)
(3])	(2, 1)
∅	(3, 2, 1)

$$invers\_aux(l_1 l_2 \ldots l_n, Col) = \begin{cases} Col & daca\ l\ e\ vida \\ invers\_aux(l_2 \ldots l_n, l_1 \oplus Col) & altfel \end{cases}$$

$$invers(l_1 l_2 \ldots l_n) = invers\_aux(l_1 l_2 \ldots l_n, \emptyset)$$

9. Să se construiască lista obținută prin ştergerea aparițiilor unui element dintr-o listă.

$$stergere(E, l_1 l_2 \ldots l_n) = \begin{cases} \phi & daca\ l\ e\ vida \\ l_1 \oplus stergere(E, l_2 \ldots l_n) & daca \quad l_1 \neq E \\ stergere(E, l_2 \ldots l_n) & altfel \end{cases}$$

10. Să se determine al *k*-lea element al unei liste (*k* ≥1).

$$element(l_1 l_2 \ldots l_n, k) = \begin{cases} \phi & daca\ \ l\ e\ vida \\ l_1 & daca\ k = 1 \\ element(l_2, \ldots l_n, k-1) & altfel \end{cases}$$

11. Să se determine diferența a două mulțimi reprezentate sub formă de listă.

$$diferenta(l_1 l_2 \ldots l_n, p_1 p_2 \ldots p_m) = \begin{cases} \phi & daca\ \ l\ e\ vida \\ diferenta(l_2 \ldots l_n, p_1 p_2 \ldots p_m) \quad daca & l_1 \in (p_1 p_2 \ldots p_m) \\ l_1 \oplus diferenta(l_2 \ldots l_n, p_1 p_2 \ldots p_m) & altfel \end{cases}$$

<u>Temă</u>

1. Să se verifice dacă un număr natural este sau nu prim.
2. Să se calculeze suma primelor *k* elemente dintr-o listă numerică ($l_1 l_2 \ldots l_n$).
3. Să se șteargă primele *k* numere pare dintr-o listă numerică.