Elementi di semantica denotazionale ed operazionale

Contenuti

- sintassi astratta e domini sintattici
  - un frammento di linguaggio imperativo
- semantica denotazionale
  - domini semantici: valori e stato
  - funzioni di valutazione semantica
- dalla semantica denotazionale a quella operazionale
  - modifiche nei domini e nelle funzioni di valutazione semantica
- semantica e paradigmi
- semantica e supporto a tempo di esecuzione
- verso definizioni semantiche “eseguibili”

Sintassi e semantica

- un linguaggio di programmazione, come ogni sistema formale, possiede
  - una sintassi
    - le formule ben formate del linguaggio (o programmi sintatticamente corretti)
  - una semantica
    - che assegna un significato alle formule ben formate
    - dando un'interpretazione ai simboli
      - in termini di entità (matematiche) note
- la teoria dei linguaggi formali fornisce i formalismi di specifica (e le tecniche di analisi) per trattare gli aspetti sintattici
- per la semantica esistono diverse teorie
  - le più importanti sono la *semantica denotazionale* e la *semantica operazionale*
  - ne esistono altre (per esempio, la semantica assiomatica)
- la semantica formale viene di solito definita su una rappresentazione dei programmi in *sintassi astratta*
  - invece che sulla reale rappresentazione sintattica concreta

Sintassi astratta

- in sintassi concreta, i costrutti del linguaggio sono rappresentati come stringhe
  - è necessaria l'analisi sintattica per mettere in evidenza la struttura importante dal punto di vista semantico, risolvendo problemi legati solo alla comodità di notazione
    - proprietà degli operatori concreti (associatività, distributività, precedenze)
    - ambiguità apparenti (per esempio, x := x + 1)
- in sintassi astratta, ogni costrutto è un'espressione (o albero) descritto in termini di
  - applicazione di un operatore astratto (dotato di un tipo e di una arietà n) ad n operandi, che sono a loro volta espressioni (del tipo richiesto)
  - la rappresentazione di un programma in sintassi astratta è "isomorfa" all'albero sintattico costruito dall'analisi sintattica

Domini sintattici

- la sintassi astratta è definita specificando i *domini sintattici*
  - nomi di dominio, con metavariabili relative
  - definizioni sintattiche
- assumiamo l'esistenza del dominio sintattico IDE (e poi anche semantico) degli *identificatori*, con metavariabili I, I_1, I_2, ...

Un esempio (frammento di linguaggio imperativo)

- domini
  - EXPR (espressioni), con metavariabili E, E_1, E_2, ...
  - COM (comandi), con metavariabili C, C_1, C_2, ...
  - DEC (dichiarazioni), con metavariabili D, D_1, D_2, ...
  - PROG (programma), con metavariabile P
- definizioni sintattiche
  - E ::= I | val(I) | lambda(I, E_1) | plus(E_1, E_2) | apply(E_1, E_2)
  - C ::= ifthenelse(E, C_1, C_2) | while(E, C_1) | assign(I, E) | cseq(C_1, C_2)
  - D ::= var(I, E) | dseq(D_1, D_2)
  - P ::= prog(D, C)
- non è un frammento completo, poiché mancano
  - costanti (per esempio, interi e booleani)
  - altre necessarie operazioni (per esempio, su interi e booleani)

Semantica denotazionale: domini

- la semantica denotazionale associa ad ogni costrutto sintattico la sua *denotazione*
  - una funzione che ha come dominio e codominio opportuni *domini semantici*
- i domini semantici vengono definiti da *equazioni di dominio*

  nome-dominio = espressione-di-dominio
- le espressioni di dominio sono composte utilizzando i *costruttori di dominio*
  - enumerazione di valori

    bool = { True, False } int = { 0, 1, 2,..... }
  - somma di domini

    val = [int + bool] (un elemento appartiene a int oppure a bool)
  - iterazione finita di un dominio

    listval = val^* (un elemento è una sequenza finita o lista di elementi di val)
  - funzioni tra domini

    env = IDE → val (un elemento è una funzione che mappa un elemento di IDE in uno di val)
  - prodotto cartesiano di domini

    stato = env * store (un elemento è una coppia formata da un elemento di env ed uno di store)
- per ogni costruttore di dominio, esiste un metodo per definire l'ordinamento parziale del dominio, a partire da quelle dei domini utilizzati
  - garantendo che tutti i domini siano effettivamente *reticoli completi*

Domini semantici: lo stato

- in un qualunque linguaggio ad alto livello, lo stato deve comprendere un dominio chiamato *ambiente* (environment)
  - per modellare l'associazione tra gli identificatori ed i valori che questi possono denotare
- env = IDE → dval, con metavariabili ρ, ρ_1, ρ_2, ...
  - [ρ / I ← d] indica l'ambiente $\rho' = \lambda x.$ if x = I then d else ρ x
- il nostro frammento è imperativo ed ha la nozione di entità modificabile, cioè le variabili (create con le dichiarazioni e modificate con l'assegnamento)
  - non è possibile modellare lo stato con un'unica funzione, perché è possibile che identificatori diversi denotino la stessa locazione (aliasing)
- è quindi necessario un secondo componente dello stato, chiamato *memoria* (store)
  - per modellare l'associazione fra locazioni e valori che possono essere memorizzati
- store = loc → mval, con metavariabili σ, σ_1, σ_2, ...
  - [σ / l ← m] indica la memoria $\sigma' = \lambda x.$ if x = l then m else σ x

Domini semantici: i valori

- env = IDE → dval, con metavariabili ρ, ρ_1, ρ_2, ...
- store = loc → mval, con metavariabili σ, σ_1, σ_2, ...
- abbiamo già utilizzato (anche se non definito) due domini semantici di valori
  - dval (valori denotabili), dominio dei valori che possono essere denotati da un identificatore nell'ambiente
  - mval (valori memorizzabili), dominio dei valori che possono essere contenuti in una locazione nella memoria
- è necessario introdurre un terzo dominio semantico di valori
  - eval (valori esprimibili), dominio dei valori che possono essere ottenuti come semantica di una espressione
- i tre domini sono in generale diversi anche se possono avere delle sovrapposizioni
  - in questi casi useremo delle funzioni di "trasformazione di tipo"
- per capire come definire i tre domini dobbiamo analizzare il linguaggio
- assumiamo predefiniti i domini int e bool
  - con le relative "operazioni primitive"

  ed il dominio loc con una "funzione"
  - newloc : → loc (che, ogni volta che viene applicata, restituisce una nuova locazione)

Il dominio semantico fun

E ::= I | val(I) | lambda(I, E_1) | plus(E_1, E_2) | apply(E_1, E_2)

- le espressioni contengono l'astrazione
  - lambda(I, E_1) rappresenta una funzione
    - il cui corpo è l'espressione E_1
    - con parametro formale I
- la semantica di tale costrutto è una funzione del dominio fun
- fun = store → dval → eval
  - il valore di tipo dval sarà il valore dell'argomento nella applicazione
    - come vedremo, il passaggio di parametri si effettua nell'ambiente

eval, dval, mval

E ::= I | val(I) | lambda(I, E_1) | plus(E_1, E_2) | apply(E_1, E_2)

D ::= var(I, E) | dseq(D_1, D_2)

- eval
  - deve contenere int, bool e fun
  - decidiamo che non contiene loc (decisione semantica, la sintassi lo permetterebbe)
- dval
  - deve contenere loc
  - decidiamo che contenga anche int, bool e fun
- mval
  - deve contenere int e bool
  - decidiamo che non contiene loc e fun (decisione semantica, la sintassi lo permetterebbe)

eval = [int + bool + fun]

dval = [loc + int + bool + fun]

mval = [int + bool]

Le funzioni di valutazione semantica

E ::= I | val(I) | lambda(I, E_1) | plus(E_1, E_2) | apply(E_1, E_2)

C ::= ifthenelse(E, C_1, C_2) | while(E, C_1) | assign(I, E) | cseq(C_1, C_2)

D ::= var(I, E) | dseq(D_1, D_2)

P ::= prog(D, C)

- per ogni dominio sintattico esiste una funzione di valutazione semantica

  $\mathcal{E}$: EXPR → env → store → eval

  $\mathcal{C}$: COM → env → store → store

  $\mathcal{D}$: DEC → env → store → (env * store)

  $\mathcal{P}$: PROG → env → store → store
- le funzioni di valutazione semantica assegnano un significato ai vari costrutti, con una definizione data sui casi della sintassi astratta

Semantica delle espressioni

E ::= I | val(I) | lambda(I, E_1) | plus(E_1, E_2) | apply(E_1, E_2)
env = IDE → dval
store = loc → mval
fun = store → dval → eval
eval = [int + bool + fun]
dval = [loc + int + bool + fun]
mval = [int + bool]
$\mathcal{F}$: EXPR → env → store → eval

$\mathcal{F}(I) = \lambda\rho.\lambda\sigma.\ dvaltoeval(\rho(I))$

$\mathcal{F}(val(I)) = \lambda\rho.\lambda\sigma.\ mvaltoeval(\sigma(\rho(I)))$

$\mathcal{F}(plus(E_1, E_2)) = \lambda\rho.\lambda\sigma.\ (\mathcal{F}(E_1)\ \rho\ \sigma) + (\mathcal{F}(E_2)\ \rho\ \sigma)$

- composizionalità
  - la semantica di una espressione è definita per composizione delle semantiche delle sue sottoespressioni

Semantica delle espressioni : funzioni

E ::= I | val(I) | lambda(I, E_1) | plus(E_1, E_2) | apply(E_1, E_2)
env = IDE → dval
store = loc → mval
fun = store → dval → eval
eval = [int + bool + fun]
dval = [loc + int + bool + fun]
mval = [int + bool]
$\mathcal{F}$: EXPR → env → store → eval

makefun (lambda(I, E_1), ρ)= $\lambda\sigma'.\ \lambda d.\ \mathcal{F}(E_1)\ [\ \rho\ /\ I \leftarrow d\ ]\ \sigma'$

- $\mathcal{F}(lambda(I, E_1)) = \lambda\rho.\ \lambda\sigma.$ makefun (lambda(I, E_1), ρ)

applyfun (e, d, σ)= e σ d

- $\mathcal{F}(apply(E_1, E_2)) =$
  $\lambda\rho.\lambda\sigma.\ applyfun(\mathcal{F}(E_1)\ \rho\ \sigma, evaltodval\ (\mathcal{F}(E_2)\ \rho\ \sigma), \sigma)$
- composizionalità

Semantica dei comandi

C ::= ifthenelse(E, C_1, C_2) | while(E, C_1) | assign(I, E) | cseq(C_1, C_2)
env = IDE → dval
store = loc → mval
fun = store → dval → eval
eval = [int + bool + fun]
dval = [loc + int + bool + fun]
mval = [int + bool]

$\mathcal{C}$: COM → env → store → store

$\mathcal{C}$(assign(I, E)) = λρ.λσ. [σ / ρ(I) ← evaltomval($\mathcal{E}$(E_1) ρ σ)]

$\mathcal{C}$(cseq(C_1, C_2)) = λρ.λσ. $\mathcal{C}$(C_2) ρ ($\mathcal{C}$(C_1) ρ σ)

$\mathcal{C}$(ifthenelse(E, C_1, C_2)) = λρ.λσ. if $\mathcal{E}$(E) ρ σ then $\mathcal{C}$(C_1) ρ σ else $\mathcal{C}$(C_2) ρ σ

- composizionalità
- ragioniamo sulla semantica del while
  - per dare una semantica composizionale abbiamo bisogno di un punto fisso

Semantica del comando while

C ::= ifthenelse(E, C_1, C_2) | while(E, C_1) | assign(I, E) | cseq(C_1, C_2)
env = IDE → dval
store = loc → mval
fun = store → dval → eval
eval = [int + bool + fun]
dval = [loc + int + bool + fun]
mval = [int + bool]

$\mathcal{C}$: COM → env → store → store

- $\mathcal{C}$(while(E, C_1)) = λρ.λσ.
  (μf. λσ'. if $\mathcal{E}$(E) ρ σ' then f($\mathcal{C}$(C_1) ρ σ') else σ')
  σ

- f: store → store è il minimo punto fisso (μ) del funzionale, che, una volta calcolato, viene applicato allo store corrente σ

Semantica delle dichiarazioni

D ::= var(I, E) | dseq(D_1, D_2)
env = IDE → dval
store = loc → mval
fun = store → dval → eval
eval = [int + bool + fun]
dval = [loc + int + bool + fun]
mval = [int + bool]
$\mathcal{D}$: DEC → env → store → (env * store)

- $\mathcal{D}$(var(I, E)) = λρ.λσ. let loc = newloc() in
  ([ρ / I ← loc], [σ / loc ← $\mathcal{E}$(E) ρ σ])
- $\mathcal{D}$(dseq(D_1, D_2)) = λρ.λσ. let (ρ', σ') = $\mathcal{D}$(D_1) ρ σ in $\mathcal{D}$(D_2) ρ' σ'
- composizionalità

Semantica dei programmi

P ::= prog(D, C)
env = IDE → dval
store = loc → mval
fun = store → dval → eval
eval = [int + bool + fun]
dval = [loc + int + bool + fun]
mval = [int + bool]
$\mathcal{P}$: PROG → env → store → store

- $\mathcal{P}$(prog(D, C)) = λρ.λσ. let (ρ', σ') = $\mathcal{D}$(D) ρ σ in $\mathcal{C}$(C) ρ' σ'

Caratteristiche della semantica denotazionale

- composizionalità
  - la semantica di un costrutto è definita per composizione delle semantiche dei suoi componenti
- assegna una denotazione al programma (funzione sui domini semantici)

  $\mathcal{E}$: EXPR → env → store → eval
  $\mathcal{C}$: COM → env → store → store
  $\mathcal{D}$: DEC → env → store → (env * store)
  - senza aver bisogno di conoscere lo "stato"

- la costruzione della denotazione richiede un calcolo di minimo punto fisso, poichè le funzioni di valutazione semantica sono definite in modo ricorsivo
  - la semantica di un programma P è quindi $\mathcal{F}(P)\uparrow\omega$

La semantica denotazionale standard

- nella semantica denotazionale standard, si usa un terzo dominio semantico, le continuazioni
  - funzioni da store a store, simili a quella utilizzata nella semantica punto fisso del while
  - per trattare in modo composizionale costrutti di controllo "brutti", come i salti
  - lo stile delle definizioni cambia anche per gli altri costrutti
    - ed è meno naturale

La semantica operazionale

- lo stile di definizione tradizionale è quello dei sistemi di transizione
  - insieme di regole che definiscono
    - attraverso un insieme di relazioni di transizione
  - come lo stato cambia per effetto dell'esecuzione dei vari costrutti
- un esempio di regola nel caso dei comandi
  - configurazioni: triple $< \text{COM, env, store} >$
  - relazione di transizione: configurazione $\rightarrow_{com}$ store
  - una regola di transizione:

$$\frac{< C_1, \rho, \sigma > \rightarrow_{com} \sigma_2 \qquad < C_2, \rho, \sigma_2 > \rightarrow_{com} \sigma_1}{< cseq(C_1, C_2), \rho, \sigma > \rightarrow_{com} \sigma_1}$$

  - da confrontare con
    $\mathcal{C}(cseq(C_1, C_2)) = \lambda\rho.\lambda\sigma.\ \mathcal{C}(C_2)\ \rho\ (\mathcal{C}(C_1)\ \rho\ \sigma)$

Relazioni o funzioni di transizione?

- le transizioni si rappresentano più naturalmente con le relazioni
  - se il linguaggio permette transizioni nondeterministiche
    - programmazione logica e linguaggi concorrenti
- negli altri casi è possibile rappresentare le transizioni attraverso funzioni
  - simili alle funzioni di valutazione semantica dello stile denotazionale
- l'esempio nella composizione di comandi
  - operazionale (funzione di transizione)
    $\mathcal{C}(cseq(C_1, C_2), \rho, \sigma) = \mathcal{C}(C_2, \rho,\ \mathcal{C}(C_1, \rho, \sigma))$
  - operazionale (relazione di transizione):

$$\frac{< C_1, \rho, \sigma > \rightarrow_{com} \sigma_2 \qquad < C_2, \rho, \sigma_2 > \rightarrow_{com} \sigma_1}{< cseq(C_1, C_2), \rho, \sigma > \rightarrow_{com} \sigma_1}$$

  - denotazionale
    $\mathcal{C}(cseq(C_1, C_2)) = \lambda\rho.\lambda\sigma.\ \mathcal{C}(C_2)\ \rho\ (\mathcal{C}(C_1)\ \rho\ \sigma)$

Il nostro stile di specifica della semantica operazionale

- stesso formalismo (metalinguaggio) della semantica denotazionale
- stesso dominio sintattico e stessi domini semantici relativi allo stato
- prescindendo dalle superficiali differenze, relative al metalinguaggio, si evidenziano le differenze reali tra denotazionale ed operazionale
  - diverso "tipo" delle funzioni di valutazione semantica
    da $\mathcal{C}$: COM → env → store → store
    a $\mathcal{C}$: COM * env * store → store
  - cambiamento dei domini di valori "funzionali" (astrazioni funzionali)
    da fun = store → dval → eval
    a fun = EXPR * env
  - eliminazione del calcolo di punti fissi (while) (vedi dopo)
- tutti tre i casi possono essere ricondotti all'eliminazione di domini funzionali (di ordine superiore)
- negli ultimi casi, questo risultato si può ottenere solo utilizzando oggetti sintattici nei domini semantici e/o perdendo la composizionalità

Semantica delle espressioni

E ::= I | val(I) | lambda(I, E_1) | plus(E_1, E_2) | apply(E_1, E_2)
env = IDE → dval
store = loc → mval
fun = EXPR * env
eval = [int + bool + fun]
dval = [loc + int + bool + fun]
mval = [int + bool]
$\mathcal{E}$: EXPR ∗ env ∗ store → eval

- $\mathcal{E}$(I, ρ, σ) = dvaltoeval(ρ(I))
- $\mathcal{E}$(val(I), ρ, σ) = mvaltoeval(σ(ρ(I)))
- $\mathcal{E}$(plus(E_1, E_2), ρ, σ)= $\mathcal{E}$(E_1 , ρ, σ) + $\mathcal{E}$(E_2, ρ, σ)

makefun (lambda(I, E_1), ρ) = (lambda(I, E_1), ρ)
makefun (lambda(I, E_1), ρ)= λσ'. λd. $\underline{\mathcal{E}(E_1)\ [\ \rho\ /\ I \leftarrow d\ ]\ \sigma'}$

- $\mathcal{E}$(lambda(I, E_1)), ρ, σ) = makefun (lambda(I, E_1), ρ)

applyfun ((lambda(I, E_1), ρ), d, σ) = $\underline{\mathcal{E}(E_1)\ [\ \rho\ /\ I \leftarrow d\ ]\ \sigma}$
applyfun (e, d, σ) = e σ d

- $\mathcal{E}$(apply(E_1, E_2), ρ, σ)= applyfun($\mathcal{E}$(E_1 , ρ, σ) , evaltodval ($\mathcal{E}$(E_2, ρ, σ)), σ)

Semantica dei comandi

C ::= ifthenelse(E, C_1, C_2) | while(E, C_1) | assign(I, E) | cseq(C_1, C_2)
env = IDE → dval
store = loc → mval
fun = EXPR * env
eval = [int + bool + fun]
dval = [loc + int + bool + fun]
mval = [int + bool]
$\mathcal{C}$: COM ∗ env ∗ store → store

- $\mathcal{C}$(assign(I, E), ρ, σ) = [σ / ρ(I) ← evaltomval($\mathcal{E}$(E, ρ, σ))]
- $\mathcal{C}$(cseq(C_1, C_2), ρ, σ) = $\mathcal{C}$(C_2, ρ, $\mathcal{C}$(C_1, ρ, σ))
- $\mathcal{C}$(ifthenelse(E, C_1, C_2), ρ, σ) =
  if $\mathcal{E}$(E, ρ, σ) then $\mathcal{C}$(C_1, ρ, σ) else $\mathcal{C}$(C_2, ρ, σ)

Semantica del while

C ::= ifthenelse(E, C_1, C_2) | while(E, C_1) | assign(I, E) | cseq(C_1, C_2)
env = IDE → dval
store = loc → mval
fun = EXPR * env
eval = [int + bool + fun]
dval = [loc + int + bool + fun]
mval = [int + bool]
$\mathcal{C}$: COM ∗ env ∗ store → store

- $\mathcal{C}$(while(E, C_1), ρ, σ) =
  if $\mathcal{E}$(E, ρ, σ) then $\mathcal{C}$(cseq(C_1, while(E, C_1)), ρ, σ) else σ
- invece del minimo punto fisso, un'operazione sulla sintassi, che fa perdere la composizionalità

Semantica delle dichiarazioni

D ::= var(I, E) | dseq(D_1, D_2)
env = IDE → dval
store = loc → mval
fun = EXPR ∗ env
eval = [int + bool + fun]
dval = [loc + int + bool + fun]
mval = [int + bool]
$\mathcal{D}$: DEC ∗ env ∗ store → (env * store)

- $\mathcal{D}$(var(I, E), ρ, σ) = let loc = newloc() in
  ([ρ / I ← loc], [σ / loc ← $\mathcal{E}$(E, ρ, σ)])
- $\mathcal{D}$(dseq(D_1, D_2), ρ, σ) = let (ρ', σ') = $\mathcal{D}$(D_1, ρ, σ) in $\mathcal{D}$(D_2, ρ', σ')

Semantica dei programmi

P ::= prog(D, C)
env = IDE → dval
store = loc → mval
fun = EXPR ∗ env
eval = [int + bool + fun]
dval = [loc + int + bool + fun]
mval = [int + bool]
$\mathcal{P}$: PROG ∗ env ∗ store → store

- $\mathcal{P}$(prog(D, C), ρ, σ) = let (ρ', σ') = $\mathcal{D}$(D, ρ , σ) in $\mathcal{C}$(C, ρ', σ')

Caratteristiche della semantica operazionale

- non è sempre composizionale
- non assegna una denotazione al solo programma, ma definisce come si raggiunge (per un certo programma) lo "stato finale" a partire dallo stato iniziale

  $\mathcal{E}$: EXPR ∗ env ∗ store → eval
  $\mathcal{C}$: COM ∗ env ∗ store → store
  $\mathcal{D}$: DEC ∗ env ∗ store → (env * store)

- pur essendo le funzioni di valutazione semantica ancora definite in modo ricorsivo, non c'è bisogno di calcolare punti fissi
  - la semantica di un programma P in uno stato iniziale ρ, σ
  - $\mathcal{P}(P, \rho, \sigma)$ è lo stato finale (store), ottenuto "eseguendo" le transizioni, finché possibile

Semantica (denotazionale) e paradigmi 1

- linguaggi funzionali
- un solo dominio sintattico: EXPR
- un solo dominio semantico per lo stato: env
- dval = eval
  - i valori esprimibili contengono sempre fun
- una sola funzione di valutazione semantica

❧ $\mathcal{E}$: EXPR → env → eval

Semantica (denotazionale) e paradigmi 2

- linguaggi imperativi
- tre domini sintattici: EXPR, COM e DEC
- due domini semantici per lo stato: env e store
- dval, eval ed mval sono generalmente diversi
  - i valori su cui si interpretano le astrazioni funzionali (fun) sono di solito solo denotabili
  - le locazioni sono sempre denotabili
- tre funzioni di valutazione semantica

  $\mathcal{E}$: EXPR → env → store → eval
  $\mathcal{C}$: COM → env → store → store
  $\mathcal{D}$: DEC → env → store → (env * store)

Semantica (denotazionale) e paradigmi 3

- linguaggi orientati ad oggetti
- oltre ad: EXPR, COM e DEC, dichiarazioni di classe
- tre domini semantici per lo stato: oltre a env e store, un dominio nuovo (heap), per modellare i puntatori e gli oggetti
- dval, eval ed mval sono generalmente diversi
  - i valori su cui si interpretano le astrazioni funzionali (fun) sono di solito solo denotabili
  - le locazioni sono sempre denotabili
  - gli oggetti di solito appartengono a tutti e tre i domini
- le funzioni di valutazione semantica, prendono (e restituiscono) anche la heap

  $\mathcal{C}$: COM → env → store → heap → (store ∗ heap)

Semantica (denotazionale) e supporto a run time

- le differenze fra i linguaggi dei diversi paradigmi si riflettono in differenze nelle corrispondenti implementazioni
- tutte le caratteristiche importanti per progettare un interprete o un supporto a tempo di esecuzione, si possono ricavare ispezionando i soli domini semantici della semantica denotazionale

Verso definizioni semantiche "eseguibili"

- abbiamo usato lo stesso metalinguaggio per i due stili
- quali caratteristiche sono importanti?
  - la λ-astrazione
    - con funzioni di ordine superiore
  - i tipi: il meccanismo naturale per definire domini sintattici e semantici
    - via enumerazione, somma, prodotto, iterazione, funzioni
  - la definizione di funzioni per casi
    - sui casi di un tipo (la sintassi astratta)
  - un operatore per il calcolo del punto fisso (μ)
- il metalinguaggio è un frammento di un vero e proprio linguaggio di programmazione (funzionale, di ordine superiore, tipato, con pattern matching)
  - riesprimendo le nostre semantiche in tale linguaggio, otteniamo semantiche eseguibili
    - persino la semantica denotazionale è un'implementazione