Tipi di Dato

- descrittori, tipi, controllo e inferenza dei tipi
- specifica (semantica) e implementazione di tipi di dato
  - implementazioni “sequenziali”
    - pile non modificabili
  - implementazione delle liste con la heap
  - termini con pattern matching e unificazione
  - ambiente, tabelle
  - tipi modificabili
    - pile modificabili
    - S-espressioni
  - programmi come dati e metaprogrammazione
- meccanismi per la definizione di nuovi tipi
- astrazioni sui dati

Tipi di Dato di Sistema e di Programma

- in una macchina astratta (e in una semantica) si possono vedere due classi di tipi di dato (o domini semantici)
  - i tipi di dato di sistema
    - domini semantici che definiscono lo stato e strutture dati utilizzate nella simulazione di costrutti di controllo
  - i tipi di dato di programma
    - domini corrispondenti ai tipi primitivi del linguaggio ed ai tipi che l'utente può definire (se il linguaggio lo permette)
- tratteremo insieme le due classi
  - anche se il componente "dati" del linguaggio comprende ovviamente solo i tipi di dato di programma

Cos'è un Tipo di Dato e cosa vogliamo sapere di lui

- una collezione di valori
  - rappresentati da opportune strutture dati +
- un insieme di operazioni per manipolarli
- come sempre ci interessano a due livelli
  - semantica
    - una specie di semantica algebrica "implementata" con i meccanismi per la definizione di nuovi tipi (mediante termini) in OCAML
  - implementazione
    - altre implementazioni in OCAML, date tipicamente in termini di strutture dati "a basso livello" (array)

I Descrittori di Dato

- immaginiamo di voler rappresentare una collezione di valori utilizzando quanto ci viene fornito da un linguaggio macchina
  - un po' di tipi numerici, caratteri, etc.
  - sequenze di celle di memoria
- qualunque valore è alla fine una stringa di bits
- per poter riconoscere il valore e interpretare correttamente la stringa di bits
  - è necessario (in via di principio) associare alla stringa un'altra struttura che contiene la descrizione del tipo (*descrittore di dato*), che viene usato ogniqualvolta si applica al dato un'operazione
    - per controllare che il tipo del dato sia quello previsto dall'operazione (type checking "dinamico")
    - per selezionare l'operatore giusto per eventuali operazioni overloaded

Tipi a tempo di compilazione e a tempo di esecuzione

- se l'informazione sul tipo è conosciuta completamente "a tempo di compilazione"
  - si possono eliminare i descrittori di dato
  - il type checking è effettuato totalmente dal compilatore (type checking statico)
  - FORTRAN, ML
- se l'informazione sul tipo è nota solo "a tempo di esecuzione"
  - sono necessari i descrittori per tutti i tipi di dato
  - il type checking è effettuato totalmente a tempo di esecuzione (type checking dinamico)
  - LISP, PROLOG
- se l'informazione sul tipo è conosciuta solo parzialmente "a tempo di compilazione"
  - i descrittori di dato contengono solo l'informazione "dinamica"
  - il type checking è effettuato in parte dal compilatore ed in parte dal supporto a tempo di esecuzione
  - JAVA

Tipi a tempo di compilazione: specifica o inferenza?

- l'informazione sul tipo viene di solito fornita con delle asserzioni (specifiche)
  - nelle dichiarazioni di costanti e variabili
  - nelle dichiarazioni di procedura (tipi dei parametri e tipo del risultato)
  - i fans di LISP sostengono che essere costretti a specificare tipi ovunque e sempre è palloso e rende il linguaggio poco flessibile, ma ....
- in alternativa (o in aggiunta) alle asserzioni fornite nel programma, il linguaggio può essere dotato di un algoritmo di analisi statica che riesce ad inferire il tipo di ogni espressione
  - tipico dei linguaggi funzionali moderni (esempio, ML)

Tipi come valori esprimibili e denotabili

- importante strumento per la definizione di astrazioni sui dati
  - manca del tutto nei linguaggi che ignorano i tipi (LISP e PROLOG) e nei linguaggi antichi (FORTRAN, ma anche ALGOL)
  - nasce con PASCAL
  - sempre più importante nei linguaggi funzionali, imperativi e object-oriented moderni
- ne parleremo alla fine di questo gruppo di lezioni

Semantica dei tipi di dato

- useremo per la semantica semplicemente OCAML
  - utilizzando il meccanismo dei tipi varianti (costruttori, etc.) per definire (per casi) un tipo (generalmente ricorsivo)
  - poche chiacchiere, alcuni esempi
    - due tipi di pila (non modificabile e modificabile)
    - liste (non polimorfe)
    - termini, sostituzioni, unificazione, pattern matching
    - S-espressioni (LISP)

Pila non modificabile: interfaccia

```
# module type PILA =
   sig
     type 'a stack
     val emptystack : int * 'a -> 'a stack
     val push : 'a * 'a stack -> 'a stack
     val pop : 'a stack -> 'a stack
     val top : 'a stack -> 'a
     val empty : 'a stack -> bool
     val lungh : 'a stack -> int
     exception Emptystack
     exception Fullstack
   end
```

Pila non modificabile: semantica

```
# module SemPila: PILA =
  struct
    type 'a stack = Empty of int | Push of 'a stack * 'a
    exception Emptystack
    exception Fullstack
    let emptystack (n, x) = Empty(n)
    let rec max = function
          | Empty n -> n
          | Push(p,a) -> max p
    let rec lungh = function
          | Empty n -> 0
          | Push(p,a) -> 1 + lungh(p)
    let push (a, p) = if lungh(p) = max(p) then raise Fullstack else Push(p,a)
    let pop = function
          | Push(p,a) -> p
          | Empty n -> raise Emptystack
    let top = function
          | Push(p,a) -> a
          | Empty n -> raise Emptystack
    let empty = function
          | Push(p,a) -> false
          | Empty n -> true
   end
```

Pila non modificabile: semantica algebrica

- semantica “isomorfa” ad una specifica in stile algebrico
  - trascuriamo i casi eccezionali

```
'a stack = Empty of int | Push of 'a stack * 'a
emptystack (n, x) = Empty(n)
lungh(Empty n) = 0
lungh(Push(p,a)) = 1 + lungh(p)
push(a,p) = Push(p,a)
pop(Push(p,a)) = p
top(Push(p,a)) = a
empty(Empty n) = true
empty(Push(p,a)) = false
```

- tipo (ricorsivo) definito per casi (con costruttori)
- semantica delle operazioni definita da un insieme di equazioni fra termini
- il tipo di dato è un’algebra (iniziale)

Pila non modificabile: implementazione

```
# module ImpPila: PILA =
  struct
    type 'a stack = Pila of ('a array) * int
    exception Emptystack
    exception Fullstack
    let emptystack (nm,x) = Pila(Array.create nm x, -1)
    let push(x, Pila(s,n)) = if n = (Array.length(s) - 1) then
                    raise Fullstack else
                    (Array.set s (n +1) x;
                    Pila(s, n +1))
    let top(Pila(s,n)) = if n = -1 then raise Emptystack
                              else Array.get s n
    let pop(Pila(s,n)) = if n = -1 then raise Emptystack
                              else Pila(s, n -1)
    let empty(Pila(s,n)) = if n = -1 then true else false
    let lungh(Pila(s,n)) = n
  end
```

Pila non modificabile: implementazione

```
# module ImpPila: PILA =
  struct
    type 'a stack = Pila of ('a array) * int
    ..........
  end
```

- il componente principale dell'implementazione è un array
  - memoria fisica in una implementazione in linguaggio macchina
- classica implementazione sequenziale
  - utilizzata anche per altri tipi di dato simili alle pile (code)

Lista (non polimorfa): interfaccia

```
# module type LISTAINT =
    sig
      type intlist
      val emptylist :  intlist
      val cons : int  * intlist -> intlist
      val tail : intlist -> intlist
      val head : intlist -> int
      val empty : intlist -> bool
      val length : intlist -> int
      exception Emptylist
    end
```

Lista: semantica

```
# module SemListaInt: LISTAINT =
  struct
    type intlist = Empty | Cons of int * intlist
    exception Emptylist
    let emptylist = Empty
    let rec length = function
          | Empty  -> 0
          | Cons(n,l) -> 1 + length(l)
    let cons (n, l) = Cons(n, l)
    let tail = function
          | Cons(n,l) -> l
          | Empty  -> raise Emptylist
    let head = function
          | Cons(n,l) -> n
          | Empty  -> raise Emptylist
    let empty = function
          | Cons(n,l) -> false
          | Empty  -> true
  end
```

Lista e Pila: stessa “semantica”

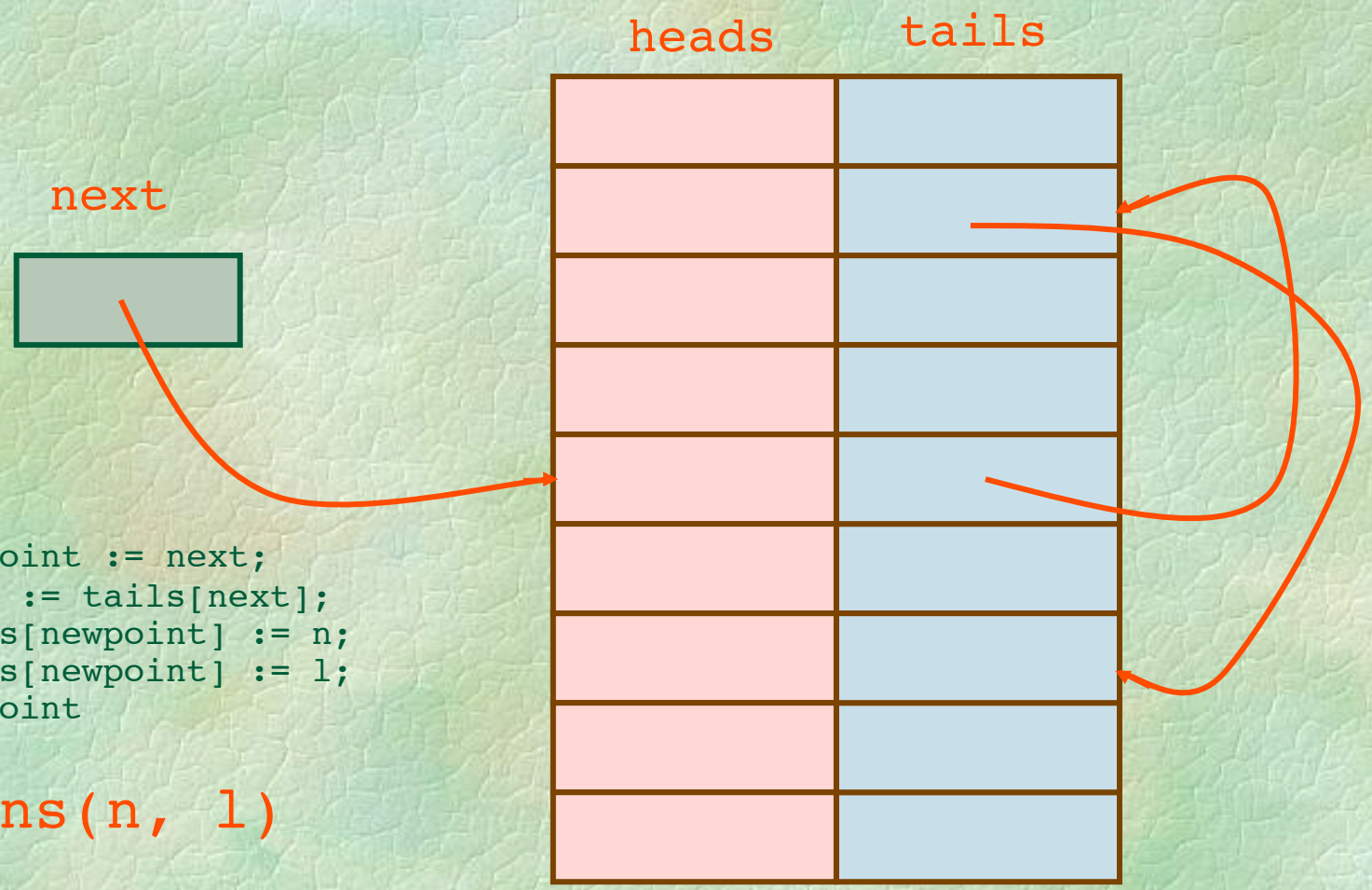
```
intlist = Empty | Cons of int * intlist
emptylist = Empty
length(Empty) = 0
length(Cons(n,l)) = 1 + length(l)
cons (n, l) = Cons(n, l)
tail(Cons(n,l)) = l
head(Cons(n,l)) = n
empty(Empty) = true
empty(Cons(n,l)) = false

'a stack = Empty of int | Push of 'a stack * 'a
emptystack (n, x) = Empty(n)
lungh(Empty n) = 0
lungh(Push(p,a)) = 1 + lungh(p)
push(a,p) = Push(p,a)
pop(Push(p,a)) = p
top(Push(p,a)) = a
empty(Empty n) = true
empty(Push(p,a)) = false
```

Stessa “implementazione”?

- non conviene implementare una lista con un array
  - vorremmo una implementazione sequenziale in cui un unico array viene utilizzato per rappresentare “tante” liste
  - la heap!
- l’array unico va bene per la pila perché è un tipo di dato di sistema
  - ne esistono un numero piccolo e predefinito nella implementazione del linguaggio
- la lista è tipicamente un tipo di dato d’utente
  - che ne può costruire un numero arbitrario nei propri programmi
- mostriamo l’implementazione delle liste con una heap senza operazioni per “disallocare”

Heap, lista libera, allocazione




```
newpoint := next;
next := tails[next];
heads[newpoint] := n;
tails[newpoint] := l;
newpoint
```

`cons(n, l)`

Lista: implementazione a heap

```
# module ImpListaInt: LISTAINT =
  struct
    type intlist = int
    let heapsize = 100
    let heads = Array.create heapsize 0
    let tails = Array.create heapsize  0
    let next = ref(0)
    let emptyheap =
        let index = ref(0) in
        while !index < heapsize do
          Array.set tails !index (!index + 1); index := !index + 1
        done;
        Array.set tails (heapsize - 1) (-1); next := 0
    exception Fullheap
    exception Emptylist
    let emptylist = -1
    let empty l = if l = -1 then true else false
    let cons (n, l) = if !next = -1 then raise Fullheap else
      ( let newpoint = !next in  next := Array.get tails !next;
        Array.set heads newpoint n; Array.set tails newpoint l; newpoint)
    let tail l = if empty l then raise Emptylist else Array.get tails l
    let head l = if empty l then raise Emptylist else Array.get heads l
    let rec length l = if l = -1 then 0 else 1 + length (tail l)
  end
```

Termini

- strutture ad albero composte da simboli
  - di variabile
  - di funzione ad n argomenti (n>= 0) (costruttori)
- un termine è
  - un simbolo di variabile
  - un simbolo di funzione n-aria applicato ad n termini
- i valori di un tipo ML definito per casi sono termini senza variabili
  - `type intlist = Empty | Cons of int * intlist`
  - `Cons(3,Empty),  Cons(1, Cons(2,Empty))`
- i pattern ML usati per selezionare i casi sono termini con variabili
  - ```
    let head = function
                | Cons(n,l) -> n
    ```
    ```

Termini con variabili e pattern matching

- i termini con variabili
  - `Cons(n,l)`

  "rappresentano" insiemi possibilmente infiniti di strutture dati
- il pattern matching
  - struttura dati (termine senza variabili) vs. pattern

  può essere utilizzato per definire "selettori"
  - ```
    let head = function
             | Cons(n,l) -> n
    ```
- le variabili del pattern vengono "legate" a sottotermini (componenti della struttura dati)
    ```

Termini e unificazione

- unificazione
  - tra due termini con variabili

  le variabili in uno qualunque dei due termini possono essere “legate” a sottotermini
- l’algoritmo di unificazione calcola la sostituzione più generale che rende uguali i due termini oppure fallisce
  - la sostituzione è rappresentata da un insieme di equazioni fra termini “in forma risolta”
    - equazioni del tipo `variabile = termine`
    - `variabile` non occorre in nessun `termine`

Termini, sostituzioni: interfaccia

```
# module type TERM =
   sig
     type term
     type equationset
     exception MatchFailure
     exception UndefinedMatch
     exception UnifyFailure
     exception OccurCheck
     exception WrongSubstitution
     val unifyterms : term * term -> equationset
     val matchterms : term * term -> equationset
     val instantiate : term * equationset  -> term
   end
```

Termini, sostituzioni: 1

```
# module Term: TERM =
  struct
   type term = Var of string | Cons of string * (term list)
   type equationset = (term * term) list
   exception MatchFailure
   exception UndefinedMatch
   exception UnifyFailure
   exception OccurCheck
   exception WrongSubstitution

   let rec ground = function
     | Var _ -> false
     | Cons(_, tl) -> groundl tl
   and groundl = function
     | [] -> true
     | t::tl1 -> ground(t) & groundl(tl1)
   ........
```

Termini, sostituzioni: 2

```
# module Term: TERM =
  struct
   type term = Var of string | Cons of string * (term list)
   type equationset = (term * term) list
   .........
   let rec occurs (stringa, termine) = match termine with
       | Var st -> stringa = st
       | Cons(_,tl) -> occursl (stringa, tl)
   and occursl (stringa, tl) = match tl with
       | [] -> false
       | t::tl1 -> occurs(stringa,t) or occursl(stringa, tl1)
   let rec occursset (stringa, e) = match e with
       | [] -> false
       | (t1,t2)::e1 -> occurs(stringa, t1) or occurs(stringa, t2)
              or occursset(stringa, e1)
   let rec rplact (t, v, t1) = match t with
     | Var v1 -> if v1 = v then t1 else t
     | Cons(s,tl) -> Cons(s, rplactl(tl,v,t1))
   and rplactl (tl, v, t1) = match tl with
     | [] -> []
     | t::tl1 -> rplact(t, v, t1) :: rplactl(tl1, v, t1)
   .....
```

Termini, sostituzioni: 3

```
# module Term: TERM =
  struct
    type term = Var of string | Cons of string * (term list)
    type equationset = (term * term) list
    .........
    let rec replace (eqset, stringa, termine) = match eqset with
        | [] -> []
        | (s1, t1) :: eqset1 ->
           (rplact(s1,stringa,termine),rplact(t1,stringa,termine))::
            replace(eqset1, stringa, termine)

    let rec pairs = function
      | [], [] -> []
      | a::l1, b::l2 -> (a,b)::pairs(l1, l2)
      | _ -> raise UnifyFailure
    ..........
```

Termini, sostituzioni: 4

```
# module Term: TERM =
  struct
    type term = Var of string | Cons of string * (term list)
    type equationset = (term * term) list
    .........
    let rec unify  (eq1, eq2) = match eq1 with
        | [] -> eq2
        | (Var x, Var y) :: eq11 -> if x = y then unify(eq11, eq2) else
            unify(replace(eq11,x,Var y),(Var x,Var y)::(replace(eq2,x,Var y)))
        | (t, Var y) :: eq11 -> unify((Var y,t)::eq11,eq2)
        | (Var x, t) :: eq11 -> if occurs(x,t) then raise OccurCheck else
          unify(replace(eq11,x,t),(Var x,t)::(replace(eq2,x,t)))
        | (Cons(x,xl), Cons(y,yl)):: eq11 -> if not(x = y) then
            raise UnifyFailure else unify(pairs(xl,yl)@eq11,eq2)

    let unifyterms (t1, t2) = unify([(t1,t2)],[])
    ..........
```

Termini, sostituzioni: 5

```
# module Term: TERM =
  struct
    type term = Var of string | Cons of string * (term list)
    type equationset = (term * term) list
    .........
    let rec pmatch  (eq1, eq2) = match eq1 with
        | [] -> eq2
        | (Var x, t) :: eq11 -> if occursset(x, eq11@eq2) then
          raise UndefinedMatch else pmatch(eq11,(Var x, t)::eq2)
        | (Cons(x,xl), Cons(y,yl)):: eq11 -> if not(x = y) then
             raise MatchFailure else pmatch(pairs(xl,yl)@eq11, eq2)
        | _ -> raise UndefinedMatch

   let matchterms (pattern, termine) =
        if not (ground(termine))  then raise UndefinedMatch else
        pmatch([(pattern,termine)],[])

    let rec instantiate (t, e) = match e with
      | [] -> t
      | (Var x, t1) :: e1 -> instantiate (rplact(t, x, t1), e1)
      | _ -> raise WrongSubstitution
   end
```

Ambiente (env)

- tipo (polimorfo) utilizzato nella semantica e nelle implementazioni per mantenere una associazione fra stringhe (identificatori) e valori di un opportuno tipo
- la specifica definisce il tipo come funzione
- l'implementazione che vedremo utilizza le liste
- è simile il dominio store

Ambiente: interfaccia

```
# 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
```

Ambiente: semantica

```
# module Funenv:ENV =
    struct
      type 't env = string -> 't
      exception WrongBindlist
      let emptyenv(x) = function y -> x
      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
```

Ambiente: implementazione

```
# module Listenv:ENV =
    struct
      type 't env = (string * 't) list
      exception WrongBindlist
      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
```

Tipi di dato modificabili

- a livello semantico, riconduciamo la modificabilità alla nozione di variabile
  - lo stato “modificabile” corrispondente sarà in realtà modellato con il dominio store
- per l’implementazione usiamo varie strutture dati modificabili come l’array

Pila modificabile: interfaccia

```
# module type MPILA =
  sig
   type 'a stack
   val emptystack : int * 'a -> 'a stack
   val push : 'a * 'a stack -> unit
   val pop : 'a stack -> unit
   val top : 'a stack -> 'a
   val empty : 'a stack -> bool
   val lungh : 'a stack -> int
   val svuota : 'a stack -> unit
   val access : 'a stack * int -> 'a
   exception Emptystack
   exception Fullstack
   exception Wrongaccess
  end
```

Pila modificabile: semantica

```
# module SemMPila: MPILA =
  struct
   type 'a stack = ('a SemPila.stack) ref
   exception Emptystack
   exception Fullstack
   exception Wrongaccess
   let emptystack (n, a) = ref(SemPila.emptystack(n, a) )
   let lungh x = SemPila.lungh(!x)
   let push (a, p) = p :=  SemPila.push(a, !p)
   let pop x =  x := SemPila.pop(!x)
   let top x = SemPila.top(!x)
   let empty x = SemPila.empty !x
   let rec svuota x =  if empty(x) then () else (pop x; svuota x)
   let rec faccess (x, n) =
      if n = 0 then SemPila.top(x) else faccess(SemPila.pop(x), n-1)
   let access (x, n) = let nofpops = lungh(x) - 1 - n in
      if nofpops < 0 then raise Wrongaccess else faccess(!x, nofpops)
  end
```

Pila modificabile: implementazione

```
module ImpMPila: MPILA =
  struct
   type 'x stack = ('x array) * int ref
   exception Emptystack
   exception Fullstack
   exception Wrongaccess
   let emptystack(nm,(x: 'a)) = ((Array.create nm x, ref(-1)): 'a stack)
   let push(x,((s,n): 'x stack)) = if !n = (Array.length(s) - 1) then
                  raise Fullstack else (Array.set s (!n +1) x; n := !n +1)
   let top(((s,n): 'x stack)) = if !n = -1 then raise Emptystack
                           else Array.get s !n
   let pop(((s,n): 'x stack)) = if !n = -1 then raise Emptystack
                           else n:= !n -1
   let empty(((s,n): 'x stack)) = if !n = -1 then true else false
   let lungh( (s,n): 'x stack) = !n
   let svuota (((s,n): 'x stack)) = n := -1
   let access (((s,n): 'x stack), k) =
    (*          if not(k > !n) then   *)
         Array.get s k
    (*          else raise Wrongaccess  *)
 end
```

S-espressioni

- la struttura dati fondamentale di LISP
  - alberi binari con atomi (stringhe) sulle foglie
  - modificabili
- vedremo solo interfaccia e semantica
  - l'implementazione (a heap) simile a quella delle liste

S-espressione: interfaccia

```
# module type SEXPR =
   sig
     type sexpr
     val nil : sexpr
     val cons : sexpr * sexpr  -> sexpr
     val node : string -> sexpr
     val car : sexpr -> sexpr
     val cdr : sexpr -> sexpr
     val null : sexpr -> bool
     val atom : sexpr -> bool
     val leaf : sexpr -> string
     val rplaca: sexpr * sexpr -> unit
     val rplacd: sexpr * sexpr -> unit
     exception NotALeaf
     exception NotACons
   end
```

S-espressione: semantica

```
# module SemSexpr: SEXPR =
  struct
   type sexpr = Nil| Cons of (sexpr ref) * (sexpr ref)| Node of string
   exception NotACons
   exception NotALeaf
   let nil = Nil
   let cons (x, y) = Cons(ref(x), ref(y))
   let node s = Node s
   let car =  function    Cons(x,y) -> !x
        | _ -> raise NotACons
   let cdr =  function  Cons(x,y) -> !y
        | _ -> raise NotACons
   let leaf =  function  Node x -> x
        | _ -> raise NotALeaf
   let null =  function  Nil -> true
        | _ -> false
   let atom =  function  Cons(x,y) -> false
        | _ -> true
   let rplaca =  function  (Cons(x, y), z) -> x := z
        | _ -> raise NotACons
   let rplacd =  function  (Cons(x, y), z) -> y := z
        | _ -> raise NotACons
  end
```

Programmi come dati

- la caratteristica fondamentale della macchina di Von Neumann
  - i programmi sono un particolare tipo di dato rappresentato nella memoria della macchina

  permette, in linea di principio, che, oltre all'interprete, un qualunque programma possa operare su di essi
- possibile sempre in linguaggio macchina
- possibile nei linguaggi ad alto livello
  - se la rappresentazione dei programmi è visibile nel linguaggio
  - e il linguaggio fornisce operazioni per manipolarla
- di tutti i linguaggi che abbiamo nominato, gli unici che hanno questa caratteristica sono LISP e PROLOG
  - un programma LISP è rappresentato come S-espressione
  - un programma PROLOG è rappresentato da un insieme di termini

Metaprogrammazione

- un metaprogramma è un programma che opera su altri programmi
- esempi: interpreti, analizzatori, debuggers, ottimizzatori, compilatori, etc.
- la metaprogrammazione è utile soprattutto per definire, nel linguaggio stesso,
  - strumenti di supporto allo sviluppo
  - estensioni del linguaggio

Meccanismi per la definizione di tipi di dato

- la programmazione di applicazioni consiste in gran parte nella definizione di “nuovi tipi di dato”
- un qualunque tipo di dato può essere definito in qualunque linguaggio
  - anche in linguaggio macchina
- gli aspetti importanti
  - quanto costa?
  - esiste il tipo?
  - il tipo è astratto?

Quanto costa? 1

- il costo della simulazione di un “nuovo tipo di dato” dipende dal repertorio di strutture dati primitive fornite dal linguaggio
  - in linguaggio macchina, le sequenze di celle di memoria
  - in FORTRAN e ALGOL’60, gli arrays
  - in PASCAL e C, le strutture allocate dinamicamente ed i puntatori
  - in LISP, le s-espressioni
  - in ML e Prolog, le liste ed i termini
  - in C++ e Java, gli oggetti

Quanto costa? 2

- è utile poter disporre di
  - strutture dati statiche sequenziali, come gli arrays e i records
  - un meccanismo per creare strutture dinamiche
    - tipo di dato dinamico (lista, termine, s-espressione)
    - allocazione esplicita con puntatori (à la Pascal-C, oggetti)

Esiste il tipo?

- anche se abbiamo realizzato una implementazione delle liste (con heap, lista libera, etc.) in FORTRAN o ALGOL
  - non abbiamo veramente a disposizione il tipo
- poichè i tipi non sono denotabili
  - non possiamo “dichiarare” oggetti di tipo lista
- stessa situazione in LISP e Prolog
- in PASCAL, ML, Java i tipi sono denotabili, anche se con meccanismi diversi
  - dichiarazioni di tipo
  - dichiarazioni di classe

Dichiarazioni di tipo

```
type nometipo = espressione di tipo
```

- il meccanismo di PASCAL, C ed ML
- il potere espressivo dipende dai meccanismi forniti per costruire espressioni di tipo (costruttori di tipo)
  - PASCAL, C
    - enumerazione
    - record, record ricorsivo
  - ML
    - enumerazione
    - prodotto cartesiano
    - iterazione
    - somma
    - funzioni
    - ricorsione
- ci possono essere tipi parametrici
  - in particolare, polimorfi (parametri di tipo tipo)

Dichiarazioni di classe

- il meccanismo di C++ e Java (anche OCAML)
- il tipo è la classe
  - parametrico in OCAML
  - con relazioni di sottotipo
- i valori del nuovo tipo (oggetti) sono creati con un'operazione di istanziazione della classe
  - non con una dichiarazione
- la parte struttura dati degli oggetti è costituita da un insieme di variabili istanza (o fields) allocati sulla heap

Il tipo è astratto?

- un tipo astratto è un insieme di valori
  - di cui non si conosce la rappresentazione (implementazione)
  - che possono essere manipolati solo con le operazioni associate
- sono tipi astratti tutti i tipi primitivi forniti dal linguaggio
  - 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
  - un meccanismo che permette di dare un nome al nuovo tipo (dichiarazione di tipo o di classe)
  - un meccanismo di "protezione" o information hiding che renda la rappresentazione visibile soltanto alle operazioni primitive
    - variabili istanza private in una classe
    - moduli e interfacce in C ed ML