

CURS 1

Introducere în programarea declarativă. Recursivitate

- Descrierea cursului, cerințele și modalitatea de notare [\[aici\]](#)

Cuprins

Bibliografie	1
1. Programare și 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 și Prolog. Aplicații în Inteligența Artificială.*, Ed. Albastră, Cluj-Napoca, 2012

1. Programare și limbaje de programare

➤ Limbaje

- **Procedurale (imperative) – limbaje de nivel înalt**
 - Fortran, Cobol, Algol, Pascal, C,...
 - program – secvență de instrucțiuni
 - instrucțiunea de atribuire, structuri de control – pentru controlul execuției secvențiale, ramificării și ciclării.
 - rolul programatorului – “**ce**” și “**cum**”
 1. să descrie **CE** e de calculat
 2. să organizeze calculul
 3. să organizeze gestionarea memoriei

— / —
CUM
 - !!! se susține că instrucțiunea de atribuire este periculoasă în limbajele de nivel înalt, așa cum instrucțiunea 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, ușor de înțeles (au o bază simplă), extensibile

- programele pot fi văzute ca descrieri care declară informații despre valori, mai degrabă decât instrucțiuni 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 și raționarea (deducții ale consecințelor și proprietăților) devin activități 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 funcțiilor și definiții de funcții), cu evaluare automată a expresiilor
 2. **limbaje logice** (de exemplu Prolog, Datalog, Parlog), care se focalizează pe aserțiuni logice care descriu relațiile dintre valorile datelor și derivări automate de răspunsuri la întrebări, plecând de la aceste aserțiuni.
 - 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 necesități practice (transcrierea directă a formulelor matematice recursive; vezi funcția lui Ackermann)
- recursivitatea este acel mecanism prin care un subprogram (funcție, 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 ieșire 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 al 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

Notatii

- 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

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

b) folosind o funcție auxiliară recursivă pentru crearea sublistei ($i, i+1, \dots, n$)

// crearea listei formată din elementele $i, i+1, \dots, n$

Model matematic recursiv

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

// crearea listei formată din elementele 1, 2, ..., n

$$createLista(n) = create(1, n)$$

Pseudocod

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

NodLSI

e: TElement //informația utilă a nodului

urm: \uparrow NodLSI //adresa la care e memorat următorul nod

LSI

prim: \uparrow NodLSI //adresa primului nod din listă

Funcția creeazaNodLSI(e)

{pre: e: TElement}

*{post: se returnează un \uparrow NodLSI conținând e ca informație utilă}
 {se alocă un spațiu de memorare pentru un NodLSI }*

{p: \uparrow NodLSI}

aloca(p)

[p].e \leftarrow e

[p].urm \leftarrow NIL

{rezultatul returnat de funcție}

creeazaNodLSI \leftarrow p

SfFuncție

Funcția creare(i, n)

*{post: se returnează un \uparrow NodLSI, pointer spre capul listei înlănțuite formate }
 { din elementele i, i+1,..., n }*

Dacă i > n atunci

creare \leftarrow NIL

altfel

*{ se alocă un spațiu de memorare pentru un NodLSI având }
 { informația utilă e }*

q \leftarrow creeazaNodLSI(i)

*{ se creează legătura între nodul q și capul listei înlănțuite }
 { formate din elementele i+1,..., n }*

[q].urm \leftarrow creare(i+1, n)

creare \leftarrow q

SfDacă

SfFuncție

Funcția creareLista(n)

*{post: se returnează un \uparrow NodLSI, pointer spre capul listei înlănțuite formate }
 { din elementele 1, 2,..., n }*

creareLista \leftarrow creare(1, n)

SfFuncție

2. Dându-se un număr natural n , să se calculeze suma $1+2+3+\dots+n$.

a) direct recursiv

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

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

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

$$suma(n) = suma_aux(n, 0)$$

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

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

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

4. Să se verifice apariția unui element în listă.

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

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

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

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

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

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

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

8. Să se returneze inversa unei liste.

a) direct recursiv

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

b) folosind o variabilă colectoare

L	Col
(1, 2, 3)	\emptyset
(2, 3)	(1)
(3)]	(2, 1)
\emptyset	(3, 2, 1)

$$invers_aux(l_1 l_2 \dots l_n, Col) = \begin{cases} Col & \text{daca } l \text{ e vida} \\ invers_aux(l_2 \dots l_n, l_1 \oplus Col) & \text{altfel} \end{cases}$$

$$invers(l_1 l_2 \dots l_n) = invers_aux(l_1 l_2 \dots l_n, \emptyset)$$

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

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

10. Să se determine al k -lea element al unei liste ($k \geq 1$).

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

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

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

Temă

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