
# PROGRAMMAZIONE 2
# 17. Interpreti, compilatori e semantica operazionale

# Linguaggi di programmazione


- Come si comprendono le caratteristiche di un linguaggio di programmazione?
- Molte risposte diverse…
  - manuali, documentazione on-line, esempi, consultazione stackoverflow, …
- La nostra risposta
  - la comprensione di un linguaggio di programmazione si ottiene dalla **semantica** del linguaggio
- **Semantica** come guida alla progettazione, all'implementazione e all'uso di un linguaggio di programmazione


# Elementi di semantica operazionale

# Sintassi e semantica


- Un linguaggio di programmazione possiede
  - Una **sintassi**, che definisce
    - le "formule ben formate" del linguaggio, cioè i programmi sintatticamente corretti, tipicamente generati da una grammatica
  - una **semantica**, che fornisce
    - un'interpretazione dei "token" in termini di entità (matematiche) note
    - un significato ai programmi sintatticamente corretti
- La **teoria dei linguaggi formali** fornisce formalismi di specifica (grammatiche) e tecniche di analisi (automi) per trattare aspetti sintattici
- Per la **semantica** esistono diversi approcci
  - **denotazionale, operazionale, assiomatica, …**
- La semantica formale viene di solito definita su una rappresentazione dei programmi in **sintassi astratta**

# Sintassi concreta


- La **sintassi concreta** di un linguaggio di programmazione è definita di solito da una **grammatica libera da contesto** (come visto a **PR1**)
- ***Esempio***: grammatica di semplici **espressioni logiche** (in Backus-Naur Form, BNF)
  - **e ::= v | Not e | (e And e)| (e Or e) | (e Implies e)**
  - **v ::= True | False**
- Notazione comoda per programmatori (operatori infissi, associatività-commutatività di operatori, precedenze)
- Meno comoda per una gestione computazionale (si pensi ai problemi di ambiguità)

# Sintassi astratta


- L'**albero sintattico** (***abstract syntax tree***) di una espressione exp mostra (risolvendo le ambiguità) come exp può essere generata dalla grammatica
- La **sintassi astratta** è una rappresentazione lineare dell' **albero sintattico**
  - gli operatori sono nodi dell'albero e gli operandi sono rappresentati dai sottoalberi
- Per gli AST abbiamo quindi sia una notazione lineare che una rappresentazione grafica

# Dalla sintassi concreta all' AST

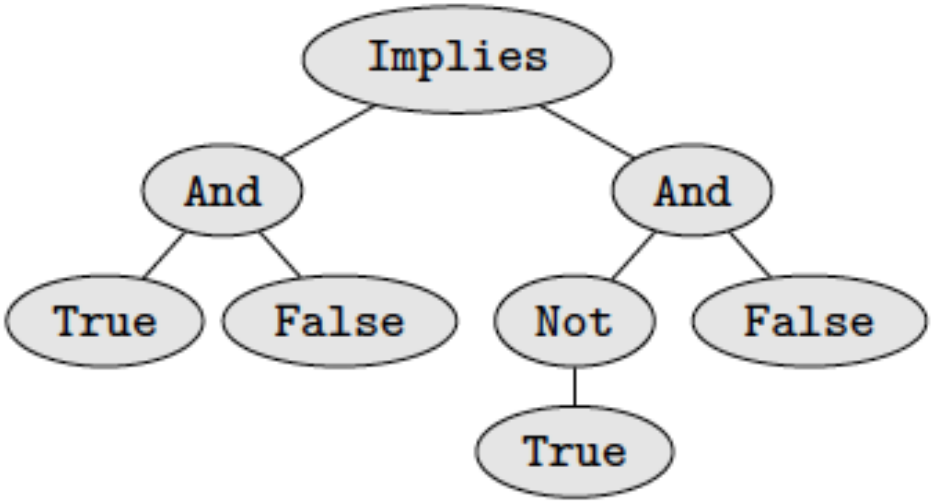
**(True And False) Implies**
**((Not True) And False)**

SINTASSI CONCRETA

**Implies(And(True,False),**

**And(Not(True),False) )**

SINTASSI ASTRATTA




ALBERO SINTATTICO

# **Semantica** dei linguaggi


- Tre metodi principali per definire **semantica**
  - ***semantica operazionale***: descrive il significato di un programma in termini dell'evoluzione (cambiamenti di stato) di una macchina astratta
  - ***semantica denotazionale***: il significato di un programma è una funzione matematica definita su opportuni domini
  - ***semantica assiomatica***: descrive il significato di un programma in base alle proprietà soddisfatte prima e dopo la sua esecuzione
- Ogni metodo ha vantaggi e svantaggi rispetto ad aspetti matematici, facilità di uso nelle dimostrazioni, o utilità nel definire un interprete o un compilatore per il linguaggio

# E di questo cosa si vede in PR2?


- Metodologie per OOP
  - qualcosa di ***semantica assiomatica***, informalmente
  - clausole Requires & Effect
  - Representation Invariant
- Comprensione dei paradigmi dei linguaggi di programmazione
  - useremo una ***semantica operazionale***

# Semantica operazionale


- Idea: la semantica operazionale di un linguaggio **L** definisce ***in modo formale, con strumenti matematici,*** una macchina astratta $\mathbf{M_L}$ in grado di eseguire i programmi scritti in **L**
- Definizione: un sistema di transizioni è costituito da
  - un insieme Config di configurazioni (stati)
  - una relazione di transizione $\longrightarrow \subseteq \text{Config} \times \text{Config}$
- Notazione: $c \longrightarrow d$ significa che c e d sono nella relazione $\longrightarrow$
- Intuizione: $c \longrightarrow d$ se lo stato c evolve nello stato d

# Semantica operazionale "small step"

- Nella semantica operazionale "small step" la relazione di transizione $\longrightarrow$ descrive un passo del processo di calcolo
- Abbiamo una transizione $e \longrightarrow d$ se partendo dall'espressione (o programma) e l'esecuzione di un passo di calcolo ci porta nell'espressione d
- Una valutazione completa di e avrà quindi la forma

  $$e \longrightarrow e_1 \longrightarrow e_2 \ldots \longrightarrow e_n$$

  dove $e_n$ può rappresentare il valore finale di e
- Nella semantica *small-step* la valutazione di un programma procede attraverso le configurazioni intermedie che può assumere il programma

# Semantica operazionale "big step"


- Nella semantica operazionale "big step" la relazione di transizione descrive la valutazione completa di un programma/espressione
- Scriviamo e => v se l'esecuzione del programma /espressione e produce il valore v
- Notazione alternativa (equivalente) utilizzata in molti testi: e ⇓ v
- Come vedremo, una valutazione completa di una espressione è ottenuta componendo le valutazioni complete delle sue sotto-espressioni

# Semantica operazionale in PR2


- Utilizzeremo la semantica operazionale “big step” come modello per descrivere i meccanismi di calcolo dei linguaggi di programmazione
- La semantica operazionale “small step” sarebbe utile nel caso di linguaggi concorrenti per descrivere le comunicazioni e proprietà quali la deadlock freedom

# Semantica operazionale "big step"

$$true \Rightarrow true$$

$$false \Rightarrow false$$

**VALORI**

$$\frac{e \Rightarrow v}{not\ \mathrm{e} \Rightarrow \neg v}(not)$$

$$\frac{e1 \Rightarrow v1 \quad e2 \Rightarrow v2}{e1\ \mathrm{and}\ e2 \Rightarrow v1 \wedge v2}(and)$$

**Regole di valutazione analoghe per OR e IMPLIES**
**Usiamo OPERATORI LOGICI sul dominio dei valori con tabelle di verità**

# Regole e derivazioni

- Le regole di valutazione possono essere composte per ottenere la valutazione di una espressione più complessa
- Questo fornisce una prova di una derivazione operazionale di calcolo

$$\dfrac{\dfrac{}{\texttt{True} \Rightarrow \texttt{True}} \qquad \dfrac{\dfrac{}{\texttt{False} \Rightarrow \texttt{False}} \quad \dfrac{}{\texttt{True} \Rightarrow \texttt{True}}}{\texttt{False And True} \Rightarrow \texttt{False}}}{\texttt{True And (False And True)} \Rightarrow \texttt{False}}$$

# Proof System

- Le regole di valutazione costituiscono un *proof system* (sistema di dimostrazione)

$$\frac{premessa_1 \quad \ldots \quad premessa_k}{conclusione}$$

- Tipicamente le regole sono definite per induzione strutturale sulla sintassi del linguaggio
- Le "formule" che ci interessa dimostrare sono transizioni del tipo e => v
- Componiamo le regole in base alla struttura sintattica di e ottenendo un *proof tree*

# Dalle regole all' interprete


- Le regole di valutazione definiscono l'interprete della macchina astratta "formale" definita dalla semantica operazionale
- Quindi le regole descrivono il processo di calcolo
- Nel corso forniremo una codifica **OCaML** della semantica operazionale
- Di conseguenza otterremo un modello eseguibile dell'interprete del linguaggio


# Interprete espressioni logiche

# Passo 1: sintassi astratta

```
type BoolExp =
    | True
    | False
    | Not of BoolExp
    | And of BoolExp  * BoolExp
```

**Definizione della sintassi astratta tramite i tipi algebrici di OCaML**

# Passo 2: dalle regole all'interprete

- Vogliamo definire una funzione **eval** tale che `eval(e)=v` se e solo se `e => v`
- Esempio: dalla regola

$$\frac{e \Rightarrow v}{not\ \mathrm{e} \Rightarrow \neg v}$$

otteniamo il seguente codice **OCaml**

```
eval Not(exp0) -> match eval exp0 with
                     True -> False
                   | False -> True
```

# Interprete di espressioni logiche


```
let rec eval exp =
      match exp0 with
        True -> True
      | False -> False
      | Not(exp0) -> (match eval exp0 with
                           True -> False
                         | False -> True)
      | And(exp0, exp1) ->
            (match (eval exp0, eval exp1) with
               (True,True) -> True
             | (_,False) -> False
             | (False,_) -> False)
```


# Espressioni a valori interi

# Sintassi OCaML (astratta)


```
type exp =
    Cnst of constant //costanti intere
  | Var of variable // variabili
  | Op of exp * operand* exp
                // operatori binari
```

# Ambiente (1)

- Per definire l'interprete introduciamo una ***struttura di implementazione (run-time structure)*** che permetta di recuperare i valori associati agli identificatori
- Un *binding* è un'associazione tra un nome e un valore
  - il nome solitamente è utilizzato per reperire il valore
  - esempio in ML

```
let x = 2 + 1 in
   let y = x + x in
      x * y
```

  - il binding di x è 3, il binding di y è 6, il valore calcolato dal programma è 18

# Ambiente (2)

- Un ambiente è una collezione di binding
- Esempio **env = {x -> 25, y -> 6}**
- L'ambiente **env** contiene due "binding"
  - l'associazione tra l'identificatore **x** e il valore **25**
  - l'associazione tra l'identificatore **y** e il valore **6**
  - l'identificatore **z** non è legato nell'ambiente
- Astrattamente (come in LPP) un ambiente è una funzione di tipo

  **Ide → Value + Unbound**
- L'uso della costante **Unbound** permette di rendere la funzione totale

# Notazione: come in LPP


- Dato un ambiente **env: Ide → Value + Unbound**
- **env(x)** denota il valore **v** associato a **x** nell'ambiente oppure il valore speciale **Unbound**
- **env[v/x]** indica l'ambiente così definito
  - **env[v/x](y) = v** se **y = x**
  - **env[v/x](y)** = **env(y)** se **y != x**
- Esempio: se **env = {x -> 25, y -> 7}** allora

  **env[5/x] = {x -> 5, y -> 7}**

# Implementazione: lista di coppie


```
let emptyenv = []
(* the empty environment *)

let rec lookup env x =
    match env with
    | []          -> failwith ("not found")
    | (y, v)::t -> if x = y then v
                   else lookup t x
```

# Regole di valutazione: codice interprete

$$\frac{env(x) = v}{env \rhd Var\ \mathrm{x} \Rightarrow v}$$

```
eval Var x env -> lookup x env
```

$$\frac{env \rhd e1 \Rightarrow v1 \quad env \rhd e2 \Rightarrow v2}{env \rhd \mathrm{Op}(e1, \mathrm{Plus}, e2) \Rightarrow v1 + v2}$$

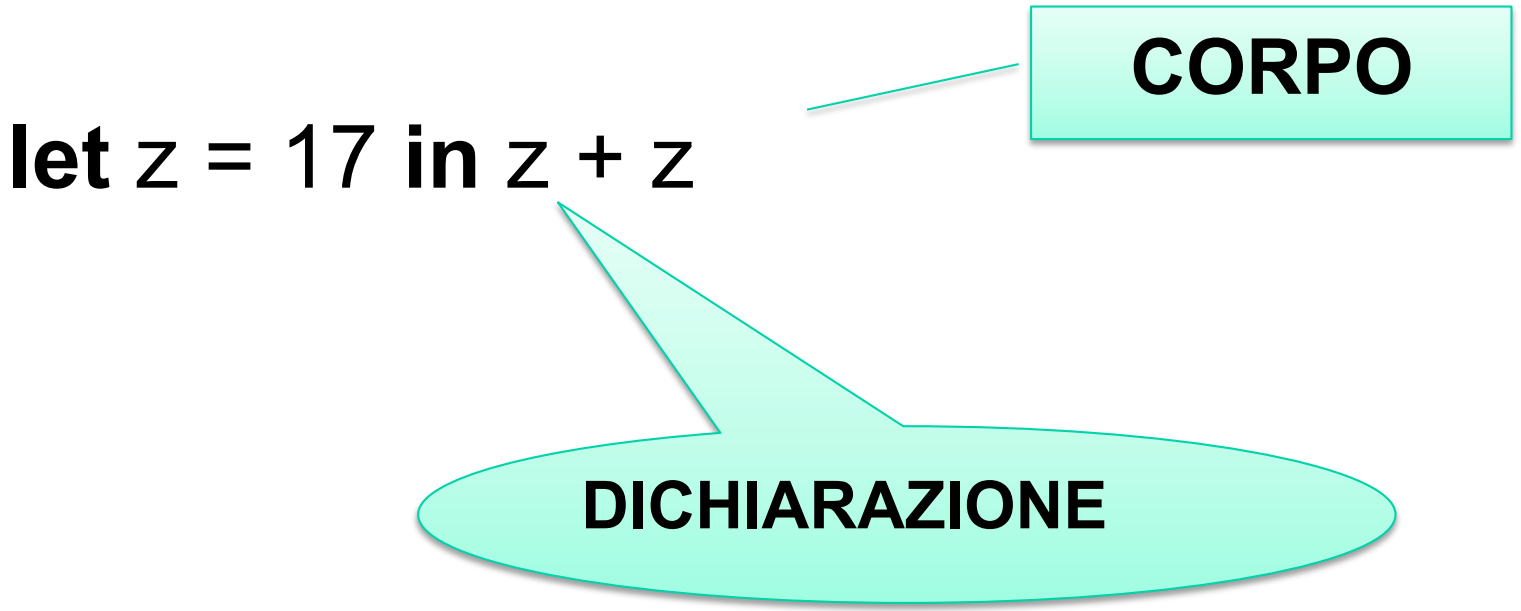
```
eval Op(e1, Plus, e2) env ->
          eval  e1 env + eval  e2 env
```

# Interprete completo per semplici espressioni intere


(* la valutazione è parametrica rispetto a env ma non viene modifcato*)

```
let rec eval (e : exp) (env : (variable * int) list) : int =
    match e  with
     | Cnst i                  -> i
     | Var x                   -> lookup x env
     | Op(e1, Plus, e2)    -> eval e1 env + eval e2 env
     | Op (e1, Minus, e2)   -> eval e1 env - eval e2 env
     | Op (e1, Times, e2)   -> eval e1 env * eval e2 env
     | Op(e1, Div, e2)     -> eval e1 env \ eval e2 env
```

# Aggiungiamo le dichiarazioni



# Espressioni con dich.: sintassi astratta


```
type exp =
 Cnst of constant
   | Var of variable
   | Op of exp * operand * exp
    | Let of variable * exp  * exp
```

**Esempio**

```
Let("z", Cnst 17, Op(Var "z", Plus, Var "z"))
```

*In sintassi concreta*

```
let z = 17 in z + z
```

# Regola di valutazione del Let

$$\frac{env \triangleright erhs \Rightarrow xval \quad env[xval / x] \triangleright ebody \Rightarrow v}{env \triangleright \text{Let x = erhs in ebody} \Rightarrow v}$$

- Si valuta **ehrs** nell'ambiente corrente *env* ottenendo **xval**
- Si valuta **ebody** nell'ambiente *env* esteso con il legame tra **x** e **xval** ottenendo il valore **v**
- La valutazione del "let" nell'ambiente corrente produce il valore **v**

# Interprete esteso con espressioni

## Let

```
eval (Let(x, erhs, ebody)) env ->
      let xval = eval erhs env
          in
        let env1 = (x, xval) :: env in
          eval  ebody env1
```

# Interprete per espressioni con dich.


```
let rec eval (e : exp) (env : (variable * int) list) :
int =
    match e with
    | Cnst i                -> i
    | Var x                 -> lookup env x
    | Let(x, erhs, ebody) ->
        let xval = eval erhs env in
            let env1 = (x, xval) :: env in
                eval ebody env1
    | Op(e1, Plus, e2)    -> eval e1 env + eval e2 env
    | Op(e1, Minus, e2)   -> eval e1 env - eval e2 env
    | Op(e1, Times, e2)   -> eval e1 env * eval e2 env
    | Op(e1, Div, e2)     -> eval e1 env \ eval e2 env
```

# Variabili libere

- In logica una variabile in una formula è *libera* se non compare nella portata di un quantificatore associato a tale variabile, altrimenti è *legata*

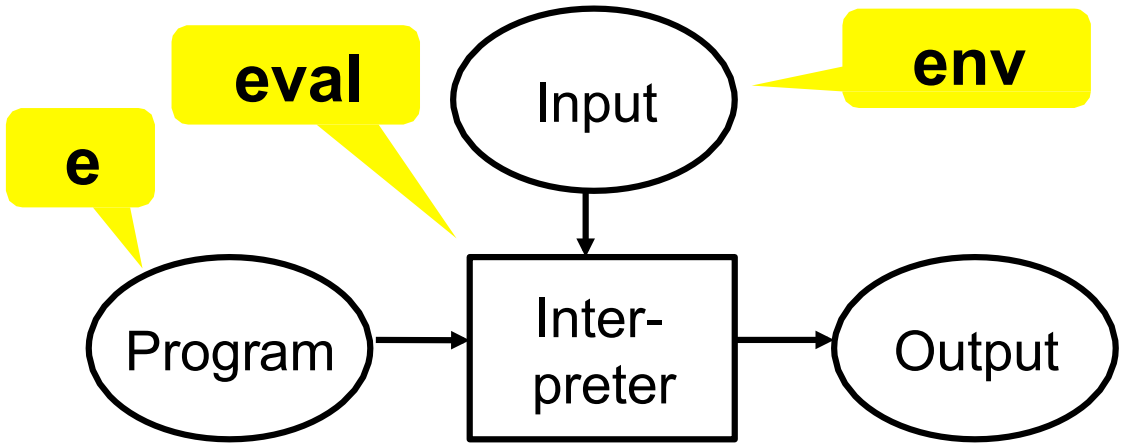
- Esempio: $\forall x.(P(x) \wedge Q(y))$
  [ o $(\forall x.\ P(x) \wedge Q(y))$ nella sintassi di LPP ]
  - x è legata
  - y è libera

# Occorrenze libere

- La nozione di variabile libera o legata si applica anche al caso del costrutto **let**
- Infatti il costrutto **let** si comporta come un quantificatore per la variabile che introduce
- Un identificatore **x** si dice "legato" se appare nel **ebody** dell'espressione **let x = ehrs in ebody**, altrimenti si dice libero
- Esempi
  - `let z = x in z + x` (* z legata, x libera *)
  - `let z = 3 in let y = z + 1 in x + y`
    (* z, y legate, x libera *)

# Interpretazione di espressioni

- L'interprete introdotto ci permette di valutare espressioni costruite con la sintassi indicata
- Il valore di una espressione chiusa non dipende da env



# Verso la compilazione


- Il nostro interprete di espressioni ogni volta che deve determinare il valore associato a una variabile effettua una operazione di lookup nell'ambiente: questo potrebbe essere oneroso
- Idea: ottimizzare l'esecuzione introducendo un piccolo compilatore che traduce tutte le occorrenze di identificatori in "indici di accesso", in modo tale che l'operazione di lookup sia eseguita senza effettuare ricerche sull'ambiente, ma in modo diretto

# Le variabili: da nomi a indici

```
Let("z", Cnst 17, Op(Var "z", Plus, Var "z"))
```

**COMPILAZIONE**

```
Let(Cnst 17, Op(Var 0, Plus, Var 0))
```

**Il valore 0 indica il binding (let) più vicino**

# Indici per variabili

**Idea**: indice di una variabile =
numero dei **let** che si attraversano per raggiungerla

```
Let("z", Cnst 17,
        Let("y", Cnst 25,
                Op(Var "z", Plus, Var "y")))
```

**COMPILAZIONE**

```
    Let(Cnst 17,
            Let(Cnst 25,
                    Op(Var 1, Plus, Var 0)))
```

# Indici per variabili


- L'idea di utilizzare indici al posto di variabili in modo tale da avere implementazioni efficienti nasce nella teoria del lambda-calcolo con gli indici di De Bruijn

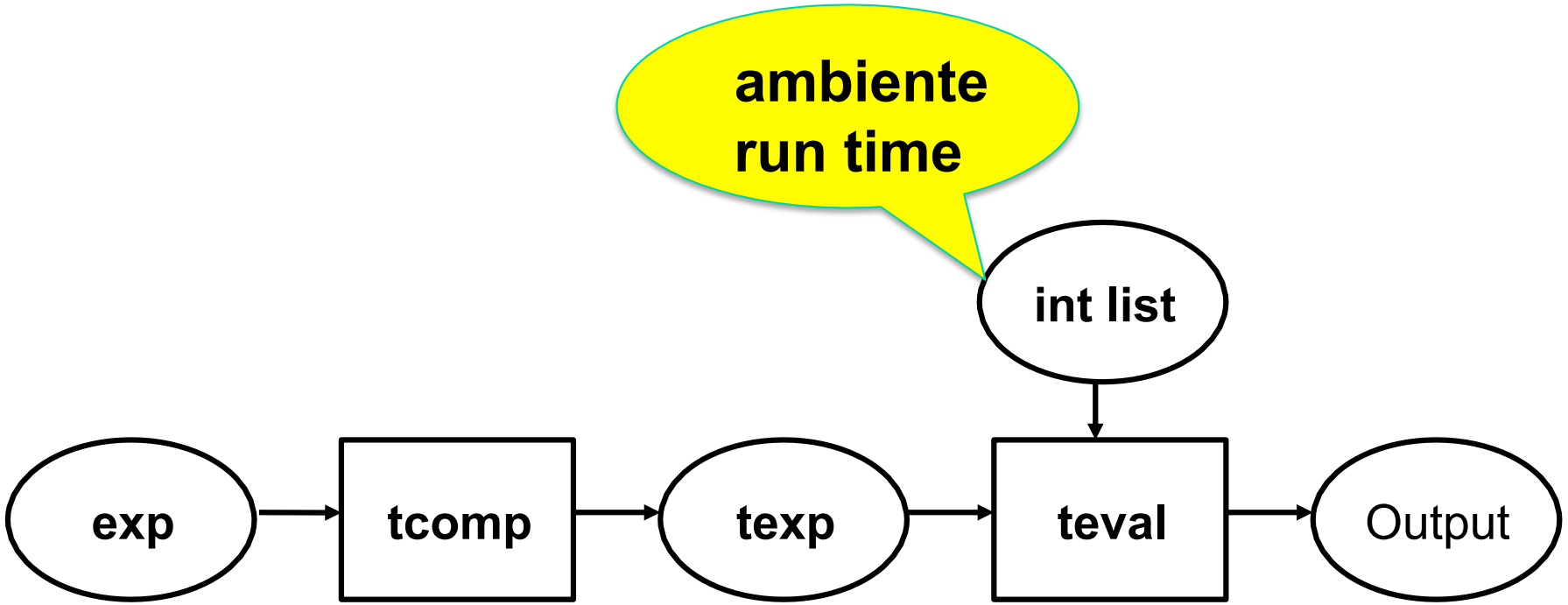
  http://en.wikipedia.org/wiki/De_Bruijn_index

# Il linguaggio intermedio


## TARGET EXPRESSION

```
type texp =          (* espressioni target *)
   | TCnst of constant
   | TVar of int     (* indice a run time *)
   | TLet of texp * texp   (* erhs e ebody *)
   | TOp of texp * operand * texp
```

# Compilazione in codice intermedio



# Compilazione in codice intermedio


```
let rec tcomp (e : texp ) (cenv : variable list) : texp =
    match e with
      | Cnst i -> TCnst i
      | Var x  -> TVar (getindex cenv x)
      | Let(x, erhs, ebody) ->
            let cenv1 = x :: cenv in
               TLet(tcomp  erhs cenv, tcomp  ebody cenv1)
      | Op(e1, op, e2) ->
               TOp(tcomp  e1 cenv, op, tcomp e2 cenv)

let rec getindex cenv  x =
    match cenv with
    | [] -> failwith("Variable not found")
    | y::yr -> if x=y then 0 else 1 + getindex yr x
```

# Interprete del. in codice intermedio


```
Open List

let rec teval (e : texp) (renv : int list) : int =
    match (e  with
       | TCnst i -> i
       | TVar n  -> List.nth renv n
       | TLet(erhs, ebody) ->
            let xval = teval erhs renv in
               let renv1 = xval :: renv  in
                  teval ebody renv1
       | Op(e1, Plus, e2) -> teval e1 renv + teval e2 renv
       | Op(e1, Minus, e2) -> teval e1 renv - teval e2 renv
       | Op(e1, Times, e2) -> teval e1 renv * tevale2 renv
       | Op(e1, Div, e2) –teval e1 renv teval e2 renv
```

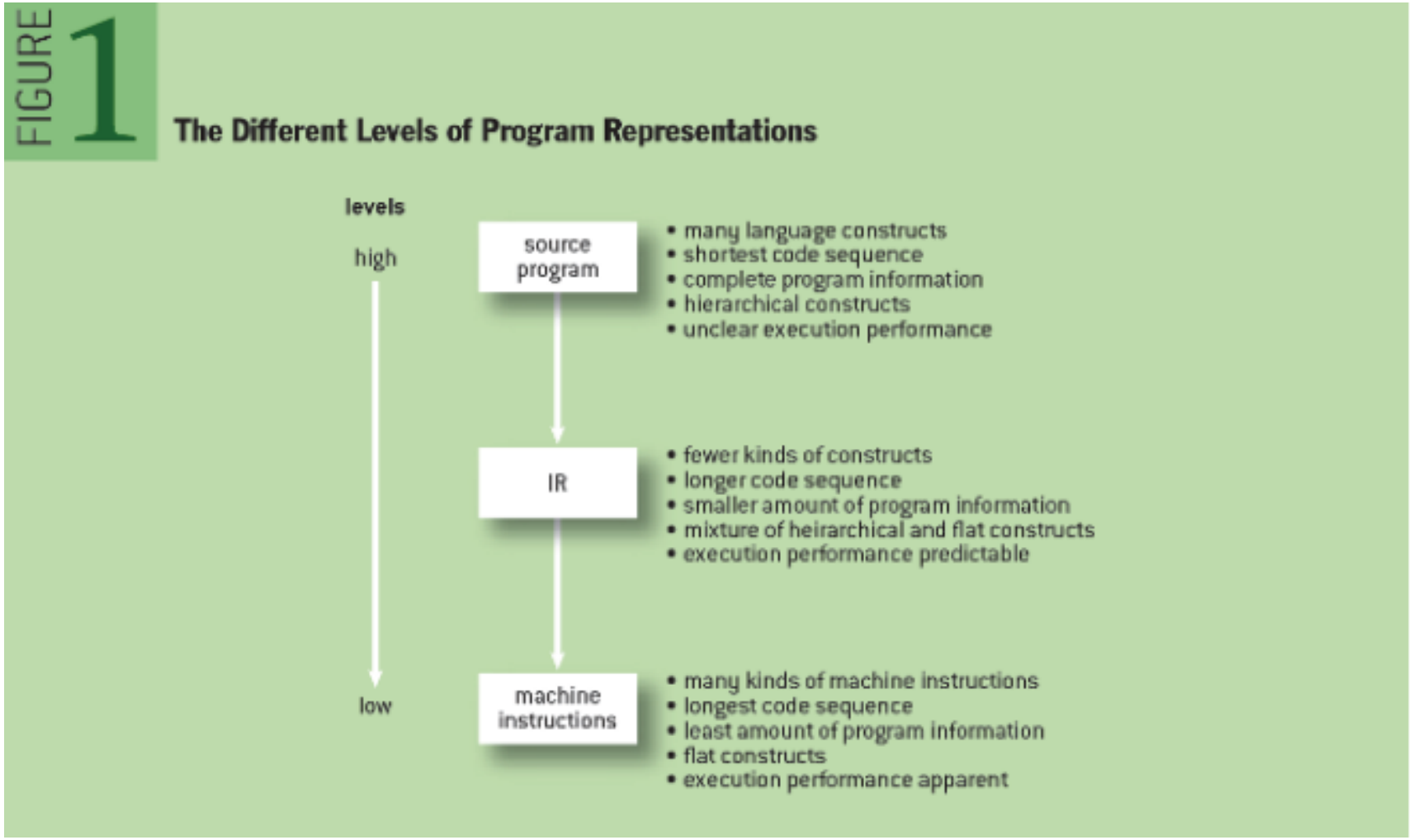
# Codice intermedio


- Rappresentare il programma sorgente in un codice intermedio permette di dominare la complessità della implementazione di un linguaggio di programmazione
- La rappresentazione in codice intermedio permette di effettuare numerose ottimizzazioni sul codice (nel nostro caso, l'eliminazione dei nomi a run-time)
- Esempi
  - Java bytecode: codice intermedio della JVM
  - Microsoft Common Intermediate Language: codice intermedio .NET

# Cosa abbiamo imparato


- Una tecnica generale usata nei back-end dei compilatori nella fase di generazione del codice per ottenere un codice maggiormente efficiente
- Usare codice intermedio e ottimizzare il codice oggetto sul codice intermedio
- Articolo divulgativo (ma tecnico) disponibile on-line: Fred Chow, Intermediate Representation, Communications of ACM 56 (12) 2013

# Visione Java



# Visione Common Intermediate Language