**OCaml: lo stile funzionale**
in Java (ma anche in C): l'effetto osservabile di un programma è la **modifica di uno stato**.
In OCaML il risultato della computazione è la **produzione di un nuovo valore**.

**Programmazione Funzionale = Value Oriented Programming**

- Un **programma OCaML** è una **espressione**
- Una **espressione OCaML** denota un **valore**

**Programma = Espressione = Valore**

L'esecuzione di un programma OCaML può essere vista come una sequenza di passi di calcolo (semplificazioni di espressione) che producono un valore.

**Espressioni**

- **Sintassi**: regole di buona formazione
- **Semantica**: regole di **type checking** (tipo o errore), regole di **esecuzione** che garantiscono che espressioni tipate producano un valore

**Valori**
Un **valore è un'espressione che non deve essere valutata ulteriormente**.

- 34 è **valore di tipo int**
- 34+17 è **espressione di tipo int** ma non è un valore

**Valore di espressione**
Notazione *<exp> => <val>* indica che l'espressione *<exp>* quando valutata calcola il valore *<val>*.

3 => 3
3+4 => 7
2*(4+5) => 18

**eval(e) = v** metanotazione per e => v

**Dichiarazioni**: let
Sintassi: `let x = e1 in e2`

- `x`: identificatore
- `e1, e2`: espressioni
- `let x = e1 in`: espressione
- `x = e1`: binding

**Espressioni**: let
Sia `e` l'espressione `let x = e1 in e2`
**Regola di valutazione**:

- `eval(e1) = v1`
- => sostituisco il **valore v1 in tutte le occorrenze di x in e2**, ottenendo e2'
  `subst(e2, x, v1) = e2'`
- `eval(e2') = v`
- `eval(e) = v`

$$\frac{eval(e1) = v1 \;\; subst(e2, x, v1) = e2' \;\; eval(e2') = v}{eval(let\ x = e1\ in\ e2) = v}$$

`let` *binding* = *scope*

es. `let x = 42 in` *scoping*
`x + (let y = "3110" in` *scoping*
`int_of_string y)`

**Overlapping**

`let x = 5 in ((let x = 6 in x) + x)`

Ho due casi possibili:

- `((let x = 6 in x) + 5)`
- `((let x = 6 in 5) + 5)` <-- è ciò che si verifica

**Alpha Conversione**

L'identità puntuale delle variabili legate non ha alcun senso. f(x) = x*x e f(y) = y*y sono la stessa cosa.
let x = 5 in ((let x = 6 in x) + x)
let x = 5 in ((let y = 6 in y) + x)

**Dichiarazione di funzioni**

```
let f (x : int) : int =
      let y = x * 10 in
            y * y;;
```

=> f(x) = (x*10)*(x*10);

**Applicazione**: `f 5;;`

- `: int 2500`

**Funzioni ricorsive**

```
let rec pow x y =
      if y = 0 then 1
      else x * pow x (y - 1);;
```

**Applicazione**: `pow 2 3;;`

- `: int = 8`

**La valutazione di una dichiarazione di una funzione è la funzione stessa**
=> le **funzioni sono valori**

**Applicazione di una funzione**

eval(e0 e1 ... en) = v' se

- eval(e0) = let f x1 ... xn = e
- eval(e1) = v1
- ...
- eval(en) = vn
- subst(e, x1, ..., xn, v1, ..., vn) = e' sostituisco in e i valori vi corrispondenti a xi
- eval(e') = v'

**High order function**: funzione da funzione ad altri valori

es. `let twice ((f : int -> int), (x : int)) : int =`
`f (f x);;`

Cioè prende una **coppia** (**non sequenza**, perché parametri messi tra parentesi e separati da virgole):

- funzione con un parametro in e ritorna int
- espressione int

e ritorna un int

**Liste**

let lst = [1; 2; 3];;

let empty = [];

let longer = 5::lst;;

let another = 5::1::2::3::[]

:: è l'operatore di append (o cons, dal LISP)

[] è la lista vuota

e1::e2 inserisce l'elemento e1 **in testa** alla lista e2

[e1; ...; en] equivalente alla lista e1::...::en::[]

**Strutturalmente** una lista può essere:

- [] la lista vuota
- la lista ottenuta da una operazione di cons di un elemento ad una lista

Per accedere agli elementi di una lista si usa il **pattern matching**. Es.:

```
let empty lst =
      match lst with
            [] -> true
            |h::t -> false;;
```

**Pattern matching**

```
match e with
      p1 -> e1
      | p2 -> e2
      | …
      | pn -> en;;
```

Le espressioni `pi` si chiamano **pattern**

[] matcha solamente il valore [] (lista vuota)

h::t matcha qualsiasi lista con almeno un elemento: lega il primo elemento della lista alla variabile h e il resto della lista alla variabile t

- a::[] matcha le liste con **esattamente un solo elemento**
- a::b::[] tutte le lista con **esattamente due elementi**
- a::b::c::d tutte le lista con **almeno tre elementi** (lista rimanente in d)

**Option type**

None simile a null, ma anche se una funzione torna None posso dire che tipo è

Some <identificatore> è equivalente a Not null?

**Definire tipi di dato in OCaML**

OCaML permette di definire nuovi tipi di dato

```
type giorno =
      Lunedi
      | Martedi
      | Mercoledi
      | Giovedi
      | Venerdi
      | Sabato
      | Domenica
```

Dichiarazione di tipo

Nome del tipo

Costruttori: definiscono i valori del tipo di dato

Es: Sabato ha tipo `giorno`, [Venerdi, Sabato, Domenica] ha tipo `giorno list`

Con il pattern matching ho un modo efficiente per accedere ai valori di un tipo di dato.

Es.:
```
let string_of_g (g : giorno) : string =
      match g with
            Lunedi -> "Lunedi"
            | Martedi -> "Martedi"
            | … ;;
```

Avrei potuto rappresentare i giorni come int, Lunedi = 1, Martedi = 2 ecc... ma gli int forniscono operazioni differenti da quelle significative su *giorno*, es.: Mercoledi - Domenica non ha senso. In più ci sono molti più valori int che di valori giorno.
-> i linguaggi moderni forniscono strumenti per definire tipi di dato.

I costruttori possono portare valori:
```
type foo =
      Nothing
      | Int of int
      | Pair of int * int
      | String of string;;
```
Avrà valori possibili
```
Nothing, Int 3, Pair (4, 5), String "hello"…
```
E posso matchare così
```
      let get_count (f : foo) : int =
            match f with
                  Nothing -> 0
                  | Int(n) -> n
                  | Pair(n, m) -> n + m
                  | String(s) -> 0;;
```

Possono anche essere ricorsivi
```
type tree =
      Leaf of int
      | Node of tree * int * tree;;
let t1 = Leaf 3;;
let t2 = Node(Leaf 3, 4, Leaf 4);;
let t3 = Node(Leaf 3, 2, Node(Leaf 5, 4, Leaf 6));;
```

**Albero binario**
```
type btree =
      Empty
      | Node of btree * int * btree;;
let t : tree = Node(Node(Empty, 1, Empty), 3, Node(Empty, 2, Node(Empty, 4,
Empty));;
```

**Ricerca**
```
let rec contains (t : tree) (n : int) : bool =
      match t with
            Empty -> false
            | Node(lt, x, rt) -> x = n || (contains lt n) || (contains rt
n);;
```

**Inserimento in ABR**

```
let rec insert (t : tree) (n : int) : tree =
      match t with
            Empty -> Node(Empty, n, Empty)        //se l'albero è vuoto…
            | Node(lt, x, rt) ->
                  if x = n then t else
                        if n < x then Node(insert lt n, x, rt)
                        else Node(lt, x, insert rt n);;
```

**Generici**

`'a list` indica una generica lista (int list, string list, ...)

**Moduli e interfacce**

```
module type Set = sig
      type 'a set
      val empty : 'a set
      val add : 'a -> 'a set -> 'a set
      val remove : 'a -> 'a set -> 'a set
      val list_to_set : 'a list -> 'a set
      val member : 'a -> 'a set -> bool
      val elements : 'a set -> 'a list
end
```

In un file .mli:

`sig...end` racchiude la segnatura, che definisce l'interfaccia

`val :` valori che devono essere definiti e dei loro tipi

Stessa idea delle interfacce Java: fornire funzionalità nascondendo implementazione.

Implementazione

```
module MySet : Set = struct
      …
end
```

Nome del modulo

Signature che deve essere implementata

Implementazione delle operazioni

Per usare, due modi:

- Dot notation (tipo Java)

  ```
  let s1 = MySet.add 3 MySet.empty;;
  ```

- Open module

  ```
  open MySet;;
  let s1 = add 3 empty;;
  ```

**Compilare programmi OCaML**

file.ml **Codice sorgente -> ocamlc ->** file.cmo (**bytecode**), a.out (**eseguibile**)

**Eseguire**

a.out -> **ocamlrun** -> **result**

**Elementi di semantica operazionale**

Un **linguaggio di programmazione** possiede:

- **sintassi**, definisce le **formule ben formate** del linguaggio, cioè **programmi sintatticamente corretti**, tipicamente generati da una grammatica
- **semantica**, fornisce **l'interpretazione dei token** in termini di entità matematiche note e **dà un significato ai programmi sintatticamente corretti**

La teoria dei linguaggi formali fornisce fomalismi di specifica (grammatiche) e tecniche di analisi (automi) per trattare aspetti sintattici. Per la semantica esistono vari approcci: denotazionale, operazionale, assiomatica... e la semantica formale viene di solito definita su una rappresentazione dei programmi in sintassi astratta.

**Sintassi concreta**

Solitamente definita da **grammatiche libere da contesto**.

Es: grammatica di semplici espressioni logiche

```
e ::= v | Not e | (e And e) | (e Or e) | (e Implies e)
v ::= True | False
```

Comoda per i programmatori (operatori infissi, associatività/commutatività, precedenze...), ma meno comoda per computazione poiché **si presta a problemi di ambiguità**.

**Sintassi astratta**

L'**albero sintattico** (***abstract syntax tree***) di una espressione `exp` mostra (risolvendo ambiguità) come `exp` possa essere generata dalla grammatica.

La **sintassi astratta** è una rappresentazione lineare dell'albero sintattico:

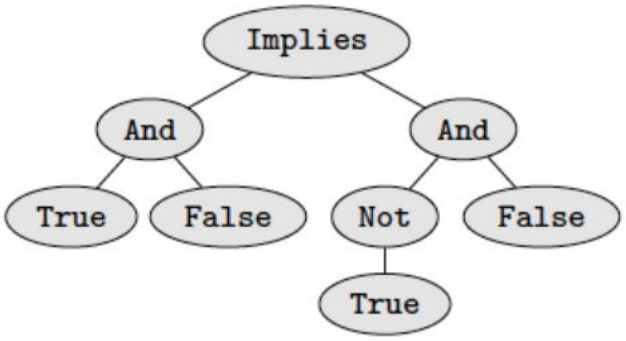
- operatori: nodi
- operandi: sottoalber

Per gli AST abbiamo sia notazione lineare che rappresentazione grafica.

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

che diventa

**Sintassi astratta** Implies(And(True, False), And(Not(True), False))




**Semantica dei linguaggi**

Tre metodi per definire semantica

- **semantica operazionale**: descrive significato di un programma in termini dei cambiamenti di stato di una macchina astratta
- **semantica denotazionale**: significato programma è funzione matematica definita su opportuni domini
- **semantica assiomatica**: descrive significato in base alle proprietà soddisfatte prima e dopo la sua esecuzione

Ognuno dei tre metodi ha i suoi vantaggi e svantaggi rispetto a **aspetti matematici**, **facilità di dimostrazione** o **utilità per definire un interprete o compilatore** per il linguaggio.

La **semantica operazionale** per un linguaggio **L** definisce **in modo formale e con strumenti matematici** una macchina astratta **M(L)** in grado di eseguire programmi scritti in **L**.
Un **sistema di transizioni** è costituito da

- **Config** insieme di configurazioni (**stati**)
- **Relazione di transizione** ↠ sottoinsieme di Config x Config

c ↠ d significa che c e d sono nella relazione ↠
=> c ↠ d se lo stato c **evolve** nello stato d

**Small step**
Nella semantica operazionale small step la relazione di transizione ↠ descrive un passo del processo di calcolo. Cioè abbiamo e ↠ d se partendo dall'espressione/programma e l'esecuzione di **un passo di calcolo** ci porta nell'espressione d.
Una valutazione completa di e avrà forma

e ↠ e1 ↠ ... ↠ en, con en che può rappresentare il valore finale

La valutazione di un programma procede attraverso le configurazioni intermedie che può assumere il programma.
Utile per descrivere comunicazioni fra thread e proprietà come la deadlock freedom in programmi concorrenti.

**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 e ⇓ v.
**Una valutazione completa di un'espressione è ottenuta componendo le valutazioni complete delle sue sotto-espressioni**.
Useremo questa.

**Valori**
true => true
false => false

**Not**
e => v
not e => !v

**And**
e1 => v1  e2 => v2
e1 and e2 => v1 && v2

Le regole di valutazione costituiscono un **proof system**
premessa1 ... premessak

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 definiscono il processo di calcolo**.

**Interprete Espressioni logiche**

1. Sintassi astratta

```
type boolexp =
      True
      | False
      | Not of boolexp
      | And of boolexp * boolexp
;;
```

2. Dalle regole all'interprete

Vogliamo definire una funzione **eval** tale che **eval(e) = v** se e solo se **e => v**

Es. dalla regola

$$\frac{e \Rightarrow v}{\text{not } e \Rightarrow !v}$$

otteniamo

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

```
let rec eval exp =
      match exp 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**

```
type exp =
      Cnst of constant
      | Var of variabile
      | Op of exp * operand * exp
;;
```

Per definire l'interprete introduciamo **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 numero**.

- Il nome è solamente usato per reperire il valore, è un tag
- esempio

```
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 è 18.

**Ambiente**

Un ambiente è una collezione di binding
env = { x -> 25, y -> 6}

- env contiene due binding
  - binding tra identificatore x e valore 25
  - binding tra identificatore y e valore 6
  - l'identificatore z non è legato nell'ambiente

Astrattamente, un ambiente è una **funzione di tipo Ide -> Value + Unbound**, con la costante Unbound che permette di rendere la funzione totale, nel caso di env z è bindato a Unbound.

- env(x) denota il valore v associato ad x nell'ambiente env, 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

Quindi dato env = {x -> 25, y -> 7}, allora env[5/x] = {x -> 5, y -> 7}.
In pratica con env[v/x] "bindo" il valore v all'identificatore x.

**Lista di coppie**

```
let emptyenv = [];;

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**

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

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

env -> e1 => v1  env -> e2 => v2
env -> Op(e1, Plus, e2) => v1 + v2

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

**Interprete completo per semplici espressioni intere**

```
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(e2, Div, e2) -> eval e1 env \ eval e2 env
;;
```

**Aggiunta delle dichiarazioni**

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

type exp =
      …
      | Let of variable * exp * exp
;;
```

Ad es.: `Let("z", Cnst 17, Op(Var "z", Plus, Var "z"));;`

env -> erhs => xval  env[xval/x] -> ebody => v

env -> Let x = erhs in ebody => v

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

```
eval (Let(x, erhs, ebody)) env ->
      let xval = eval erhs env in          //env -> erhs => xval
            let env1 = (x, xval)::env in   //env[xval/x]
                  eval ebody env1          //env[xval/x] -> ebody => v
;;
```

**Interprete completo con dichiarazioni**

```
let rec eval (e : exp) (env : (variable * int) list) : int =
      match e with
      Cnst i -> i
      | Var x -> lookup x env
      | 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(e2, 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, altrimenti è libera.
Es. ∀x (P(x) and Q(y))
x è legata, y è libera]

**Occorrenze libere**
La nozione di variabile libera o legata si applica anche a let, che si comporta come quantificatore per la variabile che introduce.
Un **identificatore** si dice **legato se compare nell'ebody**, altrimenti è libero.
Es.:

- `let x = erhs in ebody;;` //x legata se compare in ebody
- `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 permette di valutare espressioni costruite con la sintassi indicata ovviamente.
Il valore di una espressione chiusa non dipende da env

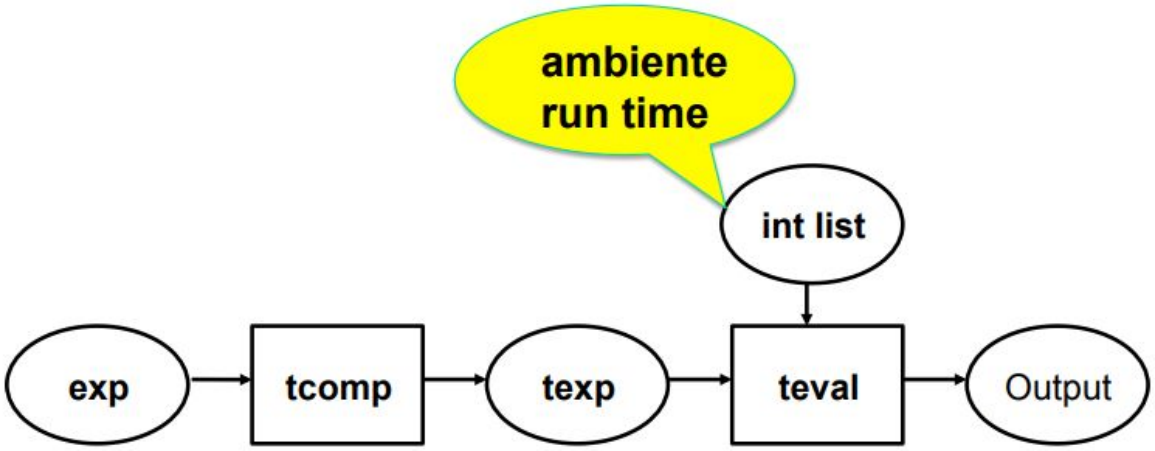
**Verso la compilazione**
Il nostro interprete **ogni volta** che deve **determinare il valore associato ad una variabile** fa un'operazione di **lookup nell'ambiente**: questo costa.
Possiamo ottimizzare esecuzione introducendo un piccolo compilatore che traduce tutte le occorrenze di identificatori in "**indici di accesso**", in modo che l'operazione di lookup venga eseguita senza effettuare una ricerca log(n) sull'ambiente ma accedendo direttamente a dov'è "memorizzato" l'identificatore.

```
Let("z", Cnst 17, Op(Var "z", Plus, Var "z"));;
```
Diventa
```
Let(Cnst 17, Op(Var 0, Plus, Var 0));;
```
Il valore 0 indica il binding (let) più vicino

**Indici per variabili**
**Indice** di una variabile = **numero di 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)));;
```
https://en.wikipedia.org/wiki/De_Bruijn_index

**Linguaggio intermedio**
Target Expression
```
type texp =
      TCnst of constant
      | TVar of int
      | TLet of texp * texp
      | TOp of texp * operando * texp
;;
```

**Compilazione in codice intermedio**




http://pages.di.unipi.it/levi/PR2-18-17.pdf 44

**Codice intermedio**
Rappresentare il programma sorgente in codice intermedio permette di dominare la complessità dell'implementazione di un linguaggio di programmazione. La rappresentazione in codice intermedio permette di effettuare ottimizzazioni sul codice (come nel nostro caso l'eliminazione dei nomi a run-time)
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.
Fred Chow, Intermediate Representation, Communications of ACM 56 (12) 2013

**Tipi di dato, a cosa servono?**

- **Livello di progetto**: organizzano l'**informazione**
  - Tipi diversi per concetti diversi
  - Meccanismi espliciti dei linguaggi per l'astrazione sui dati
- **Livello di programma**: identificano e prevengono **errori**
  - I tipi sono controllati automaticamente e costituiscono un **controllo dimensionale** (es. 3+"pippo" **deve** essere sbagliata)
- **Livello di implementazione**: permettono alcune **ottimizzazioni**
  - es.: `bool` richiede meno bit di `int`
  - Strumenti per fornire informazioni necessarie alla macchina astratta per allocare spazio di memoria

**Dati: classificazione**

- **Denotabili**: se possono essere **associati ad un nome**
  es: `let plusone = (fun x -> x+1);;`
- **Esprimibili**: se possono essere il **risultato di una espressione complessa** (diversa dal semplice nome)
  es: `let pick_one = if n = 0 then fun x -> x + 1 else fun x -> x - 1;;`
- **Memorizzabili**: se possono essere **memorizzati in una variabile**
  es: `Obj.val = Obj.val + 10;`

**In ML**

- **Denotabili**:
  `let plus (x, y) = x + y;;`
- **Esprimibili**:
  `let plus = fun x -> fun y -> x + y`
- **Memorizzabili**: no

In una macchina astratta (e in una semantica) si possono vedere due classi di tipi di dato (o **domini semantici**):

- **Tipi di dato di sistema**
  Definiscono lo **stato** e le **strutture dati utilizzate** nella simulazione di costrutti di controllo
- **Tipi di dato di programma**
  Domini corrispondenti ai **tipi primitivi** del linguaggio e ai **tipi che l'utente può definire** (se il linguaggio lo consente)

Un **tipo di dato** (TD) è una **collezione di valori** rappresentati da opportune strutture dati e da un insieme di operazioni per manipolarli. Come sempre, semantica e implementazione sono l'oggetto di studio.

**Descrittori di dato**

Obiettivo: **rappresentare una collezione di valori utilizzando quanto ci viene fornito da un linguaggio macchina**. Qualunque valore della collezione è alla fine una stringa di bit.
Problema: per poter riconoscere il valore e interpretare correttamente la stringa di bit è necessario **associare** alla stringa un'altra **struttura** che contiene la descrizione del tipo (**descrittore di dato**), che viene usata ogni qualvolta si applica al dato un'operazione per **controllare che il tipo di dato sia quello previsto dall'operazione** (**type-checking dinamico**) e per **selezionare l'operatore giusto** in caso di operazioni overloaded.

```
type exp =
      Eint of int
      | Ebool of bool
;;

type val =                  //valori di run-time
      Int of int
      | Bool of bool
;;
```

Per usare i descrittori

```
let typecheck (x, y) = match x with
      "int" -> (match y with
            Int(u) -> true
            | _ -> false)
;;

let plus (x, y) =
      if typecheck("int", x) & typecheck("int", y) then
            (match (x, y) with
                  (Int(u), Int(w)) -> Int (u+x))
      else failwith ("Type error")
;;
```

Casistiche

1. **Informazione sui tipi conosciuta completamente a tempo di compilazione (OCaML)**
   Si possono eliminare i descrittori di dato ed il type checking è effettuato completamente dal compilatore (type checking statico)
2. **Informazione sui tipi nota solamente a tempo di esecuzione (JavaScript)**
   Sono necessari descrittori per tutti i tipi di dato ed il type checking è effettuato completamente a tempo di esecuzione (type checking dinamico
3. **Informazione sui tipi conosciuta solo parzialmente durante la compilazione (Java)**
   I descrittori contengono solo l'informazione dinamica ed il type checking è effettuato in parte dal compilatore e in parte a runtime

**Esempi di tipi scalari**

- Booleani
  - val: true, false
  - op: or, and, not, condizionali
  - repr: 1 bit
  - note: C non ha un tipo bool
- Caratteri
  - val: a, A, è, ., ,, “, ...
  - op: uguaglianza, code/decode, dipendono dal linguaggio
  - repr: 1 byte (ASCII) o 2 byte (UNICODE)
- Interi
  - val: 0, 1, -3, maxint, ...
  - op: +, -, *, div, mod, ...
  - repr: alcuni byte (2 o 4), complemento a 2
  - note: interi e interi lunghi (anche 8 byte), limitati problemi nella portabilità quando la lunghezza non è specificata nella definizione del linguaggio

- Reali
    - val: valori razionali di un certo intervallo
    - op: +, -, *, /, ...
    - repr: alcuni byte (4), virgola mobile
    - note: reali e reali lunghi (8 byte), problemi di portabilità quando la lunghezza non è speficata nella definizione del linguaggio
- void
    - ha un solo valore
    - non ha operazioni
    - serve per definire il tipo di operazioni che modificano lo stato senza restituire alcun valore

    `void f (...) {...}`

    Il valore restituito da f di tipo `void` è sempre il solito e dunque non interessante.

**Tipi composti**

- Record: collezione di campi (field), ciascuno di un tipo (anche diverso) e ogni campo è selezionabile dal suo nome
- Record varianti: record dove solo alcuni campi (mutualmente esclusivi) sono attivi ad un dato istante
- Array: funzioni da un tipo indice (scalare) ad un altro tipo. Gli array di caratteri sono chiamati stringhe e hanno operazioni speciali
- Insieme: sotto-insieme di un tipo base
- Puntatore: reference ad un oggetto di un altro tipo

**Record**

Introdotti per manipolare in modo unitario dati di **tipo omogeneo**: C, C++, Lisp, Ada, Pascal... Java non ha tipi record perché sono sostituiti dalle classi.

In C:

```
struct studente {
      char nome[20];
      int matricola;
}
studente s;
s.matricola = 530257;
```

I record possono essere annidati. Sono tipo **memorizzabili, esprimibili e denotabili**.

Implementazione in OCaML

```
type label = Lab of string;;
type exp = …
      | Record of (label * exp) list
      | Select of exp * label
;;
Record[(Lab “size”, Int 7); (Lab “weight”, Int 255)];;

let rec lookupRecord body (Lab l) = match body with
      [] -> raise FieldNotFound
      | (Lab l’, v)::t -> if l = l’ then v else lookupRecord t (Lab l)
;;
```

```
let rec eval e = match e with
      …
      | Record (body) -> Record(evalRecord body)
      | Select (e, l) -> match eval e with
            Record(body) -> lookupRecord body l
            | _ -> raise TypeMismatch
;;

evalRecord body = match body with
      [] -> []
      | (Lab l, e)::t -> (Lab l, eval e)::evalRecord t
;;
```

**Array**

Collezione di dati omogenei: funzione da indice a tipo degli elementi.
Gli indici appartengono ad un insieme discreto e l'elemento può essere di qualsiasi tipo (anche se raramente è un tipo funzionale).
Principale operazione permessa: selezione di un elemento.
**Attenzione**: la modifica non è un'operazione sull'array ma sulla locazione modificabile che memorizza un elemento dell'array.

**Implementazione Array**

Elementi memorizzati in locazioni contigue, due modi:

- ordine di **riga**:
  V[1, 1]; V[1, 2]; ...; V[2, 1]; ...
- ordine di **colonna**:
  V[1, 1]; V[2. 1]; ...; V[1, 2]; ...

**Formula di accesso** (caso di array unidimensionale): vettore **V[N]** di **elem_type**, **V[n] = base + c*n** con c la dimensione necessaria a memorizzare un elem_type; Si può determinare una formula di accesso più articolata anche per gli array multidimensionali.

**Puntatori**

Valori: riferimenti, **null** (nil).
Operazioni:

- creazione
  Funzione di libreria che alloca e restituisce un puntatore (es. malloc)
- dereferenziazione
  Accesso al dato "puntato": *p
- uguaglianza
  In speciale test di uguaglianza con **null**

**Tipi di dato di sistema: pila non modificabile**

**Interfaccia**

```
module type pila = sig
      type 'a pila
      val create : int -> 'a pila
      val push : 'a * 'a pila -> 'a pila
      val pop : 'a pila -> 'a pila
      val top : 'a pila -> 'a
      val is_empty : 'a pila -> bool
      val lungh : 'a pila -> int
      exception Empty
      exception Full
end
```

**Semantica algebrica**

```
module sempila : pila = struct
      type 'a pila = New of int | Push of 'a pila * 'a
      exception Empty
      exception Full
      let create n = New n
      let rec max = function
            | New n -> n
            | Push(p, a) -> max p
      let rec lungh = function
            | New _ -> 0
            | Push(p, _) -> 1 + lungh p
      let push(a, p) = if lungh (p) = max(p)
            then raise Full
            else Push(p, a)
      let pop = function
            | Push(p, a) -> p
            | New n -> raise Empty
      let top = function
            | Push(p, a) -> a
            | New n -> raise Empty
      let is_empty = function
            | Push(p, a) -> false
            | New n -> true
end
```

**Implementazione**

Il componente principale dell'implementazione è un array: astrazione della memoria fisica in una implementazione in linguaggio macchina. Classica implementazione sequenziale usata anche per tipi di dato simili alle pila come le code

```
module imppila : pila = struct
      type 'a pila = IPila of ('a option array) * int
      let create n = IPila (Array.make n None, -1)
      let push(x, IPila(s,n)) =
            if n = (Array.length s - 1) then raise Full
            else (Array.set s (n + 1) (Some x) ; IPila (s, n + 1))
      let top(IPila(s,n)) =
            if n = -1 then raise Empty
            else (match Array.get s n with | Some y -> y)
      let pop(IPila(s,n)) =
            if n = -1 then raise Empty
            else IPila (s, n -1)
      let is_empty(IPila(s,n)) =
            if n = -1 then true
            else false
      let lungh(IPila(s,n)) = n
end
```

**Programmi come dati**

Caratteristica base della macchina di Von Neumann: **i programmi sono un particolare tipo di dato rappresentato nella memoria della macchina**. In teoria permette a qualsiasi programma di operare su di essi, oltre all'interprete.

Possibile sempre in **linguaggio macchina**, mentre nei linguaggi ad alto livello è possibile solo se:

- la rappresentazione dei programmi è visibile nel linguaggio
- il linguaggio fornisce operazioni per manipolare tale rappresentazione

**Meta-Programmazione**

Un meta-programma è un programma che opera su altri programmi: interpreti, analizzatori, debugger, ottimizzatori, compilatori...

Utile soprattutto per definire il linguaggio stesso.

**Definizione di tipi di dato**

Gran parte della programmazione di applicazioni consiste nella definizione di nuovi "tipi di dato". Un qualunque tipo di dato può essere definito in qualsiasi linguaggio, ma:

- Quanto costa?
  Dipende dalle strutture primitive fornite dal linguaggio.
  Molto utile disporre di strutture statiche sequenziali ma anche di meccanismi per creare strutture dinamiche
- Esiste il tipo definito?
  Se i tipi non sono denotabili, non abbiamo veramente a disposizione il tipo e quindi non è possibile "dichiarare" oggetti di tipo Lista (ad es.).
  In Pascal, ML, Java... i tipi sono denotabili anche se con meccanismi diversi:
  - Dichiarazioni di tipo
  - Dichiarazioni di classe
    Meccanismo di C++, Java e OCaML: **il tipo è la classe**, parametrico e con relazioni di sottotipo. I valori del nuovo tipo sono oggetti, creati con un'operazione di **istanziazione della classe** e **non con una dichiarazione**.
    La parte struttura dati degli oggetti è costituita da un **insieme di variabili d'istanza allocate sullo heap**
- Il tipo è astratto?
  Un tipo astratto è un insieme di valori di cui **non si conosce la rappresentazione** (implementazione) e che **possono essere manipolati solo con le operazioni associate**.
  Sono tipi astratti tutti i tipi primitivi forniti dal linguaggio poiché la loro rappresentazione effettiva non ci è nota, e non è comunque accessibile se non con le operazioni primitive.
  Per realizzare tipi di dato astratti servono:
  - **meccanismo che permetta di dare un nome al nuovo tipo** (dichiarazione di tipo o di classe)
  - **meccanismo di protezione** o *information hiding* **che renda la rappresentazione visibile soltanto alle operazioni primitive** come attributi privati di una classe o moduli e interfacce in C e OCaML

**Nomi**
Un nome in un linguaggio è... un nome. La maggior parte sono definiti dal programma (**identificatori**) ma ci sono anche simboli speciali (come +, super, sqrt...).
**Un nome è una sequenza di caratteri usata per denotare simbolicamente un oggetto**.

L'uso dei nomi è un meccanismo elementare di **astrazione**.
**Astrazione sui dati** perché la dichiarazione di una variabile permette di introdurre un **nome simbolico** per una **zona di memoria**, astraendo sui dettagli della gestione della stessa.
**Astrazione sul controllo** perché dichiarare il nome di una funzione/procedura è l'aspetto essenziale nel processo di **astrazione procedurale**.

**Entità denotabili**
Elementi di un linguaggio di programmazione a cui posso assegnare un nome.
- Entità i cui nomi sono **definiti dal linguaggio** (tipi e operatori primitivi, ...)
- Entità i cui nomi sono **definibili dall'utente** (variabili, parametri, procedure, tipi, costanti, classi, oggetti, ...)

**Binding e scope**
**Binding**: associazione tra **nome** e **oggetto denotato**
**Scope**: definisce quella **parte di programma nel quale è attivo** un determinato **binding**
**Binding time**: momento nel quale vengono prese le decisioni relative alla gestione delle associazioni
- **Language design time**: binding delle operazioni primitive, tipi primitivi...
- **Program writing time**: binding di sotto-programmi, classi...
- **Compile time**: associazioni per le variabili globali
- **Runtime**: associazioni tra variabili e locazioni, associazioni per tutte le entità dinamiche

**Binding statico e dinamico**
**Static binding**: associazione attivata **prima di mandare** il programma in esecuzione
**Dynamic binding**: associazione attivata **dopo aver mandato** il programma in esecuzione
Molte delle caratteristiche dei linguaggi di programmazione dipendono dalla scelta del binding statico o dinamico.
I linguaggi **compilati cercano di risolvere il binding staticamente**
I linguaggi **interpretati devono risolvere il binding dinamicamente**

**Ambiente**
Nella macchina astratta del linguaggio, **per ogni nome e** per ogni **sezione del programma l'ambiente determina l'associazione corretta**. L'ambiente è definito come l'**insieme delle associazioni nome-oggetto** esistenti a runtime in uno specifico punto del programma e in uno specifico momento dell'esecuzione.
Le **dichiarazioni** sono il **costrutto sintattico** che permette di **introdurre associazioni nell'ambiente**.

```
int x; //Dichiarazione di una variabile

int f() { return 0; } //Dichiarazione di una funzione

type BoolExp = //Dichiarazione di un tipo
      True
      | False
      | Not of BoolExp
      | And of BoolExp * BoolExp
;;
```

**Aliasing**: nomi diversi per lo stesso oggetto.

Es:
```
A a1 = new A();
A a2 = new A();
…
a1 = a2;
```

**Blocco**: regione testuale del programma che può contenere dichiarazioni

- C, Java: { ... }
- OCaML: let ... in

**Blocco associato a procedura**: corpo della procedura con le dichiarazioni dei parametri formali

**Blocco in-line**: meccanismo per raggruppare comandi

**Politica di accesso dei blocchi: LIFO**

I cambiamenti dell'ambiente avvengono all'entrata e all'uscita dai blocchi (anche annidati)

**Tipi di ambiente**

- **Ambiente locale**: l'insieme delle **associazioni dichiarate localmente**, **comprese le eventuali associazioni relative ai parametri**
- **Ambiente non locale**: **associazioni** dei nomi che sono **visibili all'interno del blocco ma non dichiarati nel blocco stesso**
- **Ambiente globale**: **associazioni** per i nomi **usabili da tutte le componenti che costituiscono il programma**

**Cambiamenti dell'Ambiente**

L'ambiente può cambiare a runtime ma ciò avviene in precisi momenti:

- **Entrando in un blocco**: creazione associazioni fra nomi locali al blocco e oggetti denotati, disattivazione delle associazioni per i nomi ridefiniti
- **Uscendo da un blocco**: distruzione associazioni fra nomi locali al blocco e oggetti denotati, riattivazione associazioni per nomi che erano stati ridefiniti

Operazioni possibili:

- **Naming** creazione associazione fra nome e oggetto (dichiarazione locale al blocco o parametro)
- **Referencing** riferimento a oggetto tramite il suo nome
- **Disattivazione** di associazione fra nome e oggetto (la nuova associazione per quel nome maschera la vecchia, che rimane disattivata fino all'uscita dal blocco)
- **Riattivazione** di associazione fra nome e oggetto denotato (una vecchia associazione che era stata disattivata è riattivata all'uscita dal blocco)
- **Unnaming** distruzione associazione fra noem e oggetto (es. variabile locale all'uscita da un blocco)

**NB: tempo vita oggetti non necessariamente uguale a tempo vita associazione**

**Implementazione ambiente**

**Env**

Tipo (polimorfo) utilizzato nella semantica e nelle implementazioni per mantenere il binding tra nomi e valori di un opportuno tipo.

La specifica definisce il tipo come funzione e l'implementazione che vedremo usa le liste.

```
module type Env = sig type 't env
      val emptyenv : 't -> 't env
      val bind : 't env * string * 't -> 't env
      val bindlist : 't env * (string list) * ('t list) -> 't env
      val applyenv : 't env * string -> 't
      exception WrongBindList
end
```

**Specifica**

```
module FunEnv : Env = struct
      type 't env = string -> 't
      exception WrongBindList
      let emptyenv (x) = function (y: string) -> x
            //x è valore di default
      let applyenv(x, y) = x y
      let bind (r, l, e) = function lu -> if lu = l then e else applyenv(r,
lu)
      let rec bindlist (r, il, el) = match (il, el) with
            ([], []) -> r
            | (i::il1, e::el1 -> bindlist (bind (r, i, e), il1, el1)
            | _ -> raise WrongBindlist
end
```

**Implementazione**

```
module ListEnv : Env = struct
      type 't env = (string * 't) list
      let emptyenv (x) = [("”), x]
      let rec applyenv (x, y) = match x with
            [(_, e)] -> e
            | (i1, e1)::x1 -> if y = i1 then e1 else applyenv(x1, y)
            | [] -> failwith("Wrong env”)
      let bind (r, l, e) = (l, e)::r
      let rec bindlist (r, il, el) = match (il, el) with
            ([], []) -> r
            | (i::il1, e::el1) -> bindlist (bind(r, i, e), il1, el1)
            | _ -> raise WrongBindList
end
```

**Regole di scope**

Lo scope di un binding definisce quella parte del programma nella quale il binding è attivo

- **scope statico o lessicale**: determinato dalla struttura sintattica del programma
- **scope dinamico**: determinato dalla struttura a tempo di esecuzione

Differiscono solo per l'**ambiente non locale**.

**Regole di visibilità**

Una dichiarazione locale in un blocco è visibile in quel blocco e nei blocchi in esso annidati (salvo ridichiarazioni con stesso nome, shadowing)

La regola di visibilità (cioè l'annidamento) è basata

- Sul testo del programma (scope statico)
- Sul flusso d'esecuzione (scope dinamico)

**Scope statico**

Dichiarazioni locali in un blocco includono solo quelle presenti nel blocco (e non quelle nei blocchi annidati).

Se si usa un nome in un blocco, l'associazione valida è quella locale al blocco. Se non esiste, si prende la prima associazione valida a partire dal blocco immediatamente esterno, dal più vicino al più lontano. Se non esiste, si considera l'ambiente predefinito del linguaggio. Se non esiste, errore.

Se il blocco ha un nome, fa parte dell'ambiente locale del blocco immediatamente esterno.

Permette di determinare tutti gli ambienti di un programma staticamente osservando la struttura sintattica del programma.

Gestione a runtime articolata: gli ambienti non locali di funzioni e procedure evolvono diversamente dal flusso di attivazione e disattivazione dei blocchi (LIFO).

**Scope dinamico**

L'associazione valida per x in un punto P del programma è la più recente associazione creata (in senso temporale) per x, che sia ancora attiva quando il flusso d'esecuzione arriva a P.

Gestione a runtime semplice

- vantaggi: flessibilità programmi
- svantaggi: difficile comprensione delle chiamate delle procedure e controllo statico tipi non possibile