Corso di Linguaggi di Programmazione

Lezione 4

Alberto Ceselli
alberto.ceselli@unimi.it

Dipartimento di Tecnologie dell'Informazione
Università degli Studi di Milano

5 Marzo 2013

Scheme - L'interprete

- Tutti gli interpreti Scheme sono basati sul seguente loop (Read-Eval-Print):
  1. **Leggi** un'espressione inserita dall'utente
  2. **Valuta** l'espressione per generare un risultato valido
  3. **Stampa a video** l'eventuale risultato
  4. Ripeti all'infinito.
- Scheme è tipicamente interattivo
  - reagisce immediatamente a quanto inserito
  - esisono anche compilatori, ma più di frequente interpreti

Funzioni aritmetiche

fun	# arg	**valore di ritorno**
+	0 o più	somma degli argomenti
−	1 o più	differenza degli argomenti (da sx a dx)
*	0 o più	prodotto degli argomenti
/	1 o più	quoziente degli argomenti (da sx a dx)
`max`	1 o più	massimo
`min`	1 o più	minimo
`truncate`	*num*	la parte intera di *num*
`sqrt`	*num*	la radice quadrata di *num*
`abs`	*num*	il valore assoluto di *num*
`expt`	*num pwr*	*num* alla *exp*
`quotient`	*num*-1 *num*-2	il quoziente di *num*-1 diviso *num*-2
`remainder`	*num*-1 *num*-2	il resto di *num*-1 diviso *num*-2

- + e * possono avere un qualsiasi numero di argomenti, nel caso non ce ne siano viene restituita l'identità (0 e 1)
- nel caso / abbia un solo argomento *n*, viene restituito $1/n$

Predicati Tipi Primitivi

fun	# arg	**restituisce** $\#t$ **sse**
`not`	*arg*	*arg* è $\#f$
$<$	*num-1* . . . *num-n*	*num-1* $<$ *num-2* . . . $<$ *num-n*
$>$	*num-1* . . . *num-n*	*num-1* $>$ *num-2* . . . $>$ *num-n*
$=$	*num-1* . . . *num-n*	*num-1* $=$ *num-2* . . . $=$ *num-n*
$<=$	*num-1* . . . *num-n*	*num-1* $<=$ *num-2* . . . $<=$ *num-n*
$>=$	*num-1* . . . *num-n*	*num-1* $>=$ *num-2* . . . $>=$ *num-n*
`zero?`	*num*	*num* $= 0$
`positive?`	*num*	*num* > 0
`negative?`	*num*	*num* < 0
`even?`	*num*	*num* pari
`odd?`	*num*	*num* dispari
`number?`	*num*	*num* è un numero
`real?`	*num*	*num* è un reale
`integer?`	*num*	*num* è un intero
`symbol?`	*num*	*num* è un simbolo

Valutare condizioni logiche in Scheme

- esistono due funzioni, `and` e `or` per costruire predicati complessi.
- esempi (DrRacket).

Ora possiamo usare l'interprete Scheme come una "calcolatrice di lusso".

Come associare un valore ad un simbolo?

- `(define identificatore valore)`
  - al simbolo `identificatore` viene associato `valore`
  - analogo alla definizione di una variabile globale
  - Es. `(define pi 3.14)`
- `(set! identificatore valore)`
  - viene modificato il valore associato al simbolo `identificatore`
  - potrebbe essere di tipo diverso
  - restituisce errore se non è mai stato associato prima un valore tramite forma speciale `define`

Costrutto `let`

```
(let (  (var-1 expr-1)
        (var-2 expr-2)
             ...
        (var-n expr-n) )
         body )
```

- Associa le espressioni alle variabili, poi valuta `body`, **senza seguire un ordine preciso**
  - quando `var-i` compare in `expr-j`, non è definito (ricorda che l'ordine della valutazione degli argomenti non è specificato)
  - `var-1` ... `var-n` sono visibili solo nel body (sono *locali*)

Costrutto let*

```
(let* (  (var-1 expr-1)
         (var-2 expr-2)
              ...
         (var-n expr-n) )
          body )
```

- In questo caso gli argomenti vengono valutati in ordine, quindi se una variabile è già stata definita, è possibile utilizzarla.

Definizione nuove funzioni

```
(define (nome-funzione lista-argomenti)
        corpo )
```

- alla funzione `corpo` viene associato il nome `nome-funzione`
- Es. `(define (somma a b) (+ a b))`

Special form `if`

```
(if condition
    action
    else-action)
```

- **Valutazione della special form `if`:**
  1. viene valutata la condizione `condition`
  2. a seconda del valore di ritorno di `condition`, viene valutata `action` o `else-action`
  3. il valore di ritorno dell'espressione valutata diventa il valore di ritorno dell'intera espressione
- il ramo else è opzionale (se assente e `condition` è `#f`, viene restituito un valore indefinito)

Special form cond

```
(cond
  (condition-1 action-1)
  (condition-2 action-2)
   ...
  (condition-n action-n)
  (else action-n+1))
```

Special form `cond`

- accetta un qualsiasi numero di argomenti, che devono avere la forma di una lista `(condizione azione)`
- **Valutazione della special form `cond`:**
  1. viene valutata ogni condizione nell'ordine dato
  2. quando una delle condizioni risulta vera, viene valutata *solo* l'azione-espressione associata
  3. il valore di ritorno dell'espressione valutata diventa il valore di ritorno dell'intera espressione
- il ramo else è opzionale (se assente e nessuna condizione è vera, restituisce un valore indefinito)

Esempi

Programmi Scheme per:

- fattoriale
- potenza
- numeri di Fibonacci

S&M Scheme expressions

Infatti ...

- In Scheme tutti i tipi di dato (simboli, numeri, atomi, liste, funzioni, forme speciali, ...) appartengono alla categoria delle *S-expression* (dove *S* sta per *Symbolic*) [McCarthy, Recursive Functions of Symbolic Expressions]
  - la caratteristica comune a tutti sta nell'uso delle parentesi come prefissi delle S-expressions (a volte conosciuta come prefix Cambridge Polish notation)
- Di conseguenza **dati e codice sono di fatto indistinguibili** e un programma può modificare se stesso, creare nuove funzioni ...
- La motivazione deriva dal desiderio di costruire un interprete che mimasse il comportamento della *macchina di Turing Universale*.

S&M Scheme expressions

- Le *S-expressions* venivano inizialmente impiegate solo come rappresentazione interna alla macchina delle *M-expressions* (M-notation expressions)
- Nell'idea di McCarthy, le M-expressions dovevano essere più simili possibile al Fortran (quindi anche più leggibili da parte di un programmatore).
- Tuttavia i programmatori Lisp iniziarono presto ad utilizzare le S-expressions come notazione di default.

L'interprete Scheme

In sintesi, l'interprete Scheme è una **funzione** che ...

- attende, come argomento, una *lista* (codice o dati, è indifferente)
- interpreta il *primo elemento* della lista come *parola chiave*
- valuta i successivi elementi *richiamando ricorsivamente se stessa*
- restituisce qualcosa (dati, funzioni ecc.) che è ottenuto "applicando" la parola chiave ai valori ottenuti dalla valutazione degli altri elementi nella lista.

Liste

- Sono uno dei tipi fondamentali di Scheme; ci sono molte funzioni predefinite per manipolarle
- Def. Una *lista* è una **sequenza ordinata** di elementi **eterogenei**
  - `()` lista vuota
  - `(1 2 3 4 5)` lista di 5 elementi - atomi
  - `(1 (2 3) 4 (5))` lista di 4 elementi - 2 atomi e 2 liste
  - `(sono una lista 4)` lista di 4 elementi - atomi

Liste

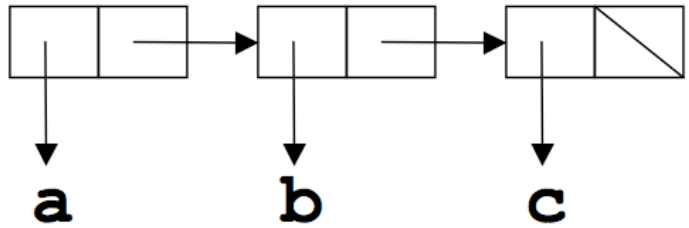
Valutazione di liste: perché una lista non venga interpretata come una funzione, deve venire commentata:

- `quote`: prende in input un'espressione e la restituisce **senza valutarla** (' versione abbreviata)
- `semiquote`: prende in input un'espressione e la restituisce valutando **solo** le espressioni precedute da una virgola
  - la forma abbeviata è ‘ (l'altro apice)
  - se l'espressione è preceduta da `,@`, l'espressione viene interpretata come una lista e il suo contenuto appeso al resto della lista.

Rappresentazione interna delle liste (1)

Rappresentazione testuale: `(a b c)`

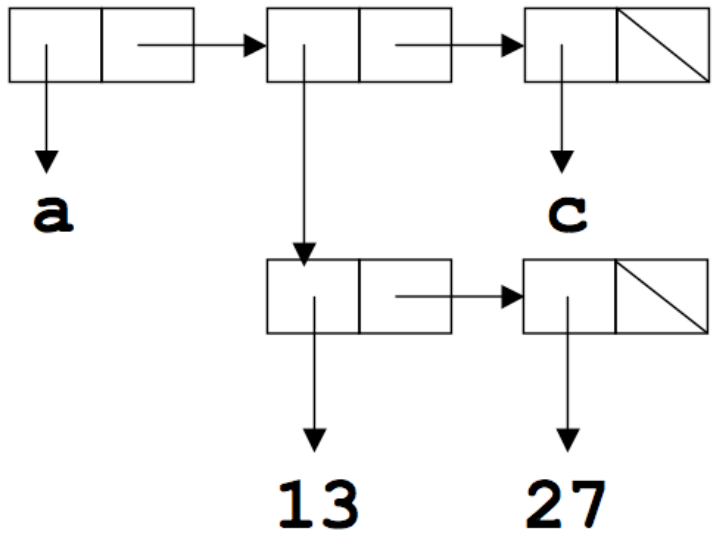
Rappresentazione interna:



Rappresentazione interna delle liste (2)

Rappresentazione testuale: `(a (13 27) c)`

Rappresentazione interna:

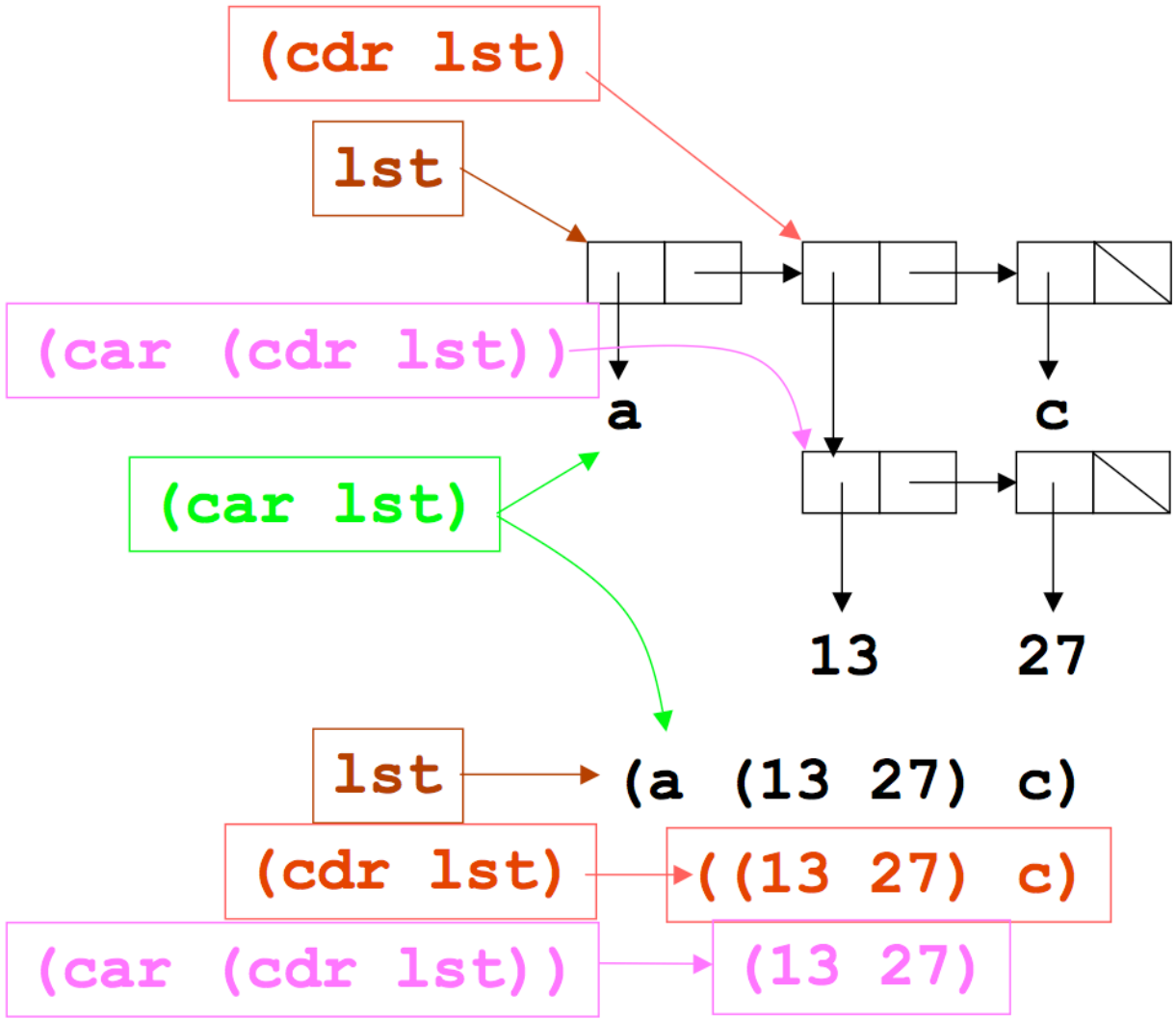


Funzioni predefinite su liste (1)

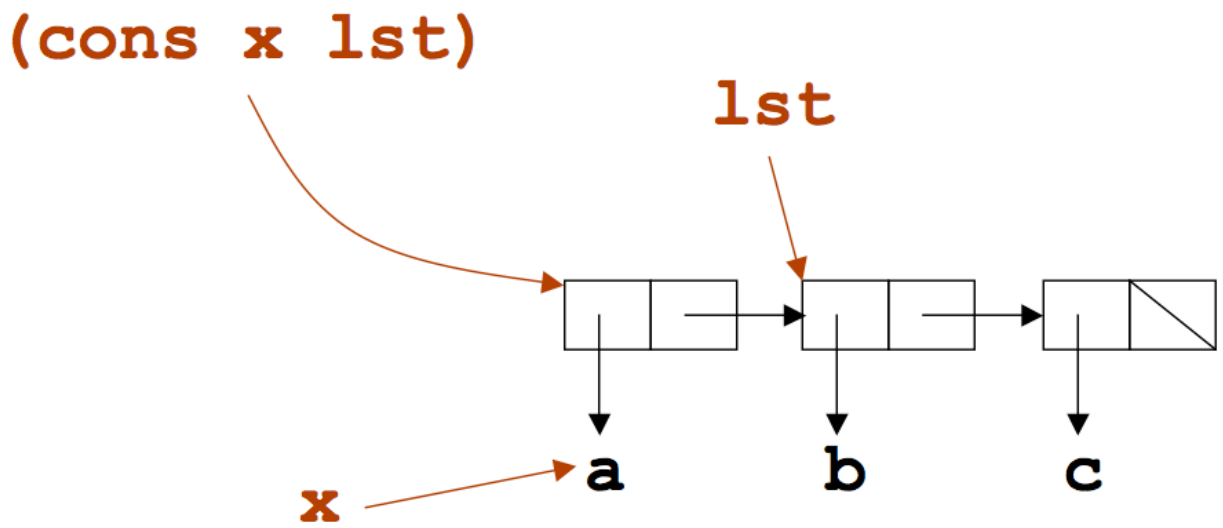
fun	args	valore ritorno
`car`	*lista*	il primo elemento di *lista*
`cdr`	*lista*	il resto di *lista* senza il primo elemento
`cons`	*elem lista*	*lista* con *elem* inserito in testa
`list`	*el1... eln*	crea la lista (*el1 el2 ... eln*)
`append`	*list1 ... listn*	la lista formata concatenando gli elementi di *list1 ... listn*
`list-ref`	*lista pos*	l'elemento alla posizione *pos* di *lista* (parte da 0)
`list-tail`	*pos lista*	il resto di *lista* da *pos* in poi
`length`	*lista*	il numero di elementi di *lista*
`member`	*elem lista*	il resto di *lista* a partire dalla prima occorrenza di *elem*, `#f` se *elem* non compare
`reverse`	*lista*	inverte gli elementi di *lista*

- Scheme permette di abbreviare la composizione di `car` e `cdr` (fino a quattro livelli di composizione), es `caar` o `cdaar`.

Funzioni predefinite su liste (2)



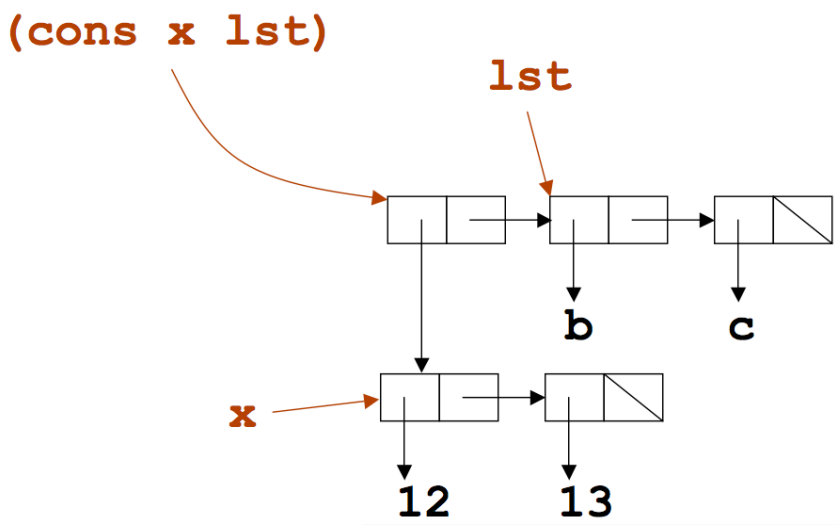
Funzioni predefinite su liste (3)




```
lst =    (b c)

(cons x lst) = (a b c)
```

Funzioni predefinite su liste (4)

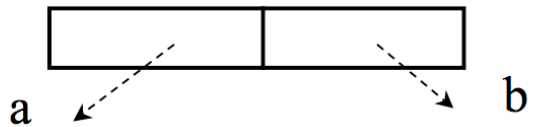



```
          lst =  (b c)
            x =  (12 13)
(cons x lst)= ((12 13) b c)
```

Rappresentazione interna delle liste (3) - Dotted Pairs

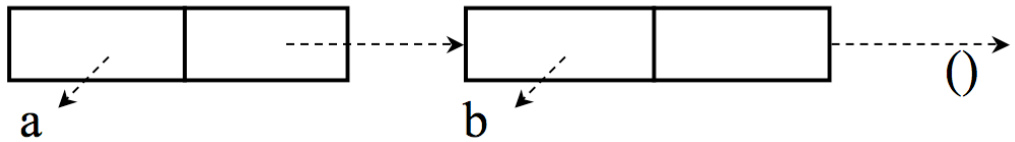
```
(cons 'a 'b)
```

= (a . b)




```
(cons 'a (cons 'b ()))
```

= ((a . b) . ()) = (a b)



Predicati su liste

pred	**args**	**restituisce vero sse**
`list?`	*arg*	*arg* è una lista
`null?`	*arg*	*arg* è una lista vuota
`equal?`	*arg1 arg2*	*arg1* è o appare lo stesso di *arg2*
`eqv?`	*arg1 arg2*	*arg1* è lo stesso di *arg2*
`eq?`	*arg1 arg2*	*arg1* è lo stesso di *arg2*, versione efficiente

Esercizi:

Implementare (o re-implementare) in Scheme semplici operazioni su liste:

- length
- member
- append

Non così semplici:

- reverse
- even_odd_split