Paradigme de Programare

S.I. dr. ing. Andrei Olaru
slides: Mihnea Muraru si Andrei Olaru

Catedra de Calculatoare

2013 – 2013, semestrul 2

Cursul 1

Introducere

Cuprins

Organizare

Resurse de bază

unde găsesc informații?

`http://elf.cs.pub.ro/pp/`

Regulament: `http://elf.cs.pub.ro/pp/regulament`

Teme și forumuri: `http://cs.curs.pub.ro` (în curând)

Notare

- Laborator: 1p (cu bonusuri, max 1p total)
- Teme: 4p (4 $\times$ 1p) (cu bonusuri, max 6p pe parcurs)
- Teste la curs: 0,5p
- Test din materia de laborator: 0,5p
- Examen: 4p

Laborator

- Accent pe lucrul efectiv
- Parcurgerea documentaţiei înaintea laboratorului
- Test la începutul laboratorului

Obiective

Conținutul cursului

Ce vom studia?

1. Diverse perspective conceptuale asupra noţiunii de calculabilitate efectivă $\rightarrow$ modele de calculabilitate

2. Influenţa perspectivei alese asupra procesului de modelare şi rezolvare a problemelor $\rightarrow$ paradigme de programare

3. Limbaje de programare aferente paradigmelor, cu accent pe aspectul comparativ

De ce?

Just a thought

I suppose it is tempting, if the only tool you have is a hammer, to treat everything as if it were a nail.

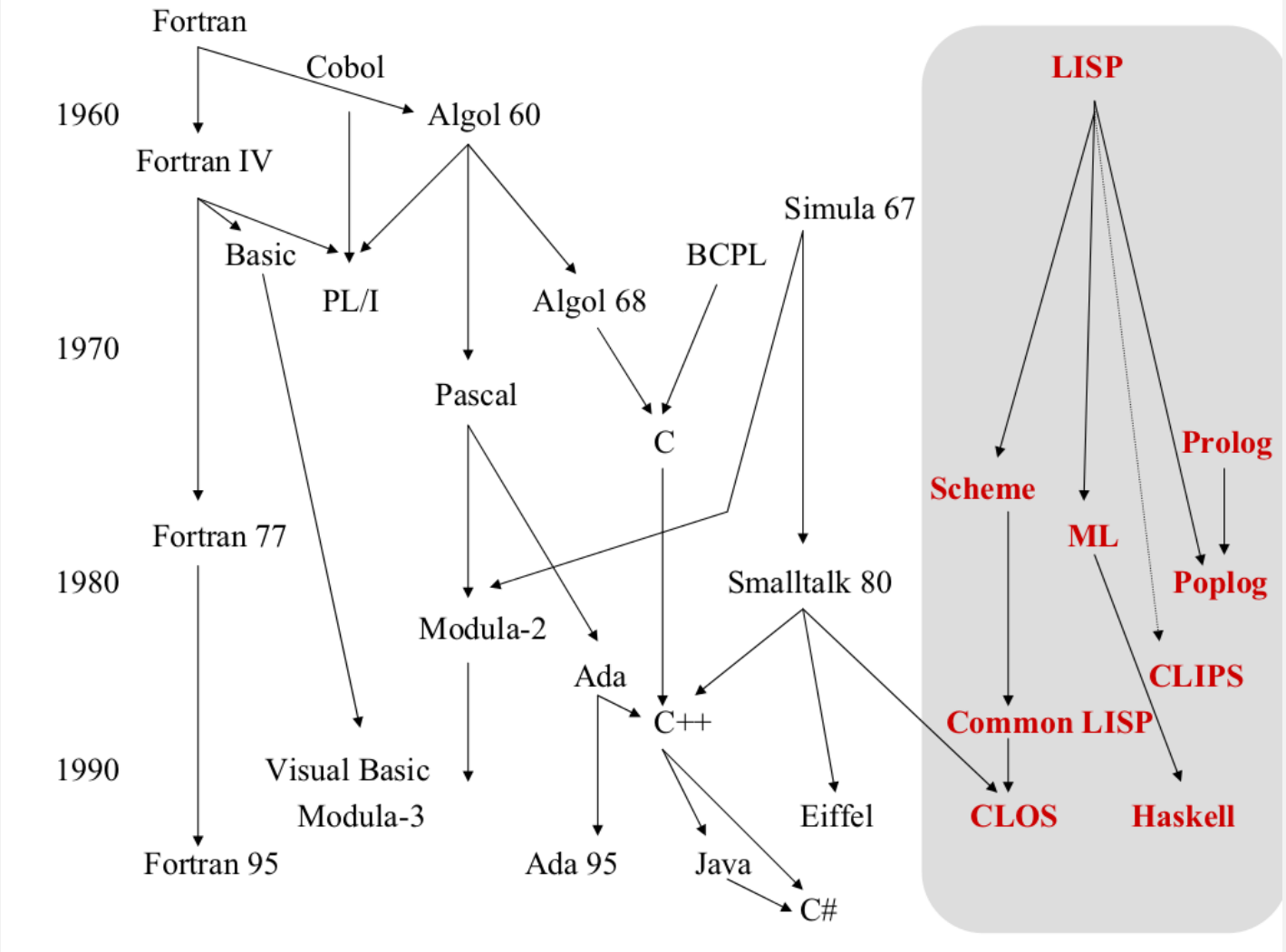
The law of instrument – Abraham Maslow

De ce?
Mai concret

- Lărgirea spectrului de abordare a problemelor
- Identificarea perspectivei naturale de modelare a unei probleme şi alegerea limbajului adecvat
- Sporirea capacităţii de învăţare a noi limbaje şi de adaptare la particularităţile şi diferenţele dintre acestea
- Exploatarea mecanismelor oferite de limbajele de programare

Limbaje de programare ’50-’00

Un pic de istorie



Limitele calculabilităţii

Ce putem calcula și cum

- Teza Church-Turing:
  efectiv calculabil = Turing calculabil
- Echivalenţa celorlalte modele de calculabilitate – şi a multor altora – cu Maşina Turing
- Există vreun model mai expresiv?

Modele de calculabilitate

Modele → paradigme → limbaje

- Mașina Turing → Paradigma imperativă
  - → Procedurală → C
  - → Orientată-obiect → Java, C++
- Calcul Lambda → Paradigma funcțională → Scheme, Haskell
- Mașina Markov → Paradigma asociativă → CLIPS
- Mașina FOL → Paradigma logică → Prolog

Exemplu

O primă problemă

Exemplul 3.1.

Să se determine elementul minim dintr-un vector.

Modelare imperativă

Varianta procedurală

minList(L, n)

1: $min \leftarrow L[1]$
2: $i \leftarrow 2$
3: **while** $i \leq n$ **do**
4: **if** $L[i] < min$ **then**
5: $min \leftarrow L[i]$
6: **end if**
7: $i \leftarrow i + 1$
8: **end while**
9: **return** min

Modelare funcţională

- Ideea: $\text{minList}(L) = \text{if}(\text{eq}(\text{length}(L), 1), \text{head}(L), \text{min}(\text{head}(L), \text{minList}(\text{tail}(L))))$
- Scheme:

```
1 (define minList
2   (lambda (l)
3     (if (= (length l) 1) (car l)
4         (min (car l) (minList (cdr l))))))
```

- Haskell:

```
1 minList [h] = h
2 minList (h:t) = min h (minList t)
```

Modelare asociativă

- Ideea: $\text{minList}(L) = m \in L \mid \nexists x \in L \bullet x < m$
- CLIPS:

```
1  (deffacts facts
2       (elem 3)
3       (elem 2)
4       (elem 0)
5       (elem 1))
6
7  (defrule minList
8       (elem ?m)
9       (not (elem ?x & :(< ?x ?m)))
10    =>
11      (assert (min ?m)))
```

Modelare logică

- Axiome:
  1. $x \leq y \implies \min(x, y, x)$
  2. $y < x \implies \min(x, y, y)$
  3. $\text{minList}([m], m)$
  4. $\text{minList}([y|t], n) \wedge \min(x, n, m) \implies \text{minList}([x, y|t], m)$
- Prolog:

```
1  min(X, Y, X) :- X =< Y.
2  min(X, Y, Y) :- Y < X.
3
4  minList([M],M).
5  minList([X,Y|T],M) :- minList([Y|T],N), min(X,N,M).
```

Paradigme

Ce este o paradigmă de programare?

- Un set de convenţii ce dirijează maniera în care gândim programele
- Ea dictează modul în care:
  - reprezentăm datele
  - operaţiile prelucrează datele respective
- Atenție! Paradigma nu are legătură cu sintaxa limbajului!
- Există diferențe importante între paradigmele de programare. Vom discuta despre efecte laterale, transparență referențială și gestionarea funcțiilor în limbaj.

Efecte laterale (*side effects*)

Definiţie

Exemplul 4.1.

În expresia `2 + (i = 3)`, subexpresia `(i = 3)`:

- produce valoarea `3`, conducând la rezultatul `5` al întregii expresii
- are efectul lateral de iniţializare a lui `i` cu `3`

Definiția 4.2 (Efect lateral).

Pe lângă valoarea pe care o produce, o expresie sau o funcţie poate modifica starea globală.

- Inerente în situaţiile în care programul interacţionează cu exteriorul → I/O!

Efecte laterale (*side effects*)

Consecinţe

Exemplul 4.3.

În expresia `x-- + ++x`, cu `x = 0`:

- evaluarea stânga $\rightarrow$ dreapta produce `0 + 0 = 0`
- evaluarea dreapta $\rightarrow$ stânga produce `1 + 1 = 2`
- dacă înlocuim cele două subexpresii cu valorile pe care le reprezintă, obţinem
  `x + (x + 1) = 0 + 1 = 1`

- Adunare necomutativă?!
- Importanţa ordinii de evaluare!
- Dependenţe implicite, puțin lizibile şi posibile generatoare de bug-uri.

Transparenţă referenţială

Definiţie

Exemplul 4.4.

1. "**Zeus** este fiul lui Cronos"
   - Zeus este Jupiter în mitologia romană
   - "**Jupiter** este fiul lui Cronos" → aceeaşi semnificaţie
2. "Ionel ştie că **Zeus** este fiul lui Cronos"
   - "Ionel ştie că **Jupiter** este fiul lui Cronos" → altă semnificaţie

Definiția 4.5 (Transparenţă referenţială).

Confundarea unui obiect cu referinţa la acesta → cazul 1.

Transparenţă referenţială

Expresii

- Expresie transparentă referenţial: posedă o unică valoare, cu care poate fi substituită, păstrând semnificaţia programului.

Exemplul 4.6.

- `x-- + ++x` $\rightarrow$ nu, valoarea depinde de ordinea de evaluare
- `x = x + 1` $\rightarrow$ nu, două evaluări consecutive vor produce rezultate diferite
- `x` $\rightarrow$ ar putea fi, în funcţie de statutul lui `x` (globală, statică etc.)

- Absentă în prezenţa efectelor laterale!

Transparenţă referenţială

Funcţii

- Funcţie transparentă referenţial: rezultatul întors depinde exclusiv de parametri

Exemplul 4.7.

```
int transparent(int x) {
    return x + 1;
}
```

```
int g = 0;

int opaque(int x) {
    return x + ++g;
}
```

- `opaque(3) - opaque(3) != 0`!
- Funcţii transparente: `log`, `sin` etc.
- Funcţii opace: `time`, `read` etc.

Transparenţă referenţială

Avantaje

- Lizibilitatea codului
- Demonstrarea formală a corectitudinii programului
- Optimizare prin reordonarea instrucţiunilor de către compilator şi prin caching
- Paralelizare masivă, prin eliminarea modificărilor concurente

Funcţii ca valori de prim rang

Definiţie

Definiția 4.8 (Valoare de prim rang).

O valoare ce poate fi:

- creată dinamic
- stocată într-o variabilă
- trimisă ca parametru unei funcţii
- întoarsă dintr-o funcţie

Exemplul 4.9.

Să se scrie funcţia `compose`, ce primeşte ca parametri alte 2 funcţii, `f` şi `g`, şi întoarce funcţia obţinută prin compunerea lor, `f ∘ g`.

Funcţii ca valori de prim rang: `Compose` C

```
1  int compose(int (*f)(int), int (*g)(int), int x) {
2      return (*f)((*g)(x));
3  }
```

- În C, funcţiile nu sunt valori de prim rang.

Funcţii ca valori de prim rang:

Java

```
1  abstract class Func<U, V> {
2      public abstract V apply(U u);
3
4      public <T> Func<T, V> compose(final Func<T, U> f) {
5          final Func<U, V> outer = this;
6
7          return new Func<T, V>() {
8              public V apply(T t) {
9                  return outer.apply(f.apply(t));
10             }
11         };
12     }
13 }
```

- În Java, funcţiile nu sunt valori de prim rang.

Funcţii ca valori de prim rang: `Compose`

Scheme & Haskell

- Scheme:

```
1  (define compose
2    (lambda (f g)
3      (lambda (x)
4        (f (g x)))))
```

- Haskell:

```
1  compose = (.)
```

- În Scheme şi Haskell, funcţiile sunt valori de prim rang.

Funcţii ca valori de prim rang

Aplicaţii parţiale

Exemplul 4.10.

```
(define sum-uncurry
  (lambda (x y)
    (+ x y)))


(sum-uncurry 1 2)


;
```

```
1  (define sum-curry
2    (lambda (x)
3      (lambda (y)
4        (+ x y))))
5
6  ((sum-curry 1) 2)
7
8  (define sum-with-1
9    (sum-curry 1))
10 (sum-with-1 2)
```

Funcţii ca valori de prim rang

Funcţii de ordin superior (funcţionale)

Definiția 4.11 (Funcţională).

Funcţie care ia funcţii ca parametru şi/sau întoarce o funcţie.

Exemplul 4.12.

```
1 (define l '(1 2 3))
2
3 ((compose car cdr) l) ; 2
4 (map list l)          ; ((1) (2) (3))
5 (filter odd? l)       ; (1 3)
6 (foldl + 0 l)         ; 6
```

Paradigma imperativă

Caracteristici

- Orientare spre acţiuni şi efectele acestora
- Cum se obţine soluţia
- Atribuirea ca operaţie fundamentală
- Efecte laterale permise, compromiţând transparenţa referenţială
- Secvenţierea instrucţiunilor
- Programe cu stare, văzută ca mulţimea valorilor variabilelor la un anumit moment, ce pot influenţa rezultatul evaluării aceleiaşi expresii

Paradigma funcţională

Caracteristici

- Funcţia văzută în sens matematic, exclusiv prin valoarea pe care o calculează
- Funcţii ca valori de prim rang
- Interzicerea efectelor laterale, pentru eliminarea dependenţelor implicite $\rightarrow$ modularitate sporită, la nivel de funcţie!
- Promovarea transparenţei referenţiale, alături de avantajele acesteia
- Diminuarea importanţei ordinii de evaluare
- Programe fără stare

Paradigma funcţională

Just a thought

It's really clear that the imperative style of programming has run its course. We're sort of done with that. However, in the declarative realm we can speculate a 10x improvement in productivity in certain domains.

Anders Hejlsberg
C# Architect

Paradigmele asociativă şi logică

Caracteristici

- Accent pe formularea proprietăţilor soluţiei
- Ce trebuie obţinut (vs. “cum” la imperativă)
- Fapte, reguli, înlănţuire înainte/înapoi
- Orientare spre date

Aplicaţii ale diverselor paradigme

- Manipulare simbolică în inteligenţa artificală
  - Sisteme expert
  - Demonstrarea de teoreme
- Calcul paralel
- Demonstrarea automată a corectitudinii programelor şi testare, datorită modelului mai simplu de execuţie
- Adoptare a paradigmei funcţionale în limbajele noi: C#, F#, Python, JavaScript, Clojure (JVM), Scala
- Erlang (Ericsson) — limbaj funcţional utilizat în telecomunicaţii, economie, comerţ electronic

Limbaje

Câteva trăsături

- Paradigmă
  - Model de abordare a problemei
  - Model de execuție / calculabilitate
- Tipare
  - Statică/dinamică
  - Tare/slabă
- Ordinea de evaluare a parametrilor funcţiilor
  - Aplicativă
  - Normală
- Legarea variabilelor
  - Statică
  - Dinamică

Sfârșitul primului curs

Ce am învățat

· Paradigme de programare, limbaje, modele de calculabilitate, The law of instrument, efect lateral, transparență referențială, valori de prim rang, Scheme, Haskell, Prolog, CLIPS.