PARADIGME DE PROGRAMARE

Curs 13

Rezumat.

Cuprins

- Rezolvare de probleme
- Dezvoltarea unui program bun
- Controlul complexității intelectuale
- Controlul complexității temporale / spațiale
- Controlul corectitudinii programului
- Concluzii

Rezolvarea problemelor în viața de zi cu zi

"Everyone in this country should learn to program a computer, because it teaches you to think." – Steve Jobs

- Abilitatea de a rezolva probleme se învață din experiență
- Strategia: să dezvolți o metodă de a aborda rezolvarea problemelor și să o practici
  - Rezolvând cât mai **multe microprobleme**
  - Nu doar probleme familiare, ci probleme care solicită noi și **noi moduri de a gândi**
- Contează **calitatea soluției** găsite (un lucru mai puțin neglijat în programare decât în viața de zi cu zi)
  - Mai ales în cazul problemelor recurente

Programele modelează lumea

- Sistemele software sunt modele ale lumii înconjurătoare și sunt dezvoltate pentru a rezolva probleme din viața de zi cu zi
- Primul pas în dezvoltarea unui sistem bun este o bună **modelare a problemei**
  - Fiecare paradigmă de programare propune un mod particular de reprezentare / prelucrare
  - Programatorul este responsabil să aleagă modul cel mai adecvat particularităților problemei
    - Obiecte multe, operații puține: POO
    - Obiecte puține, operații multe: PF
    - Multe prelucrări numerice: programare procedurală
    - Îndeplinirea unui anumit scop: PL (programare logică)
  - Lumea este în continuă schimbare, iar programele trebuie să permită **extinderea ușoară** a universului problemei
    - Noi operații: ușor în PF, greu în POO
    - Noi obiecte: ușor în POO, greu în PF

Cuprins

- Rezolvare de probleme
- Dezvoltarea unui program bun
- Controlul complexității intelectuale
- Controlul complexității temporale / spațiale
- Controlul corectitudinii programului
- Concluzii

Caracteristicile unui program bun

- Corectitudine și robustețe
- Ușurință în utilizare și dezvoltare
  - Interactivitate
  - Lizibilitate
  - Extensibilitate
- Eficiență
- Documentare

Aspecte discutate în acest curs

- Abordarea sarcinii de a dezvolta un program pentru o anumită problemă

+

- Controlul complexității intelectuale a programelor
- Controlul complexității temporale / spațiale a programelor
- O disciplină care să asigure corectitudinea programelor
- Un stil de programare care facilitează buna documentare

Abordarea scrierii unui program

1) **Înțelege** problema (fii capabil să o explici în cuvinte)
   - *"If you can't explain something in simple terms, you don't understand it."*—Richard Feynman
2) **Planifică**
   - Aspecte legate de modelare și de o strategie de ansamblu (formulare top-level)
3) **Divide** (problema în subprobleme)
   - **Wishful thinking**: formularea top-level a rezolvării identifică la nivel conceptual niște subprobleme mari, pe care ți le imaginezi gata rezolvate și pe baza cărora scrii programul
     - **Recursivitatea** ca formă de wishful thinking: presupui că există o procedură care rezolvă problema pentru versiuni mai mici decât cea curentă
   - Fiecare subproblemă este **rafinată succesiv** în același mod; păstrând fiecare pas de rafinare (detaliere) mic, ții sub control complexitatea intelectuală a întregului sistem
     - Rafinarea prezentă în orice paradigmă: la nivel de structuri de date, de funcții (PF), de relații (PL)

Cuprins

- Rezolvare de probleme
- Dezvoltarea unui program bun
- Controlul complexității intelectuale
- Controlul complexității temporale / spațiale
- Controlul corectitudinii programului
- Concluzii

Controlul complexității intelectuale

- Ciclul primitive – mijloace de combinare – abstractizare
  - Unealtă de bază pentru design-ul, întreținerea și extinderea sistemelor software
- Polimorfism
- Pattern matching
- Stil declarativ versus stil procedural

Elementele unui limbaj de programare

- **Primitive** – numere, stringuri, valori booleene, operatori aritmetici / relaționali / logici
- **Mijloace de combinare** (obțin structuri mai complexe pe baza celor simple)
  - Date compuse (liste, perechi, structuri (PL)), aplicația de funcție, predicate (PL)
- **Mijloace de abstractizare** (ascund detaliile și tratează lucrurile complexe ca pe primitive)
  - Posibilitatea de a da nume funcțiilor și expresiilor (și a le folosi apoi ca pe niște primitive)
  - Funcții/predicate ca niște cutii negre: izolarea utilizării unei proceduri de implementarea ei
    - **Modularitate**: Izolează componentele unui sistem și furnizează o interfață (input/output) pentru conectarea părților
  - **Tipuri de date abstracte** – ascund felul în care sunt grupate componentele și tratează fiecare unitate ca pe o colecție abstractă
  - **Proceduri de nivel înalt:** funcționale (PF), metapredicate (PL) – surprind șabloane frecvente de calcul (care pot avea ca parametri alte șabloane)

Bariera de abstractizare

- TDA = colecție de componente controlabilă printr-o interfață
  - **Constructorii** pun componentele împreună
  - **Selectorii** desfac întregul pentru a accesa componentele
  - În programele cu efecte laterale: funcții speciale (**mutator**) pentru alterarea componentelor
- Această interfață permite existența barierei de abstractizare – care ascunde de utilizator detaliile de implementare a tipului
  - **Barieră slabă**: utilizatorul cunoaște implementarea internă a tipului și e responsabilitatea lui să scrie cod care nu calcă bariera (ex: știe că listele sunt implementare ca perechi, sau știe că numerele complexe sunt implementate ca perechi)
  - **Barieră puternică**: utilizatorul nu cunoaște implementarea internă a tipului și nu poate să manipuleze valorile decât folosind interfața (ex: nu știe cum sunt implementate perechile)
  - Disciplina impusă este mai sigură decât cea auto-impusă ☺

Șabloane frecvente de calcul

- Transformarea fiecărui element dintr-o listă: map, maplist (Prolog)
- Găsirea tuturor elementelor care îndeplinesc o condiție: filter, findall / bagof / setof
- Verificarea că toate elementele îndeplinesc o condiție: forall
- Acumularea (eventual combinată cu prelucrarea) tuturor elementelor: fold

Efecte

- Programe mai scurte și mai ușor de citit
- Crește **puterea expresivă** a limbajului (aceste șabloane devin blocuri pe care le putem trata ca pe primitive)
- Se creează disciplina de a abstractiza și denumi sugestiv și alte șabloane frecvente

Planificarea programului

Design-ul programului presupune:

- **Design-ul structurilor / tipurilor de date**
  - ce informație este natural să fie grupată și ce interfață (care ascunde detaliile) trebuie creată pentru ea
- **Design-ul modulelor de calcul**
  - Se abstractizează: calculele des utilizate, concepte (pași în calcul care îndeplinesc un rol care poate fi ușor numit și înțeles ca un bloc independent)
  - Fiecare modul se poate implementa și testa separat
- **Design-ul interfețelor între module**
  - Se identifică input-ul și output-ul fiecărui modul
  - Se modelează fluxul informației prin sistem

Polimorfism

Polimorfism parametric = funcția se comportă la fel pentru argumente de tipuri diferite

Polimorfism ad-hoc = funcția se comportă diferit în funcție de tipul argumentelor

- **Claritate** sporită a codului
  - Putem folosi un același nume sugestiv pentru a descrie operații pe diferite tipuri
  - **Contează conceptul** de egalitate / lungime / apartenență etc. mai mult decât implementarea lui specifică
- **Reutilizare:** folosind aceeași funcție pentru diverse tipuri avem un cod mai scurt, mai ușor de întreținut

Pattern matching

- Constructorii unui TDA: închid într-o abstracțiune mai multe informații
- Selectorii: desfac constructorii pentru a extrage una din aceste informații
  - Alternativ, se poate folosi pattern-matching pe constructori – metodă naturală, similară cu experiența umană de recunoaștere a semnificației unui anumit obiect
  - **Facilitat de tiparea statică** – se pot identifica argumentele constructorilor și lega variabilele la valori – v. Haskell, care nu face nicio altă verificare suplimentară
  - **Facilitat de algoritmul de unificare** – capabil să verifice identitatea oricăror 2 structuri, fără ca acestea să fie instanțe ale unui TDA bine definit în program sau în limbaj – v. Prolog, în care pattern matching-ul este mult mai puternic, permite atât verificări cât și crearea unor noi structuri care se potrivesc cu forma dorită

Stil declarativ versus stil procedural

Stil declarativ

- Natural pentru a reprezenta și a raționa despre ceea ce este adevărat
- Nu oferă o metodă computațională de a deduce noi informații
- Mai ușor de înțeles, dar necesită mecanisme specifice care să sprijine acest stil de programare – ex: backtraking încorporat în Prolog, evaluare leneșă pentru a folosi fluxuri, etc.
  - Adesea este necesar un compromis între eleganță și eficiență

Stil procedural

- Natural pentru a reprezenta metode de calcul (secvențe de pași care transformă o mulțime de fapte într-o nouă mulțime de fapte de interes)
- Nu se aseamănă cu metoda umană de a reprezenta și deduce noi informații

Cuprins

- Rezolvare de probleme
- Dezvoltarea unui program bun
- Controlul complexității intelectuale
- Controlul complexității temporale / spațiale
- Controlul corectitudinii programului
- Concluzii

Controlul complexității temporale / spațiale

- Complexitatea operațiilor primitive
- Funcții și procesele generate de acestea
- Tipuri de evaluare: aplicativă, normală, lenesă
- Fluxuri

Complexitatea operațiilor primitive

- Ciclul primitive – mijloace de combinare – abstractizare
  - Face să putem trata operații complexe ca pe niște primitive
  - Atrage atenția asupra faptului că operațiile primitive (în limbaj) au o implementare în spate – care adesea nu e în complexitate constantă
    - Ex: length – complexitate O(n) în Racket
      – complexitate O(1) în Python
- Alte exemple (aspecte de care trebuie ținut cont la alegerea limbajului)
  - Algoritmii de sortare impliciți în diverse limbaje diferă (merge-sort, quick-sort)
  - Felul în care sunt implementate structurile de date standard (stive, cozi, vectori, hash-uri, heap-uri)

Funcții și procese

- Un anumit mod de a scrie funcțiile (de a descompune problema)
  duce la
  un anumit proces (secvență de pași efectuată în calculator pentru transformarea datelor)
- Tipurile de procese se caracterizează prin **complexitatea lor temporală / spațială**
  - Recursivitate **pe stivă** (consumă spațiu suplimentar din cauza alocărilor pe stivă)
  - Recursivitate **pe coadă** (funcția este recursivă, dar procesul generat este iterativ)
  - Recursivitate **arborescentă** (de tip divide and be conquered, duce la timpi exponențiali)
  - Recursivitate de tip **divide et impera** (timp logaritmic)
- **Exemplu:** pentru exponențiere (a^n), când n este par, putem descompune problema:
  - $a^n = (a^2)^{n/2}$
  - $a^n = (a^{n/2})^2$

Cum e mai bine?

Tipuri de evaluare

Evaluare aplicativă = evaluează argumentele înainte de apel

- Eficientă (expresia se poate înlocui prin valoarea ei), dar nu garantează terminarea calculului

Evaluare normală = evaluează subexpresiile în expresia finală, de la stânga la dreapta, când e nevoie de ele

Evaluare lenesă = ca evaluarea normală, dar salvează rezultatul evaluărilor pentru a nu reevalua o aceeași subexpresie de mai multe ori

- Evaluare normală + memoizare (necesită mecanisme speciale)
- Câștigul se vede în posibilitatea de a lucra cu structuri lenese (indefinit de lungi)

Fluxuri

- Liste cu evaluare leneșă
- Decuplarea ordinii aparente a acțiunilor de ordinea reală:
  - Programe elegante, declarative (asemenea programelor pe liste)
  - Procese eficiente (asemenea proceselor iterative)
- Elimină compromisul eleganță – eficiență

Cuprins

- Rezolvare de probleme
- Dezvoltarea unui program bun
- Controlul complexității intelectuale
- Controlul complexității temporale / spațiale
- Controlul corectitudinii programului
- Concluzii

Controlul corectitudinii programului

- Efecte laterale și transparență referențială
- Tipuri de legare a variabilelor
- Tipare
- Demonstrații de corectitudine

Efecte laterale și transparență referențială

Efecte laterale = alte efecte ale unei funcții decât cel de a întoarce o valoare

- Ex: atribuiri, operații de intrare/ieșire
- Compromis între ușurința de a obține anumite rezultate și riscul de efecte neprevăzute (bug-uri)
- Știind să programăm fără efecte laterale, limităm uzul acestora la situațiile în care este necesar

Transparență referențială = funcții pure (aplicate pe aceleași argumente întorc mereu aceeași valoare), expresii perfect substituibile cu valoarea lor

- Valori imutabile, procedurile sunt funcții matematice
- Programe atemporale, ușor paralelizabile (fire paralele nu vor modifica o resursă comună), permite evaluare leneșă (expresiile odată evaluate nu trebuie reevaluate în caz că vor produce altă valoare)
- Nu contează ordinea de evaluare, nu contează strategia de evaluare, este ușor de prezis ce va întoarce o funcție (folosind modelul substituției) → risc scăzut de bug-uri

Efectele programării cu atribuiri

Modelul substituției = se înlocuiesc parametrii formali ai funcției cu argumentele și se evaluează corpul

Modelul contextual = o funcție este un text și un context (care poate fi suprascris în timp)

Introducerea atribuirilor

- Modelul substituției trebuie înlocuit cu modelul contextual (mai complicat) pentru a include noțiunea de timp
- Expresii cu sintaxă identică pot avea semantică diferită (depinde de contextul în care sunt evaluate)
- O variabilă nu mai e un nume pentru o valoare, ci o locație în memorie → probleme de identitate (indică două variabile același obiect sau doar au aceeași valoare?)
- O procedură nu mai e o funcție, ci un obiect cu un context care ne spune cum să interpretăm simbolurile din calculul descris de procedură
- Necesitatea sincronizării firelor paralele, pentru cazurile în care mai multe fire vor să altereze aceeași zonă de memorie

Tipuri de legare a variabilelor

Legare statică = domeniu de vizibilitate controlat textual (determinat la compilare)

- Varianta implicită în majoritatea limbajelor de programare

Legare dinamică = domeniu de vizibilitate controlat dinamic (în funcție de timp) (determinat la execuție)

- Permite suprascrierea contextului moștenit de o funcție la definirea ei
- Risc de bug-uri ca de fiecare dată când introducem noțiunea de timp în programele noastre (☺)

Tipare

Tipare statică = variabilele și valorile au un tip asociat, tipuri verificate la compilare

- Ajută la design-ul programelor
  - Exprimăm funcțiile în funcție de tipul input-urilor și tipul output-ului
  - Ne asigurăm că se pasează valori valide de la o funcție la alta și că avem toate modulele necesare pentru a circula informația prin sistem
  - Inconsistențele vor fi semnalate la compilare

Tipare dinamică = doar valorile au un tip asociat, tipuri verificate la execuție

- Erorile nu vor fi detectate decât dacă le generează o secvență de execuție
- Flexibilitate în scrierea funcțiilor – programatorul are libertatea să își impună sau nu disciplina consistenței tipurilor
- Preferată în programele mici, problematică în sisteme complexe întreținute de un număr mare de programatori

Design ajutat de tipare - Exemplu

```
1. -- o funcție care primește o altă funcție fun și un număr n și
2. -- întoarce funcția care reprezintă aplicarea repetată de n ori a lui fun
3. {-
4. repeated :: (a -> a) -> Int -> (a -> a)
5. repeated _ 0 = \x -> x
6. repeated fun n = \x -> fun (repeated fun (n-1) x)
7. -}
8. repeated :: (a -> a) -> Int -> (a -> a)
9. repeated _ 0 x = x
10.repeated fun n x = fun (repeated fun (n-1) x)
```

Observație: puteam folosi cunoșințe de tipuri și pentru a ghida design-ul funcției în Racket

Tipuri definite de utilizator

- Când realizăm abstractizare la nivel de date în Racket sau Prolog – de fapt definim un tip fără disciplina unui sistem de tipuri
  - Apoi depinde de noi să prelucrăm valorile tipului doar cu operațiile destinate lor
  - Alternativă (disciplină auto-impusă): etichetăm manual datele de un anumit tip și verificăm eticheta fiecărei valori înainte de a opera asupra ei

Exemple

```
(define (make-tree root left right) (list 'tree root left right))
(define (tree? T) (equal? 'tree (first T)))
(define root second)
(define left third)
(define right fourth)
```

```
root(tree(Root, _, _), Root).
left(tree(_, Left, _), Left).
right(tree(_, _, Right), Right).
```

Tipuri definite de utilizator

- Haskell impune această disciplină
  - TDA-urile definite de utilizator sunt gata „etichetate” cu constructorii specifici fiecărui tip (nu pot avea același nume de constructor pentru tipuri diferite)

    → doar date prelucrate cu operații specifice lor (**data directed programming**)
  - Aplicarea funcțiilor specifice unui tip necesită pattern matching pe constructori și vom genera o eroare (la compilare) dacă încercăm aplicarea pe un tip necorespunzător

    → doar operații aplicate pe tipul așteptat (**defensive programming**)

Exemplu

```
data Tree a = Nil | Node a (Tree a) (Tree a)
root  (Node r _ _) = r
left  (Node _ l _) = l
right (Node _ _ r) = r
```

Alte argumente pro tipare

- Îmbunătățește **documentarea** programelor
  - Informații despre natura input-urilor și output-urilor fiecărei funcții
  - Constrângeri asupra argumentelor (ex: Eq, Ord, Num)
  - Documentarea trebuie completată cu informații despre scopul funcției și rolul fiecărui parametru
- Îmbunătățește **productivitatea**
  - Mulțumită IDE-urilor care oferă informații despre tipurile așteptate de fiecare funcție
  - Mai ales în sistemele cu inferență de tip (programatorul este asistat, nu forțat să adnoteze singur)
- Crește **eficiența**
  - Compilatorul optimizează reprezentarea / prelucrarea valorilor în funcție de particularitățile tipului

Demonstrații de corectitudine

- Cu atât mai ușoare (și de încredere) cu cât codul este mai apropiat de specificația formală
- Programarea funcțională se bazează pe funcții recursive, a căror corectitudine se poate demonstra prin **inducție** (inducție matematică, inducție structurală)
  - Demonstrează corectitudinea cazului de bază
  - Demonstrează cum dacă o versiune mai simplă a funcției este corectă, atunci prelucrarea ei cu operații adiționale duce la un răspuns corect pentru problema extinsă
  - Metoda inducției coincide cu metoda design-ului funcției recursive!
- Programarea logică se bazează pe aplicarea rezoluției (regulă de inferență consistentă și completă) asupra adevărurilor din program
  - Programele sunt corecte atât timp cât specificația formală este corectă

Cuprins

- Rezolvare de probleme
- Dezvoltarea unui program bun
- Controlul complexității intelectuale
- Controlul complexității temporale / spațiale
- Controlul corectitudinii programului
- Concluzii

Ce ați dobândit

- Experiență în rezolvarea de micro-probleme variate
- Experiență în felul în care gândiți calculul, independent de detaliile unui limbaj de programare, în scopul controlului complexității intelectuale a problemei
- Cultură generală privind alegerile care stau la baza design-ului limbajelor de programare
- O imagine a efectelor alegerilor voastre de design (efecte laterale, procese recursive sau iterative, abstractizare, disciplina tipurilor, etc.)
- Relația dintre diversele paradigme de programare și formalismele matematice pe care ele se bazează

Alte idei

- Un limbaj facilitează / impune o paradigmă, dar puteți folosi disciplina unei paradigme și în alte limbaje

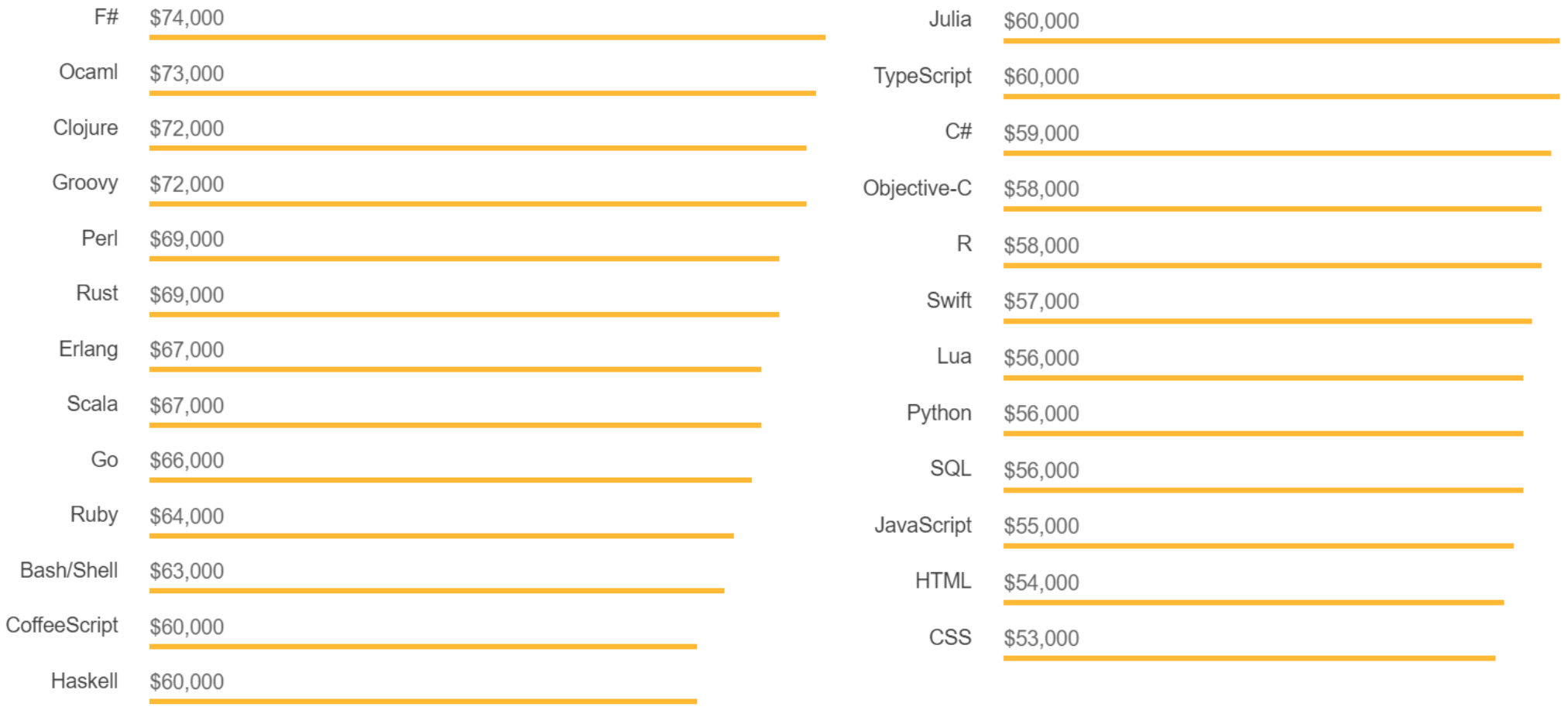
*„When you choose technology, you have to **ignore what other people are doing**, and consider only what will work the best. You can't trust the opinions of the others, because they're satisfied with whatever language they happen to use, because it dictates the way they think about programs.*

*Lisp is so great not because of some magic quality visible only to devotees, but because it is simply the most powerful language available. And the reason everyone doesn't use it is that **programming languages are not merely technologies, but habits of mind as well, and nothing changes slower**.*

If you think of using Lisp in a startup, you shouldn't worry that it isn't widely understood. You should hope that it stays that way. Back when I was writing books about Lisp, I used to wish everyone understood it. But when we started Viaweb I found that changed: I wanted everyone to understand Lisp except our competitors. ” – Paul Graham

Popular versus bine plătit

(https://insights.stackoverflow.com/survey/2018/#top-paying-technologies)



Test de recuperare

În Prolog, pentru un graf orientat dat prin fapte de tip nod/1 și arc/2, scrieți un predicat care reușește dacă fiecare nod din graf are cel puțin 2 vecini.

La revedere și...

Time for closure.

```
(λ ()
   'Sesiune_plăcută!)
```