



AA 2014-2015

## **19. Semantica dei linguaggi di programmazione e runtime support**

1



### Semantica e supporto a run time

- Le differenze fra i linguaggi di programmazione si riflettono in differenze nelle corrispondenti implementazioni
- Tutte le caratteristiche importanti per progettare un interprete o un supporto a tempo di esecuzione si possono ricavare analizzando la semantica operativa del linguaggio

2

## La nostra visione: semantiche “eseguibili”



- OCaml come metalinguaggio per esprimere la semantica dei linguaggi di programmazione
  - semantiche eseguibili che ci permettono di analizzare e valutare tutte le caratteristiche dei diversi paradigmi di programmazione e dei loro meccanismo di implementazione

3

**E i dati?**



4

## A cosa servono?



- ✎ *Livello di progetto*: organizzano l'informazione
  - tipi diversi per concetti diversi
  - meccanismi espliciti dei linguaggi per l'astrazione sui dati (ad esempio classi e oggetti)
- ✎ *Livello di programma*: identificano e prevengono errori
  - i tipi sono controllabili automaticamente
  - costituiscono un "controllo dimensionale"
    - ✓ l'espressione **3+“pippo”** deve essere sbagliata
- ✎ *Livello di implementazione*: permettono alcune ottimizzazioni
  - bool richiede meno bit di real
  - strumenti per fornire informazioni necessarie alla macchina astratta per allocare spazio di memoria

5

## Dati: classificazione



- ✎ *Denotabili*: se possono essere associati ad un nome
- ✎ *Esprimibili*: se possono essere il risultato della valutazione di una espressione complessa (diversa dal semplice nome)
- ✎ *Memorizzabili*: se possono essere memorizzati in una variabile

6

## Esempio: le funzioni in ML (puro)



### ☞ Denotabili

- let plus (x, y) = x + y

### ☞ Esprimibili

- let plus = function(x: int) -> function(y:int) -> x + y

### ☞ Memorizzabili

- NO

7

## Tipi



### ☞ Un tipo è una collezione di valori dotata di un insieme di operazioni per manipolare tali valori

- la distinzione tra collezioni di valori che sono tipi o non sono tipi è una nozione che dipende dal linguaggio di programmazione

## 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

9

## Sistema di tipi



- ✎ 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

10

## 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 linguaggio
- ☞ 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

11

## Regole di type checking



- ☞ **Regole di tipo** definiscono quando un costrutto del linguaggio soddisfa i requisiti di tipo

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

12

## 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>
```

13

## 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**

14

## 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 Robert 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

15

## Inferenza di tipo e type checking



- 👁️ Java, C, and C++, C# utilizzano un meccanismo di type checking
  - le annotazioni di tipo sono espliciti
- 👁️ ML, OCaml, F#, 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*)

16

## 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

## 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

18

## 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

19

## 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

20

## 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**) sono fortemente tipati con controllo dinamico

21

## 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#

22

# Analisi



- Polimorfismo paramerico (ML)
  - uno stesso algoritmo (codice) può avere molti tipi (basta rimpiazzare le variabili di tipo)
    - ✓ se  $f: t \rightarrow t$  allora  $f: \text{int} \rightarrow \text{int}$ ,  $f: \text{bool} \rightarrow \text{bool}$ , ...
- 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  $\text{int} * \text{int} \rightarrow \text{int}$ ,  $\text{real} * \text{real} \rightarrow \text{real}$

23

# 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*
    - ✓ definiscono lo stato e le strutture dati utilizzate nella simulazione di costrutti di controllo
  - i *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)
- ☞ Tratteremo insieme le due classi anche se il componente "dati" del linguaggio comprende ovviamente solo i tipi di dato di programma

24

## Cos'è un tipo di dato e cosa vogliamo sapere di lui



- ✎ Un TD è una collezione di valori
  - rappresentati da opportune strutture dati e un insieme di operazioni per manipolarli
- ✎ Come sempre ci interessano due livelli
  - semantica
  - implementazione

25

## I descrittori di dato



- ✎ Obiettivo: rappresentare una collezione di valori utilizzando quanto ci viene fornito da un linguaggio macchina
  - un po' di tipi numerici, caratteri
  - sequenze di celle di memoria
- ✎ 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 (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

26

## Tipi a tempo di compilazione e a tempo di esecuzione



1. **Se** l'informazione sui tipi è conosciuta completamente "a tempo di compilazione" (OCaml)
  1. si possono eliminare i descrittori di dato
  2. il type checking è effettuato totalmente dal compilatore (type checking statico)
2. Se l'informazione sui tipi è nota solo "a tempo di esecuzione" (JavaScript)
  1. sono necessari i descrittori per tutti i tipi di dato
  2. il type checking è effettuato totalmente a tempo di esecuzione (type checking dinamico)
3. Se l'informazione sui tipi è conosciuta solo parzialmente "a tempo di compilazione" (Java)
  1. i descrittori di dato contengono solo l'informazione "dinamica"
  2. il type checking è effettuato in parte dal compilatore e in parte dal supporto a tempo di esecuzione

27



## Tipi scalari

28

## Tipi scalari (esempi)



### Booleani

- o val: true, false
- o op: or, and, not, condizionali
- o repr: un byte
- o note: C non ha un tipo bool

### Caratteri

- o val: a,A,b,B, ..., è,é,ë, ; , ' , ...
- o op: uguaglianza; code/decode; dipendenti dal ling.
- o repr: un byte (ASCII) o due byte (UNICODE)

29

## Tipi scalari (esempi)



### Interi

- o val: 0,1,-1,2,-2,...,maxint
- o op: +, -, \*, mod, div, ...
- o repr: alcuni byte (2 o 4); complemento a due
- o note: interi e interi lunghi (anche 8 byte); limitati problemi nella portabilità quando la lunghezza non è specificata nella definizione del linguaggio

### Reali

- o val: valori razionali in un certo intervallo
- o op: +, -, \*, /, ...
- o repr: alcuni byte (4); virgola mobile
- o note: reali e reali lunghi (8 byte); gravi problemi di portabilità quando la lunghezza non è specificata nella definizione del linguaggio

30

## Tipi scalari (esempi)



### Il tipo `void`

- ha un solo valore
- nessuna operazione
- 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 interessa)

31

## Tipi composti



### Record

- collezione di campi (field), ciascuno di un (diverso) tipo
- un campo è selezionato col suo nome

### Record varianti

- record dove solo alcuni campi (mutuamente esclusivi) sono attivi a un dato istante

### Array

- funzione da un tipo indice (scalare) ad un altro tipo
- array di caratteri sono chiamati stringhe; operazioni speciali

### Insieme

- sottoinsieme di un tipo base

### Puntatore

- riferimento (*reference*) ad un oggetto di un altro tipo

32

# Record



- Introdotti per manipolare in modo unitario dati di tipo eterogeneo
- C, C++, CommonLisp, Ada, Pascal, Algol68
- Java: non ha tipi record, sussunti dalle classi
- Esempio in C

```
struct studente {
    char nome[20];
    int matricola; };
```
- Selezione di campo

```
studente s;
s.matricola = 343536;
```
- Record possono essere annidati
- Memorizzabili, esprimibili e denotabili
  - ✓ Pascal non ha modo di esprimere “un valore record costante”
  - ✓ C lo può fare, ma solo nell’inizializzazione (initializer)
  - ✓ uguaglianza generalmente non definita (contra: Ada)

33

# Record: implementazione



- 🔗 Memorizzazione sequenziale dei campi
- 🔗 Allineamento alla parola (16/32/64 bit)
  - spreco di memoria
- 🔗 Pudding o packed record
  - disallineamento
  - accesso più costoso

34

## Record: implementazione



```
struct x_  
{  
    char a;    // 1 byte  
    int b;    // 4 byte  
    short c;   // 2 byte  
    char d;    // 1 byte  
};
```

*L'allineamento alla parola determina uno spreco di occupazione di memoria*

35

## Record: implementazione



```
// effettivo "memory layout" (C COMPILER)  
struct x_{  
    char a;    // 1 byte  
    char _pad0[3]; // padding 'b' su 4 byte  
    int b;    // 4 byte  
    short c;   // 2 byte  
    char d;    // 1 byte  
    char _pad1[1]; // padding sizeof(x_)  
                    // multiplo di 4  
}
```

36

# Array



- Collezioni di dati omogenei
  - funzione da un tipo indice al tipo degli elementi
  - indice: in genere discreto
  - elemento: “qualsiasi tipo” (raramente un tipo funzionale)
- Dichiarazioni
  - C: `int vet[30];`                    tipo indice tra 0 e 29
- Array multidimensionali
- Principale operazione permessa
  - selezione di un elemento: `vet[3], mat[10,'c']`
  - attenzione: la modifica non è un'operazione sull'array, ma sulla locazione modificabile che memorizza un (elemento di) array

37

# Array: implementazione



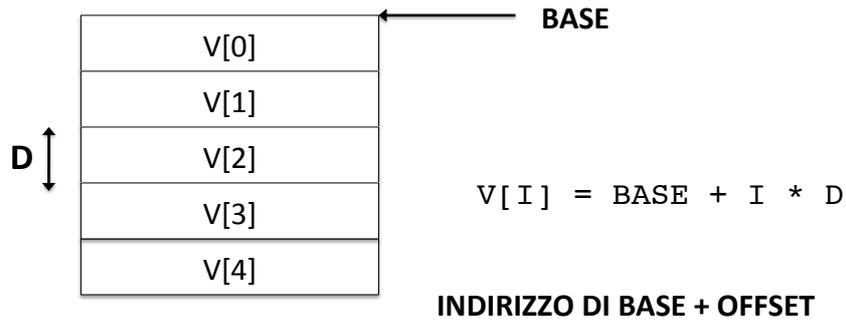
- Elementi memorizzati in locazioni contigue:
  - ordine di riga: `V[1,1];V[1,2];...;V[1,10];V[2,1];...`
    - ✓ maggiormente usato;
  - ordine di colonna: `V[1,1];V[2,1];V[3,1];...;V[10,1];V[1,2];...`
- Formula di accesso (caso lineare)
  - vettore `V[N]` of `elem_type`
  - $V[i] = \text{base} + c * i$ ,  
dove `c` è la dimensione per memorizzare un `elem_type`
- Un formula di accesso (più articolata) può essere stabilita anche per gli array multidimensionali (dettagli nel libro di testo)

38



## Accesso array: esempio

`int v[5]`



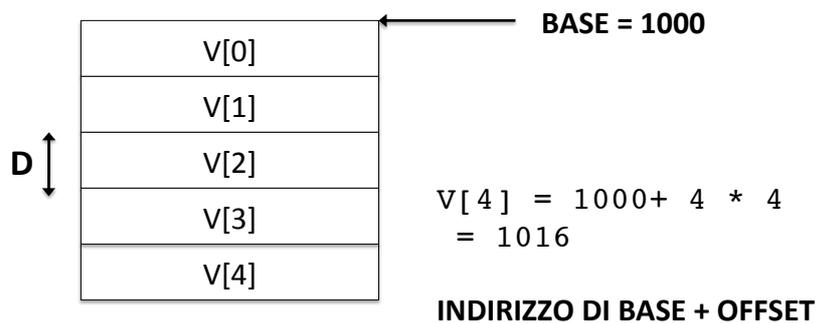
**D** dimensione in byte del tipo di base

39



## Accesso array: esempio

`int v[5]`



**D** dimensione in byte del tipo di base = 4 byte

40

## Il caso del C



- ✎ Il C non prevede controlli a runtime sulla correttezza degli indici di array
- ✎ Esempio: un array di 20 elementi di dimensione 2 byte allocato all'indirizzo 1000, l'ultima cella valida (indice 19) è allocata all'indirizzo 1038
- ✎ Se il programma, per errore, tenta di accedere il vettore all'indice 40, il runtime non rileverà l'errore e fornirà un accesso scorretto alla locazione di memoria 1080

41

## Puntatori



- ✎ Valori : riferimenti; costante **null (nil)**
- ✎ Operazioni
  - creazione
    - ✓ funzioni di libreria che alloca e restituisce un puntatore (e.g., **malloc**)
  - dereferenziazione
    - ✓ accesso al dato "puntato": **\*p**
  - test di uguaglianza
    - ✓ in specie test di uguaglianza con **null**

42

## Array e puntatori in C



- Array e puntatori sono intercambiabili in C (!!)

```
int n;  
int *a;    // puntatore a interi  
int b[10]; // array di 10 interi  
  
...  
a = b;     // a punta all'elemento iniziale di b  
n = a[3];  // n ha il valore del terzo elemento di b  
n = *(a+3); // idem  
n = b[3];  // idem  
n = *(b+3); // idem
```

- Ma `a[3] = a[3]+1;`  
modificherà anche `b[3]` (è la stessa cosa!)

43



## Tipi di dato di sistema

44

## 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 lugh : 'a stack -> int
    exception Emptystack
    exception Fullstack
  end
```

45

## Pila non modificabile: semantica



```
# module SemPila: PILA =
  struct
    type 'a stack = Empty of int | Push of 'a stack * 'a (*tipo algebrico *)
    exception Emptystack
    exception Fullstack
    let emptystack (n, x) = Empty(n)
    let rec max = function
      | Empty n -> n
      | Push(p,a) -> max p
    let rec lugh = function
      | Empty n -> 0
      | Push(p,a) -> 1 + lugh(p)
    let push (a, p) = if lugh(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
```

46

# Semantica algebrica



'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

Semantica "isomorfa" a una specifica in stile algebrico

Semantica delle operazioni definita da insiemi di equazioni fra termini

Il tipo di dato è un'algebra (iniziale)

47

# 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
```

48

## 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
  - (astrazione della) memoria fisica in una implementazione in linguaggio macchina
- Classica implementazione sequenziale
  - utilizzata anche per altri tipi di dato simili alle pile (code)

49

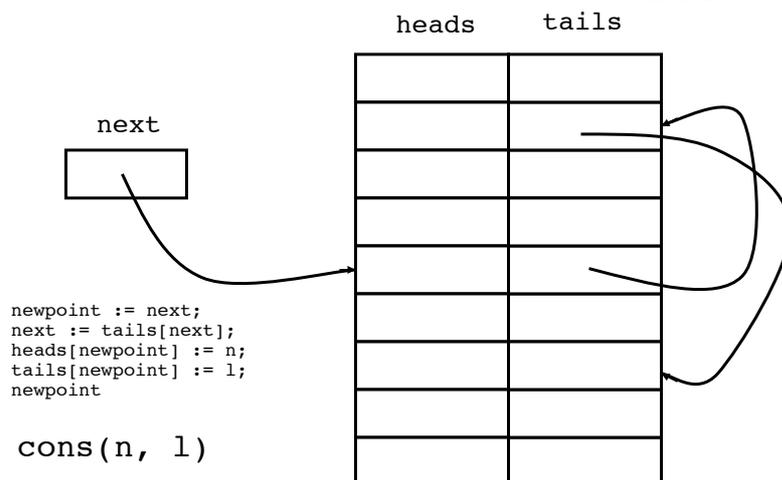
## Lista (non polimorfa): interfaccia



```
# 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
```

50

## Heap, lista libera, allocazione



51

## 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

```

52

## 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
```

53

## 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
```

54

## 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
```

55

## Programmi come dati



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

56

# Metaprogrammazione



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

57

# 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?

58

## 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 array
  - in PASCAL e C, le strutture allocate dinamicamente e i puntatori
  - in LISP, le s-espressioni
  - in ML e Prolog, le liste ed i termini
  - in C++ e Java, gli oggetti

59

## Quanto costa?, 2



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

60

## 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

61

## Dichiarazioni di classe



- ✎ Il meccanismo di C++ e Java (anche OCaml)
- ✎ Il tipo è la classe
  - parametrico, 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 field) allocati sullo heap

62

# 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 d’istanza private in una classe
    - ✓ moduli e interfacce in C e ML