Semantica operazionale per type-checking

A cosa servono i tipi?

- A garantire che un programma abbia comportamenti corretti rispetto ai valori sui quali opera
  - non si somma un valore di tipo intero con un valore di tipo stringa: 25 + "abcd"
  - le variabili sono dichiarate prima del loro utilizzo all'interno di un programma
- A ottimizzare la generazione del codice
  - viene occupata esattamente la memoria che serve per i valori

Controlli statici

- La compilazione in un codice intermedio più efficiente è solo uno degli aspetti significativi dell'utilizzo di tecniche di compilazione
- Oltre agli aspetti di analisi sintattica e ottimizzazione del codice, il compilatore effettua diversi controlli di analisi statica sulla struttura del codice
- Esempio: **type checking** per l'analisi statica dell'uso corretto dei tipi associati ai costrutti linguistici
- Vediamo un semplice esempio basato sul nostro linguaggio di espressioni

Type checking

- Cosa controlla un type checker (anche se ovviamente dipende dal tipo di linguaggio)
  - chiamata di una funzione con un numero errato di parametri attuali
  - uso di variabili non dichiarate
  - funzioni che non restuiscono valori
  - “array out of bound indices” (pensate al C)
  - :

Come descrivere un type checker?

- Associazione di tipo ai costrutti di un linguaggio di programmazione è un asserzione della forma

$$env \triangleright \mathrm{e} \Rightarrow T$$

- Nell'ambiente env l'espressione e ha tipo T
- L'ambiente contiene le associazioni tra identificatori e tipi

Regole di tipo

- Le regole di tipo costituiscono un *proof system*

$$\frac{J_1 \quad \ldots \quad J_k}{J}$$

- Le regole sono definite per induzione strutturale sulla sintassi del linguaggio
  - esattamente come abbiamo operato nel caso dell'interprete

Regole di tipo

- Una regola del tipo

$$\frac{J_1 \quad \ldots \quad J_k}{J}$$

può essere letta dal basso verso l'alto

  - per controllare (check) J dobbiamo controllare J1 … Jk

Type checking vs. Type inference

- **Type checking**: data un'asserzione env => e : T, si deve determinare se l'asserzione può essere derivata applicando le regole di tipo

- **Type inference**: dato un programma e, si deve determinare un tipo T e un opportuno ambiente env tale che env => e : T può essere derivato

Type checking vs. Type inference

- Java, C#: principalmente type checking
  - anche se alcuni aspetti sui generici possono richiedere type inference
- ML, Haskell: type inference

Typed expression

- Modifichiamo leggermente la sintassi del nostro linguaggio introducendo costanti con tipo intero e booleano e un costrutto condizionale

```
type tyexpr =
   | CstI of int
   | CstB of bool
   | Var of string
   | Let of string * tyexpr * tyexpr
   | Prim of string * tyexpr * tyexpr
   | If of tyexpr * tyexpr * tyexpr
```

Typed expression

- Con questa sintassi possiamo costruire programmi che non sono tipati correttamente
  - la guardia del condizionale non è booleana
  - un operatore è applicato a argomenti di tipo errato
  - i due rami del condizionale non hanno lo stesso tipo

```
Let("z", CstI 17,
    Let("y", CstB true,
        If(Var "z",
           Prim("+", Var “y", Var "y"),
                Var "z")))
```

Non è un valore booleano !

Somma di booleani !

I tipi

- Il tipo è una proprietà che caratterizza i valori che sono manipolati dal programma
  - “la variabile x contiene valori di tipo stringa”
  - “int è il tipo del valore restituito dalla funzione f”
- L’informazione di tipo è uno strumento utile nello sviluppo di programmi perché permette di evitare alcuni errori di programmazione

Tipi: statico vs. dinamico

- Il **controllo statico dei tipi** permette di verificare la correttezza dal programma rispetto alle proprietà dei tipi staticamente, cioè prima di mandare il programma in esecuzione
  - Ocaml, Java, C#, … sono linguaggi con controllo statico
- Il **controllo dinamico dei tipi** verifica le proprietà dei tipi a tempo di esecuzione
  - Lisp, Smalltalk, JavaScript, … sono linguaggi con controllo dinamico

Java: controllo dei tipi dinamico

- La divisione statico/dinamico non è sempre netta
- Consideriamo Java
  - le operazioni di “downcast” sono verificate a run-time e possono sollevare eccezioni.
  - l’accesso agli elementi degli array in Java è controllato a run-time
- La **ClassCastException** viene sollevata dal controllo dinamico dei tipi in Java: codice “unsafe” rispetto all’uso dei tipi non è dunque mai eseguito in Java

Esempio: checking dinamico in Java


```
class Calculation {
    public int f(int x) { return x; }
}
class Person {
    public String getName( ) { return "persona"; }
}
class Main {
    public static void main(String[] args) {
      Object o = new Calculation( );
      System.out.println(((Person) o).getName( ));
    }
}
```

Exception in thread "main"
java.lang.ClassCastException:
Calculation at Main.main(example.java:12)

Meglio tardi che mai…

- D'altra parte, in C o C++ il tipo base degli array (int, float) è controllato staticamente, ma
  - l'accesso agli elementi degli array non viene controllato neanche a runtime
- Anche il downcasting viene controllato a runtime
  - provate a riscrivere in C++ il programma Java che abbiamo visto in precedenza, e vedrete che il programma viene eseguito con esiti non prevedibili
- Il punto: C++ non effettua controlli dinamici di tipo

Typed expression

- Per controllare l'utilizzo dei tipi staticamente abbiamo bisogno di una struttura di implementazione che associa ai valori presenti nel programma il loro tipo
- *Ambiente di tipo*: è la struttura che associa alle variabili presenti nel programma il loro tipo
  - è un ambiente dove i valori sono i tipi
  - nel contesto dei compilatori viene anche chiamata ***Tabella dei Simboli***

I tipi del nostro esempio OCaml

- Obiettivo: definire una funzione che data una espressione con tipi (`typexpr`)
  - restituisce il tipo dell'espressione, se è ben tipata
  - altrimenti lancia un'eccezione

```
type typ =
    | TypI                    (* int *)
    | TypB                    (* bool *)
```

Serve un ambiente dei tipi

- … e binding (string, typ)
- Definizione identica a quella degli ambienti già visti, che contenevano binding (string, int)
- Sfruttiamo i tipi polimorfi di OCaml

```
type 't env = (string * 't) list

let rec lookup amb y = match amb with
      | (i1, e1) :: amb1 ->
              if y = i1 then e1 else lookup amb1 y
      | [] -> failwith("wrong typed env")
```

Type checking (I)

```
let rec typ (e : tyexpr) (env : typ env) : typ =
    match e with
     | CstI i -> TypI
     | CstB b -> TypB
     | Var x  -> lookup env x
     | Prim(ope, e1, e2) ->
        let t1 = typ e1 env in
         let t2 = typ e2 env in
          match (ope, t1, t2) with
           | ("*", TypI, TypI) -> TypI
           | ("+", TypI, TypI) -> TypI
           | ("-", TypI, TypI) -> TypI
           | ("=", TypI, TypI) -> TypB
           | ("<", TypI, TypI) -> TypB
           | ("&", TypB, TypB) -> TypB
           | _ -> failwith "unknown op, or type error"
```

Type checking (II)


```
| Let(x, eRhs, letBody) ->
      let xTyp = typ eRhs env in
       let letBodyEnv = (x, xTyp) :: env in
        typ letBody letBodyEnv
| If(e1, e2, e3) ->
      match typ e1 env with
      | TypB -> let t2 = typ e2 env in
                 let t3 = typ e3 env in
                  if t2 = t3 then t2
                  else failwith "If: branch types differ"
      | _    -> failwith "If: condition not boolean”
```

Type checking

- Si noti come il type checker utilizzi le medesime regole della semantica operazionale, ma utilizzi come dominio dei valori l'insieme dei tipi e come ambiente l'ambiente dei tipi
- Type checker: interprete del linguaggio su un dominio astratto di valori

Semantica operazionale del Let

- Regola di esecuzione

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

- Regola di type checking

$$\frac{tenv \triangleright ehrs \Rightarrow tval \quad tenv[tval / x] \triangleright ebody \Rightarrow t}{tenv \triangleright \text{Let x = erhs in ebody} \Rightarrow t}$$

Implementazione in OCaml del Let


```
typ Let(x, eRhs, letBody) tenv ->
      let xTyp = typ eRhs tenv in
        let letBodyEnv = (x, xTyp) :: tenv in
          typ letBody letBodyEnv
```

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

Sistema di tipi

- I linguaggi moderni prevedono di associare tipi con i valori manipolati dai costrutti linguistici
- ***Sistema di tipi***: il complesso delle informazioni che regolano i tipi nel linguaggio di programmazione
  - tipi predefiniti
  - meccanismi per definire e associare un tipo ai costrutti
  - regole per definire equivalenza, compatibilità e inferenza
- Un sistema di tipi per un linguaggio è detto ***type safe*** quando nessun programma può violare le distinzioni tra i tipi del linguaggio
  - nessun programma durante l'esecuzione può generare un errore che derivi da una violazione di tipo

Type checking

- Strumento che assicura che un programma segue le regole di compatibilità dei tipi

- Un linguaggio è **strongly typed** se evita l'uso non conforme ai tipi richiesti delle operazioni del linguaggi
- Un linguaggio è ***statically typed*** se è *strongly typed* e il controllo dei tipi viene fatto staticamente
- Un linguaggio è ***dynamically typed*** se il controllo dei tipi viene fatto a run time

Inferenza di tipo

- **Type inference:** il meccanismo di inferenza di tipi consente di dedurre il tipo associato a un programma senza bisogno di dichiarazioni esplicite
- OCaml

```
# let revPair (x, y) = (y, x);;
val revPair : 'a * 'b -> 'b * 'a = <fun>
```

Come opera l'inferenza?

```
# let f x = 2 + x ;;
val f : int -> int = <fun>
```

1. Quale è il tipo di f?
2. L'operatore + ha due tipi
   int → int → int,
   real → real → real,
3. La costante 2 è di tipo int
4. Questo ci permette di concludere che + : int → int → int
5. Dal contesto di uso deriviamo che x: int
6. In conclusione f(x: int) = 2 + x ha tipo int → int

L'ALGORITMO EFFETTIVO DI ML È PIÙ COMPLESSO

Storia più articolata

- L'algoritmo di inferenza di tipo è stato introdotto da Haskell Curry e Robert Feys per il lambda calcolo tipato semplice nel 1958
- Nel 1969, Roger Hindley ha esteso l'algoritmo dimostrando che restituisce il tipo più generale
- Nel 1978 Robin Milner introduce in modo indipendente un algoritmo, denominato W, per il linguaggio ML; l'algoritmo è in seguito dimostrato essere equivalente a quello proposto da Hindley
- Nel 1982 Luis Damas dimostra la completezza dell'algoritmo per ML

Inferenza di tipo e type checking

- Java, C, C++ e C# utilizzano un meccanismo di type checking
  - le annotazioni di tipo sono espliciti
- ML, OCaml, F# e Haskell utilizzano l'inferenza di tipo (ma lo usano anche C# 3.0 e Visual Basic .Net 9.0)
  - Il compilatore determina il tipo più generale (*the most general type)*

Statico vs dinamico

- JavaScript: controllo di tipo dinamico

```
js> var f= 3;
js> f(2);
typein:3: TypeError: f is not a function
js>
```

- ML: controllo di tipo statico

  f(x) f : A → B <fun> e x : A

Python


```
Python 2.7.10 (default, Oct 23 2015, 19:19:21)
[GCC 4.2.1 Compatible Apple LLVM 7.0.0 (clang-700.0.59.5)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> a = 3
>>> print(a)
3
>>> a = "aba"
>>> print(a)
aba
>>>
```

Controlli statici e dinamici

- ***Controllo dinamico***: la macchina astratta deve controllare che ogni operazione sia applicata a operandi del tipo corretto
  - overhead in esecuzione
- ***Controllo statico***: i controlli vengono effettuati dal compilatore prima della generazione del codice
  - efficienza dovuta all'analisi statica
  - prezzo da pagare: progettazione del linguaggio e compilazione più lenta

Ancora statico vs. dinamico

- Consideriamo il seguente frammento di programma (ML-like)

```
let x = 1 in
    if (0 = 1) then x = "errore"
    else x = 5
```

- **Il frammento non causa alcun errore per l'uso scorretto della variabile x**
- **il sistema di tipi di ML invece lo segnala come non corretto**

Decidibilità

- Esiste un metodo generale per stabilire se un programma determina un errore di tipo?

```
int X;
P; //invocazione della procedura P
X = “errore”;
```

- Se esistesse, lo potremmo applicare al nostro semplice programma…
- …e ciò implicherebbe poter decidere della “terminazione” di P, che NON è decidibile

Linguaggi e tipi

- **OCaml**: *strongly typed* e la maggior parte dei controlli è statica
- **Java**: *strongly typed* ma con controlli a run-time
- **C**: difficilmente fa controlli a run-time
- I linguaggi di scripting moderni (**Python, JavaScript**) hanno un controllo dinamico

Polimorfismo

- Idea di base è fare in modo che una operazione possa essere applicata a un insieme di tipi
- OCaml supporta il polimorfismo parametrico staticamente mediante un meccanismo per l'inferenza di tipo
- Polimorfismo di sottotipo: una variabile x di tipo T può essere usata in tutti quei contesti nei quali è previsto un tipo T' derivato da T
  - C++, Java, Eiffel, C#

Analisi

- Polimorfismo paramerico (ML)
  - uno stesso algoritmo (codice) può avere molti tipi (basta rimpiazzare le variabili di tipo)
    - ✓se f: t→t allora f: int→int, f: bool→bool, ...
- Polimorfismo da sottotipo
  - uno stesso simbolo può fare riferimento a algoritmi differenti
  - la scelta dell'algoritmo effettivo da eseguire è determinata dal contesto dei tipi
  - tipi associati ai nomi possono essere differenti
    - ✓+ ha tipo int*int→int, real*real→real

Esercizio

- Mettete assieme tutte le cose che abbiamo visto per creare un **sistema di valutazione di espressioni tipate** che
  - effettua il type checking, lanciando un'eccezione se l'espressione non è ben tipata
  - compila l'espressione in un codice intermedio senza variabili
  - esegue il codice intermedio ottenuto

  testando il codice prodotto su alcuni programmi
- Utilizzate liberamente il codice OCaml presentato a lezione (messo in linea sulla pagina del corso)

Cosa abbiamo ottenuto?

- La semantica operazionale (**eval**) è l'interprete del linguaggio
  - definito in modo ricorsivo
    - **`eval Prim("-", e1, e2) env ->`**
      `eval e1 env - eval e2 env`

    utilizzando la ricorsione di OCaml (ling. di implementazione)
- La semantica operazionale sul dominio dei tipi (**typ**) definisce il type checker del linguaggio
- È una vera implementazione? Certamente, ma…
  - **eliminando la ricorsione** dall'interprete ne otteniamo una versione più a basso livello, più vicina a una "vera" implementazione

Ricorsione

- La ricorsione può essere rimpiazzata con l'iterazione ma sono necessari degli stack per simulare il passo ricorsivo
  - a meno di definizioni ricorsive con una struttura molto semplice (tail recursion)
- NB. la struttura ricorsiva di **eval** ripropone quella del dominio sintattico delle espressioni (composizionalità)
- Il dominio delle espressioni non è tail recursive
  - Prim of string * exp * exp | ...
- Morale: per eliminare la ricorsione serve una rappresentazione esplicita degli stack

Macchine virtuali

- Ci sono fondamentalmente due modi principali per implementare una macchina virtuale
  - Stack VM (JVM, C# CLR, OCaml)
  - Register VM (C, LUA)
- La differenza tra i due approcci consiste nel meccanismo utilizzato per il trattamento del trasferimento dati (recupero e memorizzazione di operandi e risultati)

Stack machine

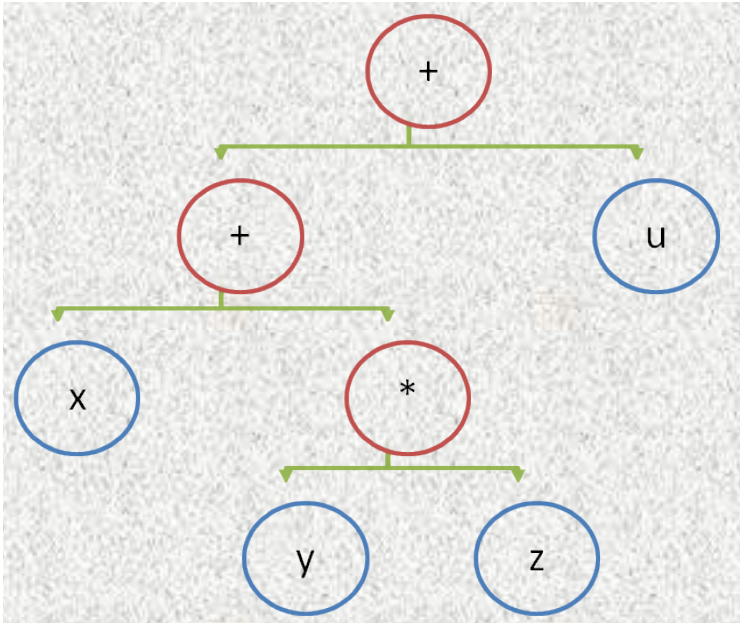
- Una macchina a stack implementa i registri con uno stack. Gli operandi dell'unità logica aritmetica (ALU) sono sempre i primi due registri dello stack e il risultato della ALU viene memorizzato nel registro in cima alla pila
- Il set di istruzioni utilizza la Notazione Polacca Inversa (Reverse Polish Notation) e le operazioni fanno riferimento solamente alla pila e non ai registri o alla memoria principale

Esempi di Stack VM

- UCSD Pascal p-machine (sostanzialmente la Burroughs stack machine del 1961)
- Java virtual machine
- VES (Virtual Execution System) del Common Intermediate Language (CIL) di Microsoft .NET
- Forth virtual machine
- Adobe Machine per PostScript
- Sun SwapDrop (linguaggio di progr. per Smartcard)
- Per i curiosi
  - **http://en.wikipedia.org/wiki/Stack_machine**

Esempio di valutazione

x + y * z + u



Notazione Polacca Inversa

x y z * + u +

codice compilato per una Stack Machine basata su NPI

push x
push y
push z
multiply
add
push u
add

Valutazione ricorsiva di espressioni

Consideriamo il linguaggio di espressioni intere

```
type expr =
   | CstI of int
   | Var of string
   | Prim of string * expr * expr
```

e l'interprete OCaml già visto

```
let rec eval e (env : (string * int) list) : int =
    match e with
    | CstI i              -> i
    | Var x               -> lookup env x
    | Prim("+", e1, e2) -> eval e1 env + eval e2 env
    | Prim("*", e1, e2) -> eval e1 env * eval e2 env
    | Prim("-", e1, e2) -> eval e1 env - eval e2 env
    | Prim _              -> failwith "unknown primitive";
```

Eliminare la ricorsione

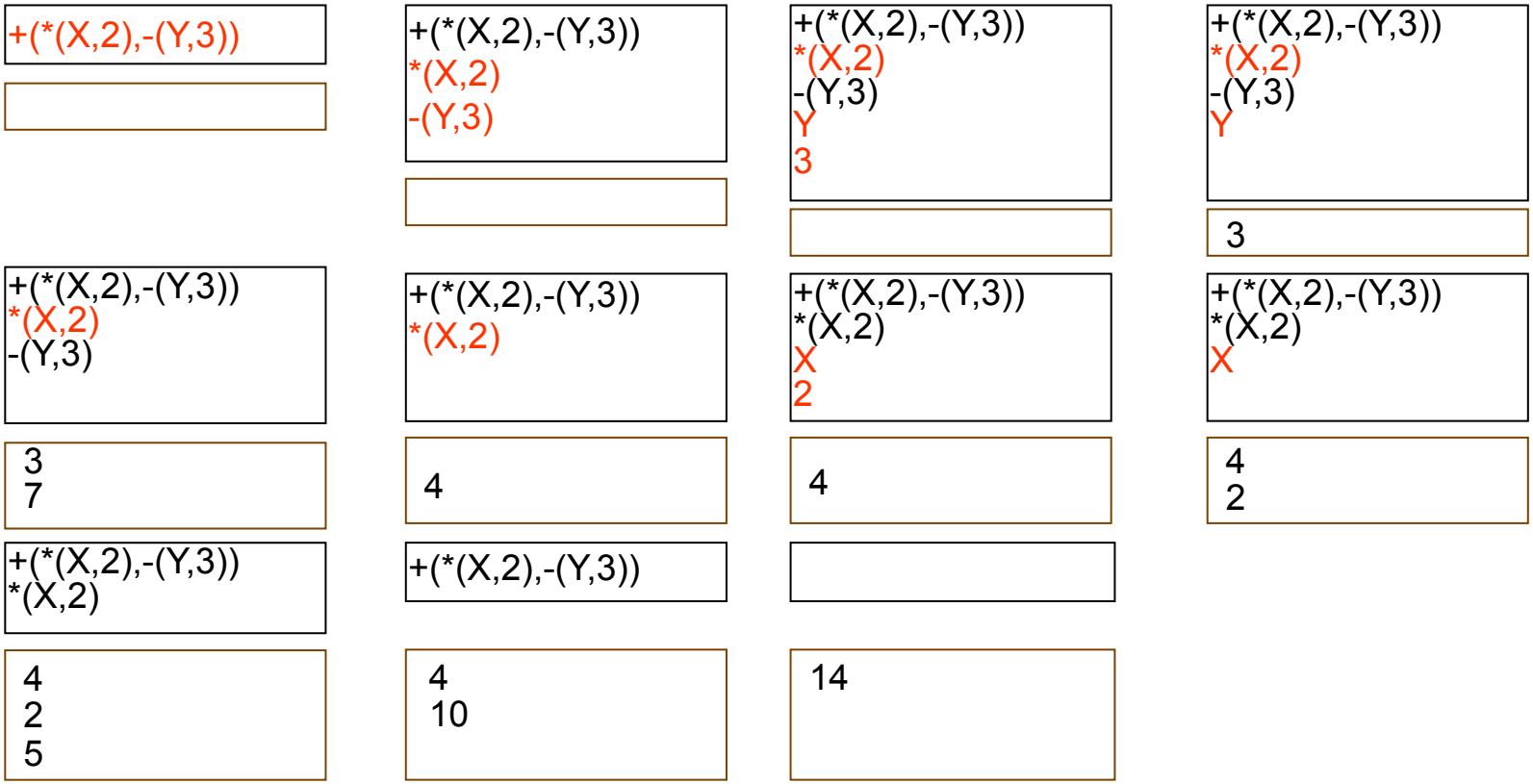
- La funzione **eval** ha due argomenti (espressione da valutare e ambiente) e calcola il risultato (un intero) ricorsivamente
- Obiettivo: definire una nuova funzione di valutazione per espressioni nella sintassi vista (non in NPI), **che non sia ricorsiva**, quindi utilizzando degli stack
- NB. l'ambiente non è modificato nelle chiamate ricorsive
- L'informazione da memorizzare in stack è costituita da
  - le sotto-espressioni da valutare
  - il valore calcolato per le sotto-espressioni

Eliminare la ricorsione: idea

- Usiamo una pila di espressioni “colorate” (il **continuation stack** **`contSt`**)
  - **espressione rossa**: deve essere valutata
  - **espressione blu**: è cominciata la valutazione delle sotto-espressioni, e se ne attendono i risultati per applicare l’operatore
- Una pila di di valori interi (**`tempSt`**)
  - contiene i risultati temporanei
- NB. sono le informazioni che inserivamo nello stack della Abstract Stack Machine di Java
- Vediamo l’algoritmo su un esempio

La valutazione di una espressione

`+(*(X,2),-(Y,3))` nell'ambiente **`{X -> 5, Y -> 7}`**



Definizioni ausiliarie: gli stack

Usiamo un semplice tipo polimorfo per gli stack: in una situazione reale avremmo usato un TdA, definito usando i moduli OCaml

```
type 'a stack = Empty | Push of 'a stack * 'a
   let emptystack = Empty
   let isempty p = (p = emptystack)
   let push p a = Push(p, a)
   let pop p = match p with
     | Push(p1, _) -> p1
     | Empty -> failwith "pop on empty stack"
   let top p = match p with
     | Push(_, a) -> a
     | Empty -> failwith "top on empty stack"
```

Espressioni colorate, ambienti e stack ausiliari

Elementi di `coloredExpr` sono espressioni intere colorate

```
type coloredExpr =
    | Blue of expr
    | Red of expr
```

Ambiente polimorfo con lookup

```
type 't env = (string * 't) list

let rec lookup ide (amb: 't env) = match amb with
    | (ide1, val1) :: amb1 ->
       if ide = ide1 then val1 else (lookup ide amb1)
    | [] -> failwith  (" "^ide^" not bound");;

(* Stack di coloredExpr, continuazioni *)
let contSt =  ref Empty       (* NB. variabili in OCaml! *)

(* Stack di interi, risultati temporanei *)
let tempSt =  ref Empty
```

L'interprete iterativo (1)

```
let evalIt ((e: expr), (rho: int env)) =
   contSt := push !contSt (Red(e));
   while not(isempty(!contSt)) do
      match top(!contSt) with
         | Red(x) ->
            (contSt :=  pop !contSt;
            match x with
               | CstI a -> tempSt := push !tempSt a;
               | Var str ->
            tempSt := push !tempSt (lookup str rho);
               | Prim (op, e1, e2) ->
                 (contSt := push !contSt (Blue(x));
                  contSt := push !contSt (Red(e1));
                  contSt := push !contSt (Red(e2)))
            )     (* continua *)
```

L'interprete iterativo (2)

```
        | Blue(x) ->
           (contSt :=  pop !contSt;
            match x with
            | Prim (op, _, _) ->
              let e1 = top !tempSt in
               tempSt := pop !tempSt;
               let e2 = top !tempSt in
                (tempSt := pop !tempSt;
                 match op with
                 |"+" -> tempSt := push !tempSt (e1 + e2)
                 |"-" -> tempSt := push !tempSt (e1 - e2)
                 |"*" -> tempSt := push !tempSt (e1 * e2)
                 |str -> failwith (" unknown op "^str)
                       )
            | _ -> failwith (" cerroneous blue expr ")
           )
  done;
let res = top !tempSt in tempSt := pop !tempSt; res ;;
```

Testando l'interprete iterativo


```
(* Esempio di valutazione: l'espressione dell'animazione *)
(* +( *(X,2),-(Y,3)) nell'ambiente {X -> 5, Y -> 7} *)
(* L'espressione *)
let myExpr = Prim("+", Prim("*", Var("X"), CstI(2)),
                  Prim("-", Var("Y"), CstI(3)));;
(* L'ambiente *)
let myEnv =  [("X",5);("Y",7)];;
(* La valutazione: ci aspettiamo 14 *)
evalIt(myExpr, myEnv);;

(* Test per vedere errori *)
(* Variable non bound *)
evalIt(Prim("*", Var("Z"), CstI(2)),[("X",5);("Y",7)])
(* Operatore non conosciuto *)
evalIt(Prim("/", Var("X"), CstI(2)),[("X",5);("Y",7)])
```

Stack Machine

- Abbiano realizzato una macchina virtuale basata sulla nozione di stack: la struttura di memoria in cui sono memorizzati gli operandi è una pila
- Le operazioni sono effettuate in tre passi
  - estrarre dati dalla pila
  - elaborare i dati
  - inserire i risultati sulla pila
- Sia JVM che MS-CLI sono stack machine (più articolate di quella che abbiamo visto…)

Parentesi

- Quello che abbiamo visto è la compilazione di un semplice linguaggio in una macchina virtuale simile alla SECD machine di Peter Landin e alla Functional Abstract Machine di Luca Cardelli per la realizzazione di ML
  - lettura interessante

    http://en.wikipedia.org/wiki/SECD_abstract_machine
- Il progetto e la realizzazione dei linguaggi di programmazione fanno emergere i fondamenti teorici dell'informatica