

Linguaggi di programmazione

Prof.ssa Gabriella Pasi

a.a. 2018/2019

SOMMARIO

| | |
|---|-----------|
| Obiettivi | 9 |
| Programma esteso | 9 |
| Materiale didattico | 9 |
| Link utili..... | 9 |
| 1 Regole di scrittura del codice | 10 |
| 2 Una classificazione dei linguaggi di programmazione | 10 |
| 2.1 Paradigma imperativo (procedurale) | 10 |
| 2.2 Paradigmi funzionale e logico: caratteristiche comuni | 11 |
| 2.2.1 <i>Paradigma logico</i> | 11 |
| 2.3 Una comparazione tra lo stile prescrittivo e lo stile dichiarativo | 12 |
| 2.4 Paradigma funzionale | 12 |
| 2.5 Linguaggio LISP (LISt Processing) | 13 |
| 2.6 Ambienti “run time” di linguaggi logici, funzionali (e non) | 13 |
| 2.6.1 <i>Stack e valutazione di procedure: richiami</i> | 14 |
| 2.6.2 <i>Heap e Garbage collection</i> | 15 |
| 3 Introduzione alla logica..... | 16 |
| 3.1.1 <i>Regole di inferenza e calcoli logici</i> | 17 |
| 3.2 Logica proposizionale | 17 |
| 3.2.1 <i>Calcolo proposizionale: regole di inferenza</i> | 18 |
| 3.2.2 <i>Regole di inferenza</i> | 19 |
| 3.2.3 <i>Dimostrazioni per assurdo</i> | 20 |
| 3.2.4 <i>Assiomi (conoscenze pregresse)</i> | 20 |
| 3.2.5 <i>Esempio</i> | 21 |
| 3.2.6 <i>Sintassi e semantica</i> | 22 |
| 3.2.7 <i>Tautologie e modelli</i> | 22 |
| 4 Logica del primo ordine..... | 22 |
| 4.1 Logica proposizionale vs. logica del primo ordine..... | 22 |
| 4.2 Sintassi della logica del primo ordine | 23 |
| 4.3 Altre regole nei linguaggi del primo ordine..... | 24 |
| 5 Prolog | 25 |
| 5.1 Programmazione logica | 25 |
| 5.2 Programmazione logica e la logica matematica | 25 |
| 5.3 Stile dichiarativo della programmazione logica | 25 |
| 5.4 Prolog | 25 |

| | | |
|--------|--|----|
| 5.5 | Formule ben formate (FBF) e forma normale “a clausole” | 25 |
| 5.5.1 | <i>Forma normale congiunta (Conjunctive Normal Form – CNF)</i> | 26 |
| 5.6 | Prolog – linguaggio dichiarativo | 27 |
| 5.6.1 | <i>Termini</i> | 27 |
| 5.6.2 | <i>Fatti o predicati</i> | 28 |
| 5.6.3 | <i>Le regole</i> | 28 |
| 5.6.4 | <i>Relazioni definite da più regole</i> | 29 |
| 5.6.5 | <i>Ricorsione</i> | 29 |
| 5.6.6 | <i>Operatori logici</i> | 29 |
| 5.6.7 | <i>Sintassi</i> | 29 |
| 5.6.8 | <i>Interrogazioni (queries o goals)</i> | 29 |
| 5.6.9 | <i>Esempio di programma Prolog</i> | 30 |
| 5.6.10 | <i>Le variabili delle interrogazioni</i> | 30 |
| 5.6.11 | <i>Unificazione</i> | 30 |
| 5.6.12 | <i>Diverse rappresentazioni di dati ed interrogazioni</i> | 31 |
| 5.6.13 | <i>Le liste in Prolog</i> | 32 |
| 5.6.14 | <i>L'operatore </i> | 32 |
| 5.6.15 | <i>L'interprete Prolog: consult</i> | 32 |
| 5.6.16 | <i>L'interprete Prolog: reconsult</i> | 33 |
| 5.7 | clausole..... | 34 |
| 5.7.1 | <i>Implicazione</i> | 34 |
| 5.7.2 | <i>Clausole di Horn</i> | 34 |
| 5.8 | Un programma logico | 35 |
| 5.8.1 | <i>Sostituzioni</i> | 35 |
| 5.8.2 | <i>Esecuzione di un programma</i> | 35 |
| 5.8.3 | <i>Risoluzione ad input lineare (SLD)</i> | 36 |
| 5.8.4 | <i>Strategia di selezione di un sottogoal</i> | 36 |
| 5.8.5 | <i>Modello di esecuzione Prolog</i> | 39 |
| 5.8.6 | <i>Estensioni</i> | 39 |
| 5.8.7 | <i>Il controllo di esecuzione di un programma</i> | 39 |
| 5.8.8 | <i>Il predicato cut “!”</i> | 39 |
| 5.9 | Predicati meta-logici | 45 |
| 5.10 | Ispezione di termini | 45 |
| 5.11 | Programmazione di ordine superiore..... | 46 |
| 5.11.1 | <i>Predicati su insieme</i> | 46 |
| 5.11.2 | <i>Predicati di ordine superiore e meta variabili</i> | 47 |

| | | |
|----------|---|-----------|
| 5.12 | Modifica della base di conoscenze | 48 |
| 5.13 | Input e Output in Prolog..... | 49 |
| 6 | Interpreti in “Prolog” | 50 |
| 6.1 | Automi | 50 |
| 6.2 | automi a pila..... | 51 |
| 6.3 | Meta-interpreti..... | 52 |
| 7 | Paradigmi di programmazione | 55 |
| 7.1 | I linguaggi imperativi | 55 |
| 7.2 | Effetti collaterali | 55 |
| 7.3 | Trasparenza referenziale | 56 |
| 7.4 | Linguaggi di programmazione funzionali..... | 56 |
| 7.4.1 | <i>Composizione</i> | 57 |
| 7.4.2 | <i>Ricorsione e operatori speciali</i> | 57 |
| 8 | Lisp e il paradigma funzionale | 58 |
| 8.1 | Programmazione funzionale: interpreti, ambienti e compilatori..... | 58 |
| 8.1.1 | <i>Espressioni in LISP</i> | 58 |
| 8.1.2 | <i>Ordine di valutazione</i> | 59 |
| 8.1.3 | <i>Definizione di variabili e di funzioni</i> | 60 |
| 8.1.4 | <i>Nomi in common lisp</i> | 61 |
| 8.1.5 | <i>Valutaizone di funzioni</i> | 61 |
| 8.1.6 | <i>Funzioni anonime: espressioni LAMBDA</i> | 62 |
| 8.1.7 | <i>Operatori speciali: condizionali</i> | 62 |
| 8.1.8 | <i>Operatori speciali: booleani</i> | 63 |
| 8.2 | Funzioni ricorsive..... | 63 |
| 8.3 | Strutture dati e funzioni | 64 |
| 8.4 | Le cons-cells e la funzione CONS | 64 |
| 8.5 | Liste e la funzione LIST..... | 66 |
| 8.6 | Liste..... | 66 |
| 8.6.1 | <i>Elementi di una lista</i> | 67 |
| 8.7 | Dati simbolici e operazione quote..... | 67 |
| 8.8 | Simboli, numeri, liste e “atomi” | 68 |
| 8.8.1 | <i>Simboli e liste (e altri elementi), ovvero le symbolic expressions</i> | 68 |
| 8.9 | Valutazione di espressioni e funzioni ricorsive..... | 69 |
| 8.9.1 | <i>Dettagli e funzione eval</i> | 69 |
| 8.9.2 | <i>Esempi con liste</i> | 69 |
| 8.9.3 | <i>Last</i> | 70 |

| | | |
|----------|---|-----------|
| 8.9.4 | <i>Ricorsioni semplici e doppie</i> | 70 |
| 8.10 | Funzioni di uguaglianza | 73 |
| 8.10.1 | <i>Eq1</i> | 73 |
| 8.10.2 | <i>Equal</i> | 73 |
| 8.11 | Liste e funzioni..... | 74 |
| 8.12 | Funzioni anonimie ed operatore lambda | 75 |
| 8.13 | Operatore lambda ed operatore let..... | 76 |
| 8.14 | Tipiche funzioni di ordine superiore..... | 76 |
| 8.14.1 | <i>Funzione compose</i> | 76 |
| 8.14.2 | <i>Funzione filter</i> | 77 |
| 8.14.3 | <i>Funzione accumula</i> | 77 |
| 8.15 | Utili variazioni sul tema | 78 |
| 8.16 | Input/output in common lisp | 79 |
| 8.16.1 | <i>Read</i> | 79 |
| 8.16.2 | <i>Print</i> | 80 |
| 8.16.3 | <i>Output</i> | 80 |
| 8.16.4 | <i>Output formattato</i> | 81 |
| 8.16.5 | <i>Streams common lisp</i> | 81 |
| 8.16.6 | <i>Interazione con l'ambiente common lisp</i> | 82 |
| 8.17 | Valutazione di espressioni e funzioni | 83 |
| 8.17.1 | <i>apply</i> | 83 |
| 8.17.2 | <i>eval</i> | 83 |
| 8.17.3 | <i>Sequenza di valutazioni</i> | 84 |
| 8.17.4 | <i>Funzioni utili per la valutazione delle sexp</i> | 85 |
| 8.18 | La rappresentazione interna di “funzioni” | 86 |
| 8.19 | ambienti (environments)..... | 86 |
| 8.19.1 | <i>Make-frame</i> | 86 |
| 8.19.2 | <i>Extend-env</i> | 86 |
| 8.20 | Riscrittura di espressioni | 87 |
| 9 | Funzioni Lisp utili | 88 |
| 9.1 | Commenti | 88 |
| 9.2 | Format, force-output..... | 88 |
| 9.3 | read-line e lettura valori..... | 89 |
| 9.4 | Defun | 89 |
| 9.5 | List..... | 90 |
| 9.6 | Plist | 91 |

| | | |
|-----------|--|--|
| 9.7 | Getf..... | 91 |
| 9.8 | defvar, defparameter, variabili globali e push | 92 |
| 9.9 | loop..... | 92 |
| 9.10 | with-open-file | 93 |
| 9.11 | Print | 93 |
| 9.12 | remove-if-not | 93 |
| 9.13 | Lambda | 93 |
| 9.14 | Funzioni con selettori vari | 94 |
| 9.15 | evenp, oddp..... | 95 |
| 9.16 | Mapcar..... | 95 |
| 9.17 | IF, case | 95 |
| 9.18 | quote | 96 |
| 9.19 | EQ, EQL, EQUAL, EQUALP | 96 |
| 9.20 | Function | 96 |
| 9.21 | Funcall, apply..... | 96 |
| 9.22 | Let | 97 |
| 9.23 | When, unless | 97 |
| 9.24 | Cond..... | 98 |
| 9.25 | And, or, not..... | 98 |
| 9.26 | Do, dolist, dotimes..... | 98 |
| 9.27 | Loop | 99 |
| 9.27.1 | <i>Forma semplice</i> | 100 |
| 9.27.2 | <i>Forma estesa</i> | 100 |
| 9.28 | Defmacro | 100 |
| 9.29 | String e chars | 101 |
| 9.30 | Collezioni | 102 |
| 9.30.1 | <i>Vettori e array</i> | 102 |
| 9.31 | Subsequence Manipulations | 104 |
| 9.32 | I/O..... | 104 |
| 9.33 | NTH | 105 |
| 10 | Introduzione a C e C++..... | 106 |
| 10.1 | Compilazione ed esecuzione | 106 |
| 10.2 | Tipi fondamentali..... | 106 |
| 10.3 | Variabili..... | Errore. Il segnalibro non è definito. |
| 10.3.1 | <i>Arrays</i> | 107 |
| 10.3.2 | <i>Strutture</i> | 108 |

| | | |
|-----------|--|------------|
| 10.3.3 | <i>Puntatori</i> | 108 |
| 10.3.4 | <i>Puntatori ed array</i> | 109 |
| 10.3.5 | <i>Puntatori, array e aritmetica</i> | 109 |
| 10.3.6 | <i>Puntatori, array, aritmetica e stringhe</i> | 110 |
| 10.4 | Blocchi e operatori condizionali e di iterazione | 110 |
| 10.5 | Altri “statements” | 111 |
| 10.6 | Espressioni..... | 111 |
| 10.7 | Funzioni | 112 |
| 10.7.1 | <i>Funzioni e variabili locali</i> | 112 |
| 10.7.2 | <i>Funzioni e passaggio di parametri</i> | 113 |
| 10.8 | La compilazione dei programmi in C/C++..... | 114 |
| 10.8.1 | <i>Preprocessore</i> | 115 |
| 10.8.2 | <i>Compilazione separata e “header” files</i> | 115 |
| 10.9 | Come organizzare le librerie in C..... | 118 |
| 10.10 | Memoria dinamica in C/C++ | 119 |
| 10.11 | Modificatori di dichiarazione..... | 120 |
| 10.12 | Costanti..... | 120 |
| 10.13 | “Streams”, input, output e files..... | 121 |
| 10.13.1 | <i>Output</i> | 121 |
| 10.13.2 | <i>Streams e file handlers in C++</i> | 122 |
| 10.13.3 | <i>Input in C</i> | 122 |
| 10.13.4 | <i>Input in C++</i> | 123 |
| 10.14 | File I/O in C | 123 |
| 10.14.1 | <i>File I/O in C++</i> | 124 |
| 10.15 | Abbreviazioni: <code>typedef</code> | 124 |
| 11 | Code di priorità (priority queues) e C | 125 |
| 11.1 | Code con priorità | 125 |
| 11.2 | Esempi | 126 |
| 11.2.1 | <i>Ricerca degli elementi più grandi</i> | 126 |
| 11.2.2 | <i>Implementazioni elementari</i> | 126 |
| 12 | La struttura dati heap binario | 128 |
| 12.1 | Proprietà degli heap (mucchi) binari | 128 |
| 13 | Funzioni C utili (appunti economia) | 129 |
| 13.1 | Inclusione librerie | 129 |
| 13.2 | Dichiarazione tipi | 129 |
| 13.3 | Operatori | 130 |

| | | |
|-----------|--|------------|
| 13.3.1 | <i>Printf</i> | 130 |
| 13.3.2 | <i>scanf</i> | 130 |
| 13.3.3 | <i>if - else</i> | 131 |
| 13.3.4 | <i>switch-case</i> | 131 |
| 13.3.5 | <i>for</i> | 131 |
| 13.3.6 | <i>while</i> | 131 |
| 13.3.7 | <i>do-while</i> | 132 |
| 13.3.8 | <i>break</i> | 132 |
| 13.3.9 | <i>continue</i> | 132 |
| 13.3.10 | <i>exit</i> | 132 |
| 13.3.11 | <i>goto</i> | 132 |
| 13.3.12 | <i>system ("PAUSE")</i> | 132 |
| 13.4 | Operatori logici e matematici..... | 132 |
| 13.5 | Puntatore..... | 132 |
| 13.6 | funzioni | 133 |
| 14 | Lezione riassunto | 134 |
| A | Appendice A: Emacs e Prolog | 137 |
| B | Codici Lisp | 143 |
| C | Codici C/C++ | 144 |

OBIETTIVI

Gli studenti apprenderanno vari paradigmi programmazione, in particolare il paradigma logico e quello funzionale. Apprenderanno inoltre a utilizzare ambienti per programmare nei principali linguaggi presentati. Gli studenti saranno in grado di sviluppare progetti di piccole e medie dimensioni in (Common) Lisp e Prolog utilizzando gli ambienti di programmazione presentati.

PROGRAMMA ESTESO

- I paradigmi di programmazione: imperativo, logico (dichiarativo) e funzionale. Richiami delle nozioni di "run-time" e di esecuzione di un programma su un'architettura idealizzata a pila (stack). Nozioni base degli ambienti di programmazione per i diversi sistemi presentati.
- "Il paradigma di programmazione logico. Introduzione al linguaggio di programmazione Prolog."
- "Il paradigma di programmazione funzionale. Introduzione al linguaggio di programmazione LISP (Common Lisp)."
- "Il paradigma di programmazione imperativo. Introduzione al linguaggio di programmazione C."
- Utilizzo dei vari paradigmi in situazioni e contesti diversi.

MATERIALE DIDATTICO

Leon Sterling and Ehud Shapiro, *The Art of Prolog Advanced Programming Techniques* - 2nd Edition;

Abelson, Sussman e Sussman, *Structure and Interpretation of Computer Programs* (SICP);

Peter Seibel, *Practical Common Lisp* (PCL);

Brian W. Kernighan & Dennis M. Ritchie, *C Programming Language* (2nd Edition), Prentice Hall, 1988;

Robert Sedgewick, *Algorithms in C, Parts 1-5 (Bundle): Fundamentals, Data Structures, Sorting, Searching, and Graph Algorithms* (3rd Edition), Addison Wesley, 2001

LINK UTILI

- SWI Prolog: <http://www.swi-prolog.org>
- Lispworks Personal Edition: <http://www.lispworks.com/products/lispworks.html#personal>
- Haskell: <http://www.haskell.org>
- Node.js: <https://nodejs.org>

1 REGOLE DI SCRITTURA DEL CODICE

- Le linee non devono essere più lunghe di 80 colonne
- Attorno agli operatori e dopo le virgolette si usano gli spazi
 - `area = pi * radius ^ 2`
 - `colors = [red, green, blue]`
- NON si mette lo spazio tra il nome di una funzione predicato e la parentesi
 - `Factorial(42, 1)`
- Si mette uno spazio tra la parola chiave di uno statement di controllo e le parentesi
 - `While (itsFalse || itsTrue)`
 - `for (int i = 0; i < n; i++)`
 - `if (something > 42)`
- Non si usano commenti a scatola

2 UNA CLASSIFICAZIONE DEI LINGUAGGI DI PROGRAMMAZIONE

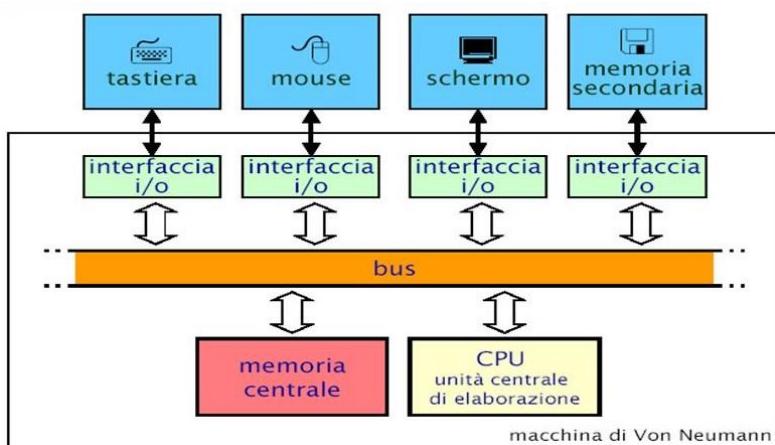
- Linguaggi imperativi
 - Vari assemblers, Fortran, Pascal, C, Python...
- Linguaggi logici
 - Prolog...
- Linguaggi funzionali
 - FP, List, Common Lisp, Scheme, Haskell...

Ognuna di queste categorie può contenere dei linguaggi ad oggetti, il paradigma della progettazione ad oggetti è quindi trasversale anche se presume alcune caratteristiche tipiche dei linguaggi imperativi (memoria e "stato" di un'istanza).

2.1 PARADIGMA IMPERATIVO (PROCEDURALE)

Le caratteristiche essenziali dei linguaggi imperativi sono strettamente legate all'architettura di Von Neumann. L'architettura è costituita da due componenti fondamentali:

- Memoria (componente passiva)
- Processore (componente attiva)



La principale attività del processore consiste nell'eseguire calcoli e assegnare valori a celle di memoria, il che implica che il concetto di "variabile" è un'astrazione di cella di memoria.

I linguaggi assembler, fortran, Pascal e C, ad esempio, sono basati su distinti livelli di astrazione di questa architettura.

I linguaggi imperativi adottano uno stile prescrittivo:

- Un programma scritto in un linguaggio imperativo prescrive le operazioni che il processore deve seguire per modificare lo stato del sistema (ad esempio l'assegnamento)
- Esecuzione delle istruzioni nell'ordine in cui appaiono nel programma ad eccezione delle strutture di controllo (for, while, ecc.)

Sono realizzati sia attraverso l'interpretazione, sia mediante la compilazione.

PROGRAMMA = ALGORITMI + STRUTTURE DATI

La struttura del programma consiste in:

- Una parte di dichiarazione in cui si dichiarano tutte le variabili del programma e il loro tipo
- Una parte che descrive l'algoritmo risolutivo utilizzato, mediante istruzioni del linguaggio

Le istruzioni si dividono in:

- Istruzioni di lettura e scrittura
- Istruzioni di assegnamento
- Istruzioni di controllo

Sono nati per la necessità di gestire applicazioni a più alto livello di astrazioni e nel tentativo di sviluppare programmi più concisi, semplici da scrivere, più vicini alla logica del problema, la cui correttezza dia più semplice da verificare.

2.2 PARADIGMI FUNZIONALE E LOGICO: CARATTERISTICHE COMUNI

Sono linguaggi ad altissimo livello, utilizzabili, in teoria, anche da non programmatori.

Generati per manipolazione simbolica e non numerica.

Concetti di programma e di struttura dati non nettamente separati a seconda del linguaggio: un programma è specificato per mezzo di una struttura dati.

Basati su concetti matematici e non come astrazioni della macchina di Von Neumann.

I linguaggi funzionali e logici adottano uno stile essenzialmente dichiarativo.

2.2.1 Paradigma logico

- Concetto primitivo: deduzione logica
 - Base: logica formale
 - Obiettivo: formalizzare il ragionamento
- Programmare significa:
 - Descrivere il problema con frasi (formule logiche) del linguaggio
 - Interrogare il sistema, che effettua deduzioni sulla base della “conoscenza rappresentata”

PROGRAMMA = CONOSCENZA + CONTROLLO

Stile dichiarativo: la conoscenza del problema è espressa indipendentemente dal suo utilizzo (COSA non COME).

Alta modularità e flessibilità.

Definire un linguaggio logico significa definire come il programmatore può esprimere la conoscenza e quale tipo di controllo si può utilizzare nel processo di deduzione.

2.2.1.1 Linguaggio Prolog

Un programma Prolog è costituito da:

- Asserzioni incondizionate (fatti):
 - A.
- Asserzioni condizionate (o regole):
 - A :- B, C, D, ..., Z.
A è la conclusione (conseguente)
B, C, D, ..., Z sono le premesse (o antecedenti)
- Una interrogazione ha la forma:
 - :- K, L, M, ..., P.

| Esempio: due individui sono colleghi se lavorano per la stessa ditta | |
|--|----------------|
| collega (X, Y) :- | |
| lavora (X, Z), | REGOLA |
| lavora (X, Z), | |
| diverso (X, Y). | |
| lavora(ciro, ibm). | |
| lavora(ugo, ibm). | |
| lavora(olivia, samsung). | FATTI |
| lavora(ernesto, olivetti). | |
| lavora(enrica, samsung). | |
| :- collega (X, Y). | INTERROGAZIONE |

2.3 UNA COMPARAZIONE TRA LO STILE PRESCRITTIVO E LO STILE DICHIARATIVO

Problema: ordinare una lista

- Stile prescrittivo
 - Controlla prima se la lista L è vuota; se sì dai come risultato la lista vuota. Altrimenti calcola una permutazione L₁ di L se è ordinata; se sì termina dando come risultato L₁, altrimenti calcola un'altra permutazione di L etc..
 - Il programmatore deve specificare la sequenza di istruzioni che servono a generare la sequenza di permutazioni della lista L
- Stile dichiarativo
 - Il risultato dell'ordinamento di una lista vuota è la lista vuota
 - Il risultato dell'ordinamento di una lista L è L₁ se la lista L₁ è ordinata ed è la permutazione di L
 - L'ambiente (Prolog o di theorem proving) si fa carico di generare le possibili permutazioni della lista L secondo un processo di deduzione matematica

2.4 PARADIGMA FUNZIONALE

Concetto primitivo: funzione.

Una funzione è una regola di associazione tra due insiemi, che associa un elemento del primo insieme (dominio) ad un elemento del secondo insieme (codominio).

La definizione di una funzione ne specifica il dominio, il codominio e la regola di associazione.

Dopo averne dato definizione, una funzione può essere applicata a un elemento del dominio (argomento) per restituire l'elemento del codominio ad esso associato (valutazione).

L'unica operazione utilizzata nel paradigma funzionale (puro) è l'applicazione di funzioni.

Il ruolo dell'esecutore di un linguaggio funzionale si esaurisce nel valutare l'applicazione di una funzione (il programma) e produrre un valore.

Nel paradigma funzionale “puro” il valore di una funzione è determinato soltanto dal valore degli argomenti che riceve al momento della sua applicazione, e non dallo stato del sistema rappresentato dall'insieme complessivo dei valori associati a variabili (e/o locazioni di memoria) in quel momento, vi è quindi “assenza di effetti collaterali”, ovvero non vengono modificati altri dati.

Il concetto di variabile utilizzato è quello di “costante” matematica, i cui valori non sono mutabili (nessun assegnamento).

L'essenza della programmazione funzionale consiste nel combinare funzioni mediante composizione e utilizzo del concetto di ricorsione.

PROGRAMMA = COMPOSIZIONE DI FUNZIONI + RICORSIONE

La struttura del programma consiste nella definizione di un insieme di funzioni (mutualmente ricorsive). L'esecuzione del programma consiste nella valutazione dell'applicazione di una funzione principale a un insieme di argomenti.

2.5 LINGUAGGIO LISP (LIST PROCESSING)

Proposto nel 1958 da John McCarthy, il progetto originale era per un linguaggio funzionale puro.

Nel corso degli anni sono stati sviluppati molti ambienti di programmazione Lisp.

| |
|--|
| Esempio: controllare se un elemento (item) appartiene ad un insieme (rappresentato con una lista) |
| (defun member (item list) |
| (cond ((null list) nil) |
| ((equal item (first list)) T) |
| (T (member item (rest list)))) |
| |
| (member 42 (list 12 34 42)) |

Il programma è rappresentato da una struttura fondamentale dati del linguaggio (omoiconicità).

Non c'è assegnamento.

2.6 AMBIENTI “RUN TIME” DI LINGUAGGI LOGICI, FUNZIONALI (E NON)

Per eseguire un programma in un qualsiasi linguaggio il sistema (ovvero il sistema operativo) deve mettere a disposizione un ambiente “run time”, che fornisca almeno due funzionalità

- Mantenimento dello stato della computazione (program counter, limiti di memoria, ecc.)

- Gestione della memoria disponibile (fisica e virtuale)

L'ambiente run time può essere una vera e propria macchina virtuale quale quella di Java o .Net Microsoft.

In particolare, la gestione della memoria avviene usando due aree concettualmente ben distinte con funzioni diverse:

- Lo stack dell'ambiente run time serve per la gestione delle chiamate – anche e soprattutto ricorsive – a procedure (o metodi, o funzioni, o subroutines, ecc.)
- Lo heap dell'ambiente run time serve per la gestione di strutture dati dinamiche

I linguaggi logici e funzionali (ma anche Java) utilizzano pesantemente lo heap dato che forniscono come strutture dati “built in” liste e, spesso, vettori di dimensione variabile.

2.6.1 Stack e valutazione di procedure: richiami

La valutazione di procedure avviene mediante la costruzione (sullo stack di sistema) di activation frames.

I parametri formali di una procedura vengono associati ai valori (nb: si passa tutto per valore; ribadiamo che non esistono effetti collaterali)

Il corpo della procedura viene valutato (ricorsivamente) tenendo conto di questi legami in maniera “statica”, ovvero bisogna tener presente cosa accede con variabili che risultano “Libere” in una sotto-espressione.

Ad ogni sotto-espressione del corpo si sostituisce il valore che essa denota (computa).

Il valore (valori) restituito dalla procedura in un'espressione return o con meccanismi analoghi è il valore del corpo della procedura (che non è altro che una sotto-espressione). Quando il valore finale viene “ritornato” i legami temporanei ai parametri formali spariscono, lo stack di sistema subisce una “pop” e l'activation frame viene rimosso.

Ad esempio: “int doppio(int x) { return 2*x; }”

Definizione del legame tra doppio e la sua definizione. La chiamata “int d = doppio(3);”:

- Estende l'ambiente corrente dove è stata dichiarata la variabile d, con quello locale che contiene i legami tra parametri formali e valori dei parametri attuali, un activation frame viene inserito in cima allo stack di valutazione.
- Valuta il corpo della procedura
- Ripristina l'ambiente di partenza, il risultato della chiamata viene salvato nella variabile d e l'activation frame viene rimosso dalla cima dello stack di valutazione.

Per l'esecuzione di una procedura F, un programma deve eseguire i seguenti sei passi:

- Mettere i parametri in un posto dove la procedura possa recuperarli
- Trasferire il controllo della procedura
- Allocare le risorse (di memorizzazione dei dati) necessarie alla procedura
- Effettuare la computazione della procedura
- Mettere i risultati in un posto accessibile al chiamante
- Restituire il controllo al chiamante

Queste operazioni agiscono sui registri a disposizione e sullo “stack” utilizzato dal runtime (esecutore) del linguaggio.

Lo spazio richiesto per salvare (sullo stack) tutte le informazioni necessarie all'esecuzione di una procedura F ed al ripristino dello stato precedente alla chiamata è quindi costituito da:

- Spazio per i registri da salvare prima della chiamata di una sotto procedura
- Spazio per l'indirizzo di ritorno (nel codice del corpo della procedura)
- Spazio per le variabili, definizioni locali e valori di ritorno
- Spazio per i valori degli argomenti
- Spazio per il riferimento statico (static link)
- Spazio per il riferimento dinamico (dynamic link)
- Altro spazio dipendente dal particolare linguaggio e/o politiche di allocazione del compilatore

| |
|---|
| Return address |
| Registri |
| . |
| . |
| Static link |
| Dynamic link |
| Argomenti |
| . |
| . |
| Variabili/definizioni locali, valori di ritorno |
| . |

Ad esempio:

```
int doppio(x) {
    return 2*x; }

int doppio_42(z) {
    return doppio(z) - 42; }
```

la chiamata "doppio_42(42)" ha il seguente effetto →

static link e dynamic link sono molto importanti perché servono a mantenere informazioni circa il:

- Dove una procedura è definita
- Quando una procedura è chiamata

Il contenuto effettivo di un activation frame dipende da diverse scelte implementative. L'esempio riportato non è necessariamente completo.

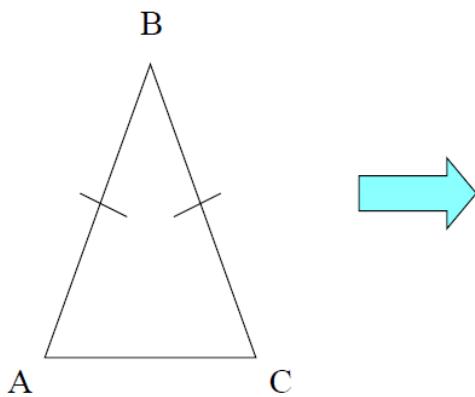
| Global frame | |
|-----------------------|------------------------|
| Return address: | 0 |
| Registri | ... |
| Static link | 0 |
| Dynamic link | 0 |
| Argomenti | ... |
| Local definitions, RV | doppio doppio_42 42 |
| Doppio_42 | |
| Return address: | #x000.... |
| Registri | ... |
| Static link | |
| Dynamic link | |
| Argomenti | z : 42 |
| Local definitions, RV | 84 |
| doppio | |
| Return address: | #x000.... |
| Registri | ... |
| Static link | |
| Dynamic link | |
| Argomenti | x : 42 |
| Local definitions, RV | |

2.6.2 Heap e Garbage collection

L'area di memoria per la manipolazione di strutture dati dinamiche verrà spiegata meglio al momento dell'introduzione delle primitive per la costruzione delle liste in Prolog e Lisp.

3 INTRODUZIONE ALLA LOGICA

Mostriamo come si dimostra un semplice teorema di geometria:



Dato un triangolo isoscele ovvero con $AB = BC$, si vuole dimostrare che gli angoli $\angle A$ e $\angle C$ sono uguali

conoscenze pregresse:

- 1) se due triangoli sono uguali, i due triangoli hanno lati ed angoli uguali
- 2) se due triangoli hanno due lati e l'angolo sotteso uguali, allora i due triangoli sono uguali
- 3) BH bisettrice di $\angle B$ ¹ cioè $\angle ABH = \angle HBC$

Dimostrazione:

- $AB = BC$ per ipotesi
- $\angle ABH = \angle HBC$ PER (3)
- Il triangolo ΔHBC è uguale al triangolo ΔABH per (2)
- $\angle A$ e $\angle C$ per (1)

Abbiamo trasformato (2) in

→ se $AB = BC$ e $BH = BH$ e $\angle ABH = \angle HBC$, allora il triangolo ΔABH è uguale al triangolo ΔHBC

Ed abbiamo trasformato (1) in

→ se triangolo ABH è uguale al triangolo HBC , allora $AB = BC$ e $BH = BH$ e $AH = HC$ e $\angle ABH = \angle HBC$ e $\angle AHB = \angle CHB$ e $\angle A = \angle C$

Obiettivo: razionalizzare il processo che permette di affermare $AB = BC \vdash \angle A = \angle C$ dove il simbolo \vdash simboleggia la derivazione logica, significa "consegue", "segue che", "allora", ecc.

Quindi:

Conoscenze pregresse

- se $AB = BC$ e $BH = BH$ e $\angle ABH = \angle HBC \rightarrow \Delta ABH = \Delta HBC$
- $\Delta ABH = \Delta HBC \rightarrow AB = BC$ e $BH = BH$ e $AH = HC$ e $\angle ABH = \angle HBC$ e $\angle AHB = \angle CHB$ e $\angle A = \angle C$

Assunti (P)

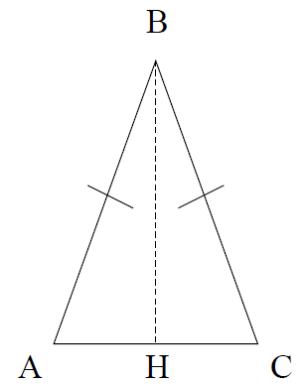
$P = \{AB = BC, \angle ABH = \angle HBC, BH = BH\}$

Tesi

$AB = BC \vdash \angle A = \angle C$

¹ \angle è 2220 + ALT + x

| Dimostrazione | |
|--|--|
| P1: AB = BC | Da P |
| P2: $\angle AHB = \angle HBC$ | Da P |
| P3: BH = BH | Da P |
| P4: AB = BC \wedge BH = BH \wedge $\angle AHB = \angle HBC$ ² | Da P1, P2, P3 e introduzione della congiunzione |
| P5: $\triangle AHB = \triangle HBC$ | Da P4, regola1 e modus ponens |
| P6: AB = BC \wedge BH = BH \wedge AH = HC \wedge $\angle AHB = \angle HBC \wedge \angle AHB = \angle CHB \wedge \angle A = \angle C$ | Da P5, regola2 e modus ponens |
| P7: $\angle A = \angle C$ | Da P6 e l' eliminazione della congiunzione |



Una dimostrazione dimostra che $S \vdash F$ ³, ovvero F è conseguenza di S

È una sequenza $DIM = \langle P_1, P_2, \dots, P_n \rangle$

Dove

- $P_n = F$
- $P_i \in S$
- Oppure P_i è ottenibile da P_{i1}, \dots, P_{im} (con $i_1 < i, \dots, i_m < i$) applicando una regola d'inferenza

3.1.1 Regole di inferenza e calcoli logici

Un insieme di regole di inferenza costituisce la base di un calcolo logico, diversi insiemi di regole danno vita a diversi calcoli logici.

Lo scopo di un calcolo logico è manipolare delle formule logiche in modo completamente sintattico al fine di stabilire una connessione tra un insieme di formule di partenza (di solito un insieme di formule dette assiomi) ed un insieme di conclusioni.

Nel seguito presenteremo due tipi di logica (due linguaggi logici) con il calcolo logico ad essere associato:

- Logica proposizionale (logica delle proposizioni)
- Logica dei predicati del primo ordine

3.2 LOGICA PROPOZIONALE

La logica proposizionale si occupa delle conclusioni che possiamo trarre da un insieme di proposizioni.

Una logica proposizionale è sintatticamente definita da un insieme P di proposizioni. All'insieme P è associata una funzione di verità, o di valutazione, V (spesso indicata con T o con I) per la quale " $V: P \rightarrow \{\text{vero, falso}\}$ " che associa un valore di verità ad ogni elemento di P, cioè ad ogni proposizione.

La funzione di valutazione è il ponte di connessione tra la sintassi e la semantica di un linguaggio logico.

² \wedge è 2227 + alt + x

³ \vdash è 22A2 + alt + x

Le proposizioni in una logica proposizionale possono essere combinate utilizzando una serie di connettivi logici:

- Congiunzione \wedge
- Disgiunzione \vee
- Negazione \neg
- Implicazione \rightarrow

Chiamiamo FBF l'insieme di tutte le formule formate dagli elementi di P e dalle loro combinazioni (formule ben formate).

Le formule atomiche in P e le loro negazioni vengono anche chiamati letterali (positivi o negativi).

Il valore di verità di una proposizione dipende dalla funzione di verità V. Il valore di verità di una formula composta dipende dal valore di verità delle sue componenti. La definizione della funzione di valutazione V viene quindi estesa sul dominio FBF.

La funzione V associa un valore di verità ad un elemento di FBF secondo le regole seguenti:

- $V(\neg s) = \text{non } V(s)$
- $V(a \wedge b) = V(a) \wedge V(b)$
- $V(a \vee b) = V(a) \vee V(b)$
- $V(p \rightarrow q) = (\text{non } V(p)) \vee V(q)$

Un metodo per calcolare il valore di verità di una proposizione composta è quello di utilizzare la tavola di verità.

Mentre la funzione di verità costituisce la parte semantica di un insieme di proposizioni, ovvero dice ciò che è vero e ciò che è falso sotto l'interpretazione considerata, un calcolo logico dice come generare nuove formule (cioè espressioni sintattiche) a partire da un insieme di partenza (gli assiomi).

Un calcolo deve garantire che tutte le nuove formule generate siano "vere" se l'insieme di assiomi consiste solo di formule "vere". Il processo di generazione si chiama dimostrazione.

3.2.1 Calcolo proposizionale: regole di inferenza

Il calcolo proposizionale è basato su una serie di regole di inferenza che ci permettono di ottenere delle nuove formule a partire da un insieme di assiomi. Una regola di inferenza ha la seguente forma generale:

| | |
|------------------------|---------------|
| F_1, F_2, \dots, F_k | [nome regola] |
| R | |

Dove ogni F_i rappresenta una formula (vera) in FBF e R è la formula generata da "inserire" in FBF, il "nome regola" ci dice che regola di inferenza stiamo usando. L'esempio tipico di regola di inferenza è il cosiddetto modus ponens.

3.2.1.1 Modus ponens

Ad esempio:

| |
|-------------------|
| $p \rightarrow q$ |
| p |
| [Modus ponens] |

- Se piove la strada è bagnata
- Piove
- Allora la strada è bagnata

Ovvero la regola sintattica del modus ponens ci permette di aggiungere le conclusioni di una regola al nostro insieme di FBF "vere"

3.2.1.2 *Modus tollens*

Ad esempio:

$$\frac{p \rightarrow q}{\neg p} \quad [\text{Modus tollens}]$$

$$\neg q$$

- Se piove, la strada è bagnata
- La strada non è bagnata
- Allora non piove

Ovvero la regola del modus tollens ci permette di aggiungere la premessa negata di una regola al nostro insieme di FBF “vere”

3.2.1.3 *Eliminazione e introduzione di “e”*

$$\frac{p_1 \wedge p_2 \wedge \dots \wedge p_i}{p_i} \quad [\text{eliminazione } \wedge]$$

La regola sintattica dell’eliminazione della congiunzione ci permette di aggiungere all’insieme FBF i singoli componenti di una formula complessa.

$$\frac{p_1, p_2, \dots, p_i}{p_1 \wedge p_2 \wedge \dots \wedge p_i} \quad [\text{introduzione } \wedge]$$

Queste regole si chiamano anche “di semplificazione” e “di congiunzione”.

3.2.1.4 *Introduzione di “o”*

La regola sintattica dell’introduzione della disgiunzione ci permette di aggiungere i singoli componenti di una formula complessa.

$$\frac{p}{p \vee q} \quad [\text{introduzione } \vee]$$

Questa regola è detta anche “di addizione”.

3.2.1.5 *Varie ed eventuali*

$$\frac{p \vee \neg p}{\text{vero}} \quad [\text{terzo escluso}]$$

$$\frac{p \wedge \text{vero}}{p} \quad [\text{eliminazione } \wedge]$$

La regola della contraddizione esprime che da una contraddizione si possa derivare qualsiasi conseguenza

$$\frac{p \wedge \neg p}{q} \quad [\text{contraddizione}]$$

$$\frac{\neg\neg p}{p} \quad [\text{eliminazione } \neg]$$

3.2.2 Regole di inferenza

Le regole di inferenza che abbiamo visto e anche altre non citate fanno parte del cosiddetto “calcolo naturale” o di “Gentzen”, nome del loro creatore. Il calcolo di Gentzen, e molte varianti, formalizzano i modi di derivare delle conclusioni a partire da un insieme di premesse, in particolare permettono di derivare “direttamente” una FBF mediante una sequenza di passi ben codificati.

La regola del modus ponens assieme al principio del terzo escluso, possono però essere usati in un altro modo, procedendo “per assurdo” alla dimostrazione di una data formula. Questa vista “estesa” della regola del modus ponens è conosciuta come “Principio di risoluzione”.

3.2.2.1 *Il principio di risoluzione*

Il principio di risoluzione è una regola di inferenza generalizzata semplice e facile da utilizzare e implementare:

- Opera su FBF trasformate in forma normale congiunta

- Ognuno dei congiunti di queste formule viene detto “clausola”

L’osservazione fondamentale alla base del principio di risoluzione è un’estensione della nozione di rimozione dell’implicazione sulla base del principio di contraddizione, solitamente è usata per fare dimostrazioni per assurdo.

| | |
|-----------------|---------------------|
| $p \vee \neg r$ | |
| $s \vee r$ | Clausola risolvente |
| $p \vee s$ | |

| | |
|----------|----------------|
| $\neg r$ | |
| r | contraddizione |
| \perp | ⁴ |

Si noti che la generazione della clausola vuota corrisponde all’aver dimostrato che il mio insieme di FBF contiene una contraddizione.

3.2.2.2 Unit resolution

| | |
|---|---------------------|
| $\neg p$ | |
| $q_1 \vee q_2 \vee \dots \vee q_k \vee p$ | Clausola risolvente |
| $q_1 \vee q_2 \vee \dots \vee q_k$ | |

| | |
|--|---------------------|
| p | |
| $q_1 \vee q_2 \vee \dots \vee q_k \vee \neg p$ | Clausola risolvente |
| $q_1 \vee q_2 \vee \dots \vee q_k$ | |

La regola di risoluzione è molto generale.

Quando una delle due clausole da risolvere è un letterale (ovvero una proposizione o, come vedremo, un predicato) anche negato, come nel caso di p nei due esempi qui sopra, allora si parla di “unit resolution”.

3.2.3 Dimostrazioni per assurdo

Supponiamo di avere a disposizione un insieme di FBF vere, data una certa interpretazione V . Supponiamo di voler dimostrare che una certa proposizione p (o formula atomica) è vera. Possiamo procedere usando il metodo della “reductio ad absurdum” (dimostrazione per assurdo):

- Assumiamo che $\neg p$ sia vera
- Se, combinandola con le proposizioni in FBF ottengo una contraddizione, allora concludo che p deve essere vera

3.2.4 Assiomi (conoscenze pregresse)

Proposizioni sempre vere:

- **A1:** $A \rightarrow (B \rightarrow A)$
- **A2:** $(A \rightarrow (B \rightarrow C)) \rightarrow ((A \rightarrow B) \rightarrow (A \rightarrow C))$
- **A3:** $(\neg B \rightarrow \neg A) \rightarrow ((\neg B \rightarrow A) \rightarrow B)$
- **A4:** $\neg(A \wedge \neg A)$ (principio di non contraddizione)
- **A5:** $A \vee \neg A$ (principio del terzo escluso)

⁴ \perp è il simbolo di clausola vuota

3.2.5 Esempio

"se l'unicorno è mitico, allora è immortale, ma se non è mitico allora è mortale. Se è mortale o immortale, allora è cornuto. L'unicorno è magico se è cornuto"

Domande:

- L'unicorno è mitico?
- L'unicorno è magico?
- L'unicorno è cornuto?

Prima dobbiamo identificare le proposizioni:

- UM = unicorno è magico
- UI = unicorno è immortale
- Umag = unicorno è magico
- UC = unicorno è cornuto

Trascrizione del testo:

- $UM \rightarrow UI$
- $\neg UM \rightarrow \neg UI$
- $\neg UI \vee UI \rightarrow UC$
- $UC \rightarrow Umag$

Quindi la rappresentazione totale è:

- $S^5 = \{UM \rightarrow UI, \neg UM \rightarrow \neg UI, \neg UI \vee UI \rightarrow UC, UC \rightarrow Umag\}$
- $S \vdash UM?$
- $S \vdash Umag?$
- $S \vdash UC?$

$S \vdash UM$ non è derivabile in quanto non abbiamo conoscenze pregresse che ci consentano di valutare l'espressione.

$S \vdash UC$

| | | |
|------------|-----------------------------|---|
| P1: | $\neg UIVUI \rightarrow UC$ | Da S |
| P2: | $\neg UIVUI$ | Da A5 (oppure da assiomi, oppure da conoscenze pregresse) |
| P3: | UC | Da P1, P2, modus ponens |

$S \vdash Umag$

| | | |
|------------|-----------------------------|---|
| P1: | $\neg UIVUI \rightarrow UC$ | Da S |
| P2: | $\neg UIVUI$ | Da A5 (oppure da assiomi, oppure da conoscenze pregresse) |
| P3: | UC | Da P1, P2, modus ponens |
| P4: | $UC \rightarrow Umag$ | Da S |
| P5: | Umag | Da P3, P4 e modus ponens |

In alternativa si sarebbe potuto includere UC in S dopo averlo dimostrato e quindi la seconda dimostrazione sarebbe stata solo:

⁵ S è l'insieme delle supposizioni e delle conoscenze pregresse, include di base gli assiomi

S \vdash Umag

| | | |
|------------|-----------------------|--------------------------|
| P1: | UC | Da S |
| P2: | UC \rightarrow Umag | Da S |
| P3: | Umag | Da P3, P4 e modus ponens |

3.2.6 Sintassi e semantica

Come abbiamo già accennato esiste una differenza tra sintassi e semantica in logica

Un argomento a sostegno dell'esistenza di questa differenza consiste nel considerare le seguenti "stringhe" di caratteri" (2A, 42, XLII, 33, 101010). Esse sono la rappresentazione (in "linguaggi" diversi dello stesso oggetto).

Un calcolo logico fornisce una manipolazione sintattica (simbolica), l'operatore di derivazione \vdash è un operatore sintattico.

La semantica di un insieme di formule dipende dalla funzione di valutazione V, simmetricamente viene introdotto l'operatore di conseguenza logica (in inglese entailment), denotato da \vDash ⁶.

In particolare, data una particolare logica: $S \vdash f$ se e solo se $S \vDash f$

Dove S è un insieme di formule iniziale ed f è una FBF, il tutto in dipendenza da una particolare funzione di verità V.

| Derivazione | Conseguenza logica |
|-------------|--------------------|
| \vdash | \vDash |

3.2.7 Tautologie e modelli

Una FBF sempre vera indipendentemente dal valore dei letterali viene detta tautologia. Una particolare interpretazione V che rende vere tutte le formule in S viene detta "modello di S".

4 LOGICA DEL PRIMO ORDINE

4.1 LOGICA PROPOZIONALE VS. LOGICA DEL PRIMO ORDINE

La logica proposizionale è molto utile:

- Ha caratteristiche computazionali chiare
- Ha una semantica altrettanto chiara

La logica proposizionale ha numerose limitazioni:

- Non ci permette di fare asserzioni circa insiemi di elementi in maniera concisa, ad esempio la frase "tutti gli uomini sono mortali" non è esprimibile in logica proposizionale

Per risolvere questi problemi, la logica del primo ordine (LPO) introduce le nozioni di:

- Variabile
- Costante
- Relazione (o predicato): sottoinsieme di prodotto cartesiano, quantificata attraverso valori V o F
- Funzione: restituisce un valore

⁶ \vDash è 22A8 + alt + X

- Quantificatore

Le query possono ricevere solo valori di verità, quindi una funzione non può essere l'operatore più esterno di una query.

Ovvero, un linguaggio logico del primo ordine è costituito da termini costituiti a partire da:

- V: insieme di simboli di variabili
- C: insieme di simboli di costante
- R: insieme di simboli di relazione o predicati di varia arietà
- F: insieme di simboli di funzione di varia arietà
- Connettivi logici (già visti) e simboli di quantificazione \forall (universale) e \exists (esistenziale)

4.2 SINTASSI DELLA LOGICA DEL PRIMO ORDINE

La query è l'unica variabile quantificata esistenzialmente, tutte le altre vengono quantificate universalmente in maniera implicita.

La costruzione di un linguaggio logico del primo ordine è ricorsiva.

I termini più semplici sono i predicati $r \subseteq C_0 \times C_1 \times \dots \times C_k$ ovvero relazioni cartesiane su C, scritte come $r(c_1, c_2, \dots, c_k)$.

Le funzioni sono definite con il seguente dominio e codominio $f: C_0 \times C_1 \times \dots \times C_m \rightarrow C$ una funzione si scrive come $f(c_1, c_2, \dots, c_m)$.

Le FBF di un linguaggio logico sono costruite ricorsivamente nel seguente modo:

- Un termine t_j può essere un elemento C, di V, oppure un'applicazione di funzione $f(t_1, t_2, \dots, t_s)$
- Un termine costituito da un predicato $r(t_1, t_2, \dots, t_k)$ dove ogni t_j è un termine, appartiene ad FBF
- Diversi elementi di FBF connessi dai connettivi logici standard (congiunzione, disgiunzione, negazione, implicazione) appartengono ad FBF
 - Denotiamo con $t(t_1, t_2, \dots, t_r)$ tale combinazione di termini,
- Le formule $\forall x.t(t_1, t_2, \dots, x, \dots, t_r)$ e $\exists x.t(t_1, t_2, \dots, x, \dots, t_r)$ appartengono a FBF

Con le definizioni precedenti, possiamo rivedere l'esempio socratico:

- Socrate è un uomo
- Tutti gli uomini sono mortali
- Allora Socrate è mortale

Costanti individuali:

- $C = \{\text{Parmenide, Socrate, Platone, Aristotele, Crisippo, Pino, Gino, Rino, Ugo}\}$

Predicati:

- $R = \{\text{uomo, mortale}\}$

Rappresentiamo le frasi:

| Asserzioni principali | Frasi | Enunciati |
|-----------------------|-------------------------------|--|
| | tutti gli uomini sono mortali | $\forall x.(\text{uomo}(x) \Rightarrow \text{mortale}(x))$ |
| | Socrate è un uomo | $\text{Uomo}(\text{Socrate})$ |

| Conclusione alla quale vogliamo arrivare | Allora Socrate è mortale | Mortale(Socrate) |
|--|--------------------------|------------------|
|--|--------------------------|------------------|

Naturalmente non possiamo semplicemente dire che la nostra conclusione “segue” dalle asserzioni iniziali. Dobbiamo giustificare questa conclusione sulla base della sintassi e della semantica del nostro linguaggio logico (del primo ordine). In altre parole, dobbiamo usare di nuovo delle regole di calcolo per ottenere la conclusione. Dato che il linguaggio è più ricco, dobbiamo costruire delle regole più sofisticate.

| | | |
|---|------------------------|--|
| $\forall x.t(..., x, ...)$ $c \in C$ | Eliminazione \forall | Con la regola appena introdotta possiamo derivare la nostra conclusione a partire dalle asserzioni iniziali. |
|---|------------------------|--|

| | | |
|--|------------------------|--|
| $(\forall x.(uomo(x) \Rightarrow mortale(x))$ $Socrate \in C$ | Eliminazione \forall | |
| $uomo(Socrate) \Rightarrow mortale(Socrate)$ | | |

| | | |
|---|----------------------------|--|
| $uomo(Socrate)$ $uomo(Socrate) \Rightarrow mortale(Socrate)$ | Eliminazione \Rightarrow | |
|---|----------------------------|--|

4.3 ALTRE REGOLE NEI LINGUAGGI DEL PRIMO ORDINE

Date le regole per l'eliminazione del quantificatore universale, dobbiamo esibire delle regole per la manipolazione del quantificatore esistenziale.

| | | |
|-------------------------------|------------------------|--|
| $T(..., c, ...)$ $c \in C$ | Introduzione \exists | |
|-------------------------------|------------------------|--|

Valgono anche le seguenti identità riguardanti l'interazione tra quantificatori e negazione:

- $\exists x. \neg T(..., x, ...) \equiv \neg \forall x.t(..., x, ...)$
- $\forall x. \neg T(..., x, ...) \equiv \neg \exists x.t(..., x, ...)$

5 PROLOG

5.1 PROGRAMMAZIONE LOGICA

La programmazione logica nasce all'inizio degli anni settanta da studi sulla deduzione automatica: il Prolog costituisce uno dei risultati principali.

La programmazione logica non è soltanto rappresentata dal Prolog: essa costituisce infatti un settore molto ricco che cerca di utilizzare la logica matematica come base dei linguaggi di programmazione. Prolog è solo l'esempio più noto.

Obiettivi del linguaggio:

- Semplicità del formalismo
- Linguaggio ad alto livello
- Semantica chiara

5.2 PROGRAMMAZIONE LOGICA E LA LOGICA MATEMATICA

Logica matematica: formalizzazione del ragionamento

Con l'avvento dell'informatica:

- Logica matematica → dimostrazione automatica di teoremi (procedura di Davis e Putnam e principio di risoluzione)
- Interpretazione procedurale di formule (Kowalski)
 - Logica come linguaggio di programmazione

Utilizzata in varie applicazioni: dalle prove di correttezza dei programmi alle specifiche, da linguaggio per la rappresentazione della conoscenza in intelligenza artificiale a formalismo per database.

5.3 STILE DICHIARATIVO DELLA PROGRAMMAZIONE LOGICA

Programma come insieme di formule con un grande potere espressivo. Computazione come costruzione cdi una dimostrazione di un'affermazione (goal).

Base formale:

- Calcolo dei predicati del primo ordine ma con limitazione nel tipo di formule (clausole di Horn)
- Utilizzo di particolari tecniche per la dimostrazione di teoremi (meccanismo di Risoluzione)

5.4 PROLOG

Acronimo per PROgramming in LOGic, nato nel 1973. Basato su una restrizione della logica del primo ordine. Utilizza uno stile dichiarativo.

Prolog è usato per determinare se una certa affermazione è vera o no e, se è vera, quali vincoli sui valori attribuibili alle variabili hanno generato la risposta.

5.5 FORMULE BEN FORMATE (FBF) E FORMA NORMALE "A CLAUSOLE"

Ogni FBF di un linguaggio logico del primo ordine può essere riscritta in forma normale a clausole.

Vi sono due forme normali a clausole:

- **Forma normale congiunta (Conjunctive Normal Form – CNF)**

- La formula è una coconiunzione di disgiunzioni di predicati o di negazioni di predicati (letterali positivi e letterali negativi)

$$\bigwedge_i \left(\bigvee_j L_{ij} \right)$$

- **Forma normale disgiunta (Disjunctive Normal Form – DNF)**

- La formula è una disgiunzione di congiunzioni di predicati o di negazioni di predicati (letterali positivi e letterali negativi)

$$\bigvee_j \left(\bigwedge_i L_{ij} \right)$$

Dove $L_{ij} \equiv^7 P_{ij}(x, y, \dots, z)$ o $L_{ij} \equiv \neg Q_{ij}(x, y, \dots, z)$.

5.5.1 Forma normale congiunta (Conjunctive Normal Form – CNF)

Consideriamo una FBF in CNF:

$$\bigwedge_i \left(\underbrace{\bigvee_j L_{ij}}_{\text{clausole}} \right)$$

Ad esempio:

$$\begin{aligned} & \underbrace{(p(x) \vee q(x, y) \vee \neg t(z))}_{\text{clausola 1}} \wedge \underbrace{(p(w) \vee \neg s(u) \vee \neg r(v))}_{\text{clausola 2}} \\ & \underbrace{(\neg t(z))}_{\text{clausola 1}} \wedge \underbrace{(p(w) \vee \neg s(u))}_{\text{clausola 2}} \wedge \underbrace{(p(x) \vee s(x) \vee q(y))}_{\text{clausola 3}} \end{aligned}$$

Se scartiamo il simbolo di congiunzione, rimaniamo con solo le clausole disgiuntive. Utilizzando il primo esempio:

$$p(x) \vee q(x, y) \vee \neg t(z)$$

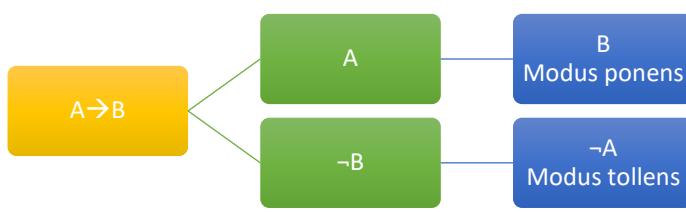
$$p(w) \vee \neg s(u) \vee \neg r(v)$$

le clausole relative al primo esempio sono anche riscrivibili come:

$$t(z) \rightarrow p(x) \vee q(x, y)$$

$$r(v) \wedge s(u) \rightarrow p(w)$$

⁷ \equiv è 2261 + alt + X



ovvero un insieme di formule in CNF è riscrivibile come un insieme (congiunzione) di implicazioni, questo perché da un'implicazione si può dedurre o il conseguente(modus ponens) o la negazione dell'antecedente (modus tollens).

Le clausole che hanno al più un solo letterale positivo (con o senza letterali negativi) si chiamano “clausole di Horn”.

- Non tutte le FBF possono essere trasformate in un insieme di clausole di Horn
- I programmi Prolog sono collezioni di clausole di Horn

5.6 PROLOG – LINGUAGGIO DICHIARATIVO

Non contiene (quasi) istruzioni, contiene solo fatti e regole (clausole di Horn):

- **Fatti:** asserzioni vere nel contesto che stiamo descrivendo
- **Regole:** ci danno gli strumenti per dedurre nuovi fatti da quelli esistenti

Un programma Prolog ci dà informazioni su un sistema ed è normalmente chiamato “base di conoscenza”.

Un programma Prolog non si “esegue”, si “interroga”, ci si chiede se i fatti descritti siano veri e il programma potrà rispondere solo “sì” o “no”.

Sintassi: un programma Prolog è costituito da un insieme di clausole della forma:

| | | |
|---|--------------------|--|
| <i>a.</i> | Fatto o asserzione | In cui a, bi, c e qj sono termini (composti). |
| <i>c :- b₁, b₂, ..., b_n.</i> | Regola | |
| ?- <i>q₁, q₂, ..., q_m.</i> | Goal o query | Notare che in molte implementazioni il prompt Prolog è anche un operatore che chiede al sistema di valutare il goal, in questo caso una congiunzione di termini. |

5.6.1 Termini

Le espressioni del Prolog sono chiamate termini. Esistono diversi tipi di termini:

- Atomi
- Variabili
- Una composizione di termini → termine composto (simbolo di funtore più uno o più argomenti)

5.6.1.1 Atomi

Un atomo è:

- Una sequenza di caratteri alfanumerici, che inizia con un carattere minuscolo (può contenere il carattere “_” underscore)
- Qualsiasi cosa racchiusa tra apici (‘ ’)
- Un numero

5.6.1.2 Variabili logiche

Una variabile (logica) è una sequenza alfanumerica che inizia con un carattere maiuscolo o con il carattere _ (underscore).

Le variabili (notare il plurale) composte solo dal simbolo _ sono chiamate “indifferenti” (don’t care) o anonime (any).

Le variabili vengono istanziate con il procedere del programma.

5.6.1.3 Termini composti

Una composizione di termini consiste in:

- Un funtore (simbolo di funzione o di predicato definito come atomo)
- Una sequenza di termini racchiusi tra parentesi tonde e separati da virgole. Questi sono chiamati argomenti del funtore. Non ci deve mai essere uno spazio tra il funtore e la parentesi di sinistra; questo per via di caratteristiche molto sofisticate del sistema di parsing di Prolog.

Esempi di termini

| Validi | Non validi |
|--|------------------------|
| <code>foo</code> | <code>hello Sam</code> |
| <code>Hello</code> | <code>Hello sam</code> |
| <code>hello_Sam</code> | <code>f(a, b</code> |
| <code>40+2</code> | <code>f a, b)</code> |
| <code>'hello'</code> | <code>f (a, b)</code> |
| <code>'Hello Sam'</code> | <code>X(a, b)</code> |
| <code>a1</code> | <code>1(a, b)</code> |
| <code>X</code> | |
| <code>_234</code> | |
| <code>f(a)</code> | |
| <code>f(hello, Sam)</code> | |
| <code>p(f(a), b)</code> | |
| <code>hello(1, hello(x, X, hello(sam)))</code> | |
| <code>t(a, t(b, t(c, t(d, []))))</code> | |

5.6.2 Fatti o predicati

Esempi

```
libro(kowalski,
prolog).
libro(yoshimoto,
kitchen).
donna(yoshimoto).
uomo(kowalski).
animale(cane).
animale(trota).
ha_le_squame(trota).
la_risposta(42).
```

Un fatto (predicato) consiste in:

- Un nome di predicato, ad esempio *fattoriale*, *genitore*, *uomo* o *animale*. Deve iniziare con una lettera minuscola.
- Zero o più argomenti come *maria*, *42* o *cane*
- Da notare che i fatti (e le regole e le domande) devono essere terminati da un punto.

5.6.3 Le regole

In Prolog si usano le regole quando si vuole esprimere che un certo fatto dipende da un insieme di altri fatti.

Per esprimere in linguaggio naturale questa dipendenza usiamo la parola “se” (usa l’ombrellino se piove).

Le regole sono anche usate per esprimere definizioni:

- X è un pesce se:
 - X è un animale
 - X ha le squame

In prolog una regola consiste di una testa (head) e di un corpo (body).

- Testa e corpo sono collegati dall'operatore :-
- La testa di una regola corrisponde al conseguente di un'implicazione logica
- Il corpo di una regola corrisponde all'antecedente di un'implicazione logica
- Le regole Prolog corrispondono alle clausole di Horn, ovvero hanno un solo termine (predicato) come conseguente
 - L'operatore Prolog :- esprime il "se" (implicazione)
 - L'operatore Prolog , equivale a "e" (and, o congiunzione)

Esempi

| | |
|---|--|
| Un pesce è un animale che ha le squame | <i>pesce (X) :- animale (X), ha_le_squame (X).</i> |
| Gigi ama chiunque ami il vino | <i>ama (gigi, X) :- ama (X, vino).</i> |
| Gigi ama le donne a cui piace il vino | <i>ama (gigi, X) :- donna (X), ama (X, vino).</i> |

5.6.4 Relazioni definite da più regole

Una relazione (ad esempio genitore) può essere definita da più regole (ovvero da più clausole) aventi lo stesso predicato come conclusione.

Esempi

```
genitore (X, Y) :-  
padre (X, Y).  
genitore (X, Y) :-  
madre (X, Y).
```

Le regole (ed i fatti) sono implicitamente connesse dall'operatore logico di congiunzione ("and"); se non si sono commessi errori – per l'appunto – logici, entrambe le implicazione qua sopra sono da ritenersi vere.

5.6.5 Ricorsione

Una relazione può anche essere definita ricorsivamente. In questo caso la definizione richiede almeno due proposizioni: una è quella ricorsiva che corrisponde al caso generale, l'altra esprime il caso particolare più semplice.

Esempi

| | |
|---|---|
| <i>antenato (X, Y) :- genitore (X, Y).</i> | Caso base: ha trovato il genitore |
| <i>antenato (X, Y) :- genitore (Z, Y), antenato (X, Z).</i> | Caso ricorsivo: cerca ricorsivamente i genitori dei genitori |

5.6.6 Operatori logici

AND: si usa la virgola ,

OR: si usa un punto e virgola ;

5.6.7 Sintassi

Ogni fatto o regola o funzione deve terminare con un punto

Ogni variabile deve iniziare con una maiuscola

I commenti si inseriscono dopo un "%" (commento di linea) o tra "/*" e "*/"

5.6.8 Interrogazioni (queries o goals)

Una volta che le regole ed i fatti sono scritte e caricate nell'interprete, eseguire un programma Prolog significa interrogare l'interprete. Una volta fatto partire, di solito, l'interprete Prolog ci presenta il prompt

?-

Un esempio di query diventa:

?- *libro (kowalski, prolog).*

Dati i fatti e le regole che abbiamo visto precedentemente, il prolog risponde "yes".

5.6.9 Esempio di programma Prolog

Fatti (DB)

- *padre(giovanni, maria).*
- *padre(carlo, giulio).*
- *madre(maria, ettore).*

Interrogazioni

- *?- padre(giovanni, maria).*
 - Yes
- *?- padre(giovanni, carlo)*
 - No

5.6.10 Le variabili delle interrogazioni

Esempi

```
?- libro(kowalski, LINGUAGGIO).
Yes
      LINGUAGGIO = prolog
?- libro(AUTORE, prolog).
Yes
      AUTORE = kowalski
?- libro(AUTORE, LINGUAGGIO).
Yes
      AUTORE = kowalski
      LINGUAGGIO = prolog
```

Le interrogazioni possono contenere variabili interpretate come variabili esistenziali.

Le variabili sono istanziate quando il Prolog prova a rispondere alla domanda.

Tutte le variabili istanziante vengono mostrate nella risposta.

5.6.11 Unificazione

L'operazione di istanziazione di variabili durante la "prova" di un predicato è il risultato di una procedura particolare, detta unificazione.

Esempi

| | |
|--|-----------------------|
| <i>Mgu(42, 42)</i> | → {} |
| <i>Mgu(42, X)</i> | → {X/42} |
| <i>Mgu(X, 42)</i> | → {X/42} |
| <i>Mgu(foo(bar, 42), foo(bar, X))</i> | → {X/42} |
| <i>Mgu(foo(Y, 42), foo(bar, X))</i> | → {Y/bar, X/42} |
| <i>Mgu(foo(bar(42), baz), foo(X, Y))</i> | → {X/bar(42), Y/baz} |
| <i>Mgu(foo(X), foo(bar(Y)))</i> | → {X/bar(Y), Y/_G001} |

Dati due termini, la procedura di unificazione crea un insieme di sostituzione delle variabili. Questo insieme di sostituzione (brevemente: sostituzione o

assegnamento) permette di rendere "uguali" i due termini.

Esempi

```
?- 42 = 42.
Yes
?- 42 = X.
      X = 42
      Yes
?- foo(bar, 42) = foo(bar, X).
      X = 42
      Yes
?- foo(42, bar(X), trillian) = foo(Y, bar(Y), X).
      No
```

Tradizionalmente la procedura di unificazione costruisce un insieme di sostituzioni chiamato "most general unifier" ed indicato con Mgu. Una sostituzione è indicata come una sequenza ordinata di coppie variabile/valore.

Notare l'ultimo esempio con ridenominazione di variabili. Il "most general unifier" non è nient'altro che il risultato finale della procedura di valutazione – ovvero di prova – del Prolog.

Il modo più semplice per vedere come la procedura di unificazione funziona è di usare l'operatore Prolog =.

5.6.12 Diverse rappresentazioni di dati ed interrogazioni

Consideriamo il seguente esempio: vogliamo descrivere un insieme di fatti riguardanti i corsi offerti dal dipartimento.

Possibilità 1

Tutte le informazioni concentrate in una relazione con 6 campi.

```
corso(linguaggi, lunedì, '9:30', 'U4', 3, antoniotti).
corso(biologia_computazionale, lunedì, '14:30', 'U14', t023,
      antoniotti).
```

A partire da questa definizione possiamo poi costruire altri predicati.

```
aula(Corso, Edificio, Aula) :-
    corso(Corso, _, _, Edificio, Aula, _).

docente(Corso, Docente) :-
    corso(Corso, _, _, _, _, Docente).
```

Possibilità 2

Tutte le informazioni sono concentrate in una relazione con 4 campi; le informazioni sono concentrate in termini funzionali che rappresentano informazioni raggruppate logicamente.

```
corso(linguaggi, orario(lunedì, '9:30'), aula('U4', 3),
      antoniotti).

corso(biologia_computazionale,
      orario(lunedì, '14:30'),
      aula('U4', 3),
      antoniotti).
```

A partire da questa definizione possiamo poi costruire altri predicati

```
aula(Corso, Edificio, Aula) :- corso(Corso, _, aula(Edificio,
Aula), _).

docente(Corso, Docente) :- corso(Corso, _, _, Docente).
```

%%% oppure...

```
aula(Corso, Luogo) :- corso(Corso, _, Luogo, _).
```

Possibilità 3

I predicati che abbiamo definito a partire dalle relazioni con 6 o 4 campi possono essere ricodificate con predicati binari

```

giorno(linguaggi, martedì).
orario(linguaggi, '9:30').
edificio(linguaggi, 'U4').
aula(linguaggi, 3).
docente(linguaggi, antoniotti).

```

Le relazioni a 6 o 4 argomenti possono essere ricostruite a partire da queste relazioni binarie. La costruzione di schemi RDF/XML (e, a volte, SQL) corrisponde a questa operazione di ri rappresentazione.

5.6.13 Le liste in Prolog

Esempi

| | |
|-------------------------|---------------------------------|
| [a, b, c] | Testa: a Coda: [b, c] |
| [a, b] | Testa: a Coda: [b] |
| [a] | Testa: a Coda: [] |
| [[a]] | Testa: [a] Coda: [] |
| [[a, b], c] | Testa: [a, b] Coda: [c] |
| [[a, b], [c], d] | Testa: [a, b] Coda: [[c], d] |

Si definisce una lista in Prolog racchiudendo gli elementi (termini e/o variabili logiche) della lista tra parentesi quadre '[' e ']' e separandoli da virgolette

Gli elementi di una lista in Prolog possono essere termini qualsiasi o liste.

[] indica la lista vuota.

Ogni lista non vuota può essere divisa in due parti: una testa ed una coda.

- La testa è il primo elemento della lista
- La coda rappresenta tutto il resto ed è sempre una lista

5.6.14 L'operatore |

Esempi (notare la convenzione Ys, Zs)

```

?- [X | Ys] = [mia, vincent, jules, yolanda].
   X = mia
   Ys = [vincent, jules, yolanda]
   Yes

```

Prolog possiede uno speciale operatore usato per distinguere tra l'inizio e la coda di una lista: l'operatore |.

```

?- [X, Y | Zs] = [the, answer, is, 42].
   X = the
   Y = answer
   Zs = [is, 42]
   Yes

```

La lista vuota [] in prolog viene gestita come una lista speciale.

```

?- [X, 42 | _] = [41, 42, 43, foo(bar)].
   X = 41
   Yes

```

?- [X | Ys] = [].

No

5.6.15 L'interprete Prolog: consult

La base di conoscenza è nascosta ed è accessibile solo tramite opportuni comandi o tramite ambiente di programmazione. Ovviamente è necessario poter inizializzare o caricare un insieme di fatti e regole nell'ambiente Prolog.

Il comando principale che assolve questa funzione è consult:

- Il comando `consult` appare come un predicato da valutare (un goal) e prende almeno un termine che denota un file come argomento
- Il file deve contenere un insieme di fatti e regole

Esempi

```
?- consult('guida-astrostoppista.pl').
   Yes
?- consult('Projects/Lang/Prolog/Code/esempi-liste.pl').
   Yes
```

Il predicato `consult` può essere usato anche per inserire fatti e regole direttamente alla console usando il termine speciale `user`.

Esempi

```
?- consult('guida-astrostoppista.pl').
   Yes

% A questo punto la base di dati Prolog contiene il
% nuovo contenuto del file.
```

5.6.16 L'interprete Prolog: `reconsult`

Esempi

```
?- reconsult(user). % Notare il sotto-prompt.
| - foo(42).
| - friends(zaphod, trillian).
| - ^D
   Yes

% A questo punto la base di dati Prolog
% contiene i due
% fatti inseriti manualmente.

?- friends(zaphod, W).
W = trillian
   Yes
```

Il predicato `reconsult` deve invece essere usato quando si vuole ricaricare un file (ovvero un data o knowledge base) nell'ambiente prolog.

L'effetto è di riprendere i predicati presenti nel file, rimuoverli completamente dal database interno e di reinstallarli utilizzando le nuove definizioni.

Prolog lavora su strutture ad albero, anche i programmi sono strutture dati manipolabili (mediante predici "extra-logici"). Prolog utilizza la ricorsione e non l'assegnamento.

Una metodologia di programmazione è quella di concentrarsi sulla specifica del problema rispetto alla strategia di soluzione.

Un programma prolog è un insieme di clausole di Horn che rappresentano:

- Fatti riguardanti gli oggetti in esame e le relazioni che intercorrono tra di loro
- Regole sugli oggetti e sulle relazioni (se... allora...)
- Interrogazioni (goals o queries, clausole senza testa), sulla base della conoscenza definita

5.7 CLAUSOLE

5.7.1 Implicazione

Sappiamo che $\neg B \vee A$ corrisponde a $B \rightarrow A$, oppure A implicato da B (che si legge "A se B"), $A \leftarrow B$.

Data una clausola

$$A_1 \vee A_2 \vee \dots \vee A_n \vee \neg B_1 \vee \neg B_2 \vee \dots \vee \neg B_m$$

usando la nota formula di riscrittura (teorema di De Morgan):

$$(A_1 \vee A_2 \vee \dots \vee A_n) \vee \neg(B_1 \wedge B_2 \wedge \dots \wedge B_m)$$

Da cui otteniamo

$$(A_1 \vee A_2 \vee \dots \vee A_n) \leftarrow (B_1 \wedge B_2 \wedge \dots \wedge B_m)$$

Ovviamente anche il percorso inverso è valido.

Qua la dimostrazione di De Morgan:

| A | B | Non(A) | Non(B) | Non(A) e non(B) | A o B | Non(A o b) |
|---|---|--------|--------|-----------------|-------|------------|
| V | V | F | F | F | V | F |
| V | F | F | V | F | V | F |
| F | V | V | F | F | V | F |
| F | F | V | V | V | F | V |

| A | B | Non(A) | Non(B) | Non (A) o non(B) | A e B | Non(A e b) |
|---|---|--------|--------|------------------|-------|------------|
| V | V | F | F | F | V | F |
| V | F | F | V | V | F | V |
| F | V | V | F | V | F | V |
| F | F | V | V | V | F | V |

Definizione: le A e le B nelle formule precedenti si dicono letterali, negativi quelli negati e positivi gli altri.

5.7.2 Clausole di Horn

Una clausola di Horn ha – al più – un solo letterale positivo

$$A \leftarrow B_1 \wedge B_2 \wedge \dots \wedge B_m \quad \text{oppure} \quad B_1 \wedge B_2 \wedge \dots \wedge B_m \rightarrow A$$

In particolare, possiamo classificare le clausole di Horn nel modo seguente:

| | |
|-----------------------|---|
| Fatti | $A \leftarrow$ |
| Regole | $A \leftarrow B_1 \wedge B_2 \wedge \dots \wedge B_m$ |
| Goals | $\leftarrow B_1 \wedge B_2 \wedge \dots \wedge B_m$ |
| Contraddizione | \leftarrow |

E, come abbiamo visto, in Prolog questi diventano

| | |
|-----------------------|-----------------------------|
| Fatti | $A.$ |
| Regole | $A :- B_1, B_2, \dots, B_m$ |
| Goals | $:- B_1, B_2, \dots, B_m$ |
| Contraddizione | $fail$ |

5.8 UN PROGRAMMA LOGICO

Un programma logico che manipola una rappresentazione dei numeri naturali

```
sum(0, X, X) .  
sum(s(X), Y, s(Z)) :- sum(X, Y, Z) .
```

possiamo interpretare $s(N)$ come il successore del numero N , quindi $0, s(0), s(s(0)), s(s(s(0))) \dots$

rappresentano $0, 1, 2, 3, \dots$. Questo programma definisce la somma fra due numeri naturali (rappresentati in "unario").

Possiamo interrogare il programma nel modo seguente:

$$\begin{aligned} \exists X \ sum(s(0), 0, X) & \quad \{X / s(0)\} \\ \exists W \ sum(s(s(0)), s(0), W) & \quad \{W / s(s(s(0)))\} \end{aligned}$$

Mediante il procedimento di negazione e di trasformazione in sintassi Prolog otteniamo

$$\begin{aligned} :- \ sum(s(0), 0, N) & \quad \{N / s(0)\} \\ :- \ sum(s(s(0)), s(0), W) & \quad \{W / s(s(s(0)))\} \end{aligned}$$

5.8.1 Sostituzioni

Nell'esempio precedente abbiamo visto le sostituzioni $\{W / s(s(s(0)))\}$ e $\{N / s(0)\}$. Una sostituzione ci dice con che valori possiamo sostituire le variabili in un termine. Di solito si denota una sostituzione nel modo seguente: $\sigma = \{X_1/v_1, X_2/v_2, \dots, X_k/v_k\}$.

Una sostituzione può essere considerata come una funzione, applicabile ad un termine (T è l'insieme dei termini) $\sigma:T \rightarrow T$.

Esempio, data la sostituzione $\sigma=\{X/42, Y/foo(s(0))\}$:

$$\sigma(\text{bar}(X, Y)) = \text{bar}(42, \text{foo}(s(0)))$$

5.8.2 Esecuzione di un programma

Una computazione corrisponde al tentativo di dimostrare, tramite la regola di risoluzione, che una formula segue logicamente da un programma (è un teorema). Inoltre, si deve determinare una sostituzione per le variabili del goal (detto anche query) per cui la query segue logicamente dal programma.

Dato un programma P e la query $:- p(t_1, t_2, \dots, t_m)$.

Se X_1, X_2, \dots, X_n sono le variabili che compaiono in t_1, t_2, \dots, t_m il significato della query è:

$$\exists X_1, X_2, \dots, X_n. p(t_1, t_2, \dots, t_m)$$

E l'obiettivo è quello di trovarne una sostituzione

$$S=\{X_1/s_1, X_2/s_2, \dots, X_n/s_n\}$$

Dove gli si sono trovati termini tali per cui

$$P \vdash_S [p(t_1, t_2, \dots, t_m)]$$

Dato un insieme di clausole di Horn, è possibile derivare la clausola vuota se ce n'è almeno una senza testa, ovvero se abbiamo almeno una query G_0 da provare.

Si deve dimostrare che da $P \cup \{G_0\}$ è possibile derivare la clausola vuota \rightarrow dimostrazione per assurdo mediante applicazione del principio di risoluzione.

5.8.3 Risoluzione ad input lineare (SLD)

Il sistema Prolog dimostra la veridicità o meno di un'interrogazione (un goal) eseguendo una sequenza di passi di risoluzione. L'ordine complessivo con cui questi passi vengono eseguiti rende i sistemi di prova di teoremi basati su risoluzione più o meno "efficienti".

In Prolog la risoluzione avviene sempre fra l'ultimo goal derivato in ciascun passo e una "clausola di programma", mai fra due clausole di programma o fra una clausola di programma ed un goal derivato in precedenza.

Questa particolare forma di risoluzione viene detta risoluzione SLD (Selection function for Linear and Definite sentences resolution, dove le frasi lineari sono essenzialmente le clausole di Horn).

Il risultato finale può essere:

- Successo: viene generata la clausola vuota, ovvero se per n finito G_n è uguale alla clausola vuota $G_n \equiv :-$
- Insuccesso finito: se per n finito G_n non è uguale a $:-$ e non è più possibile derivare un nuovo risolvente da G_n ed una clausola di programma
- Insuccesso infinito: se è sempre possibile derivare nuovi risolventi tutti diversi dalla clausola vuota

La sostituzione di risposta è la sequenza di unificatori usati; applicata alle variabili nei termini del goal iniziale dà la risposta finale.

Durante il processo di generazione di goal intermedi si costruiscono delle varianti dei letterali e delle clausole coinvolti mediante la rinominazione di variabili.

Una variante per una clausola C è la clausola C_1 ottenuta da C rinominando le sue variabili.

5.8.4 Strategia di selezione di un sottogoal

Si possono adottare diverse strategie di ricerca per queste clausole:

- In profondità (depth first): si sceglie una clausola e si mantiene fissa questa scelta, finché non si arriva alla clausola vuota o all'impossibilità di fare nuove risoluzioni.
- In ampiezza (breadth first): si considerano in parallelo tutte le possibili alternative

Il prolog adotta una strategia in profondità con backtracking che permette di risparmiare memoria.

La regola di calcolo è variabile:

- Leftmost: dal sottogoal più a sinistra
- Rightmost: dal sottogoal più a destra
- Sottogoal casuale
- Sottogoal migliore

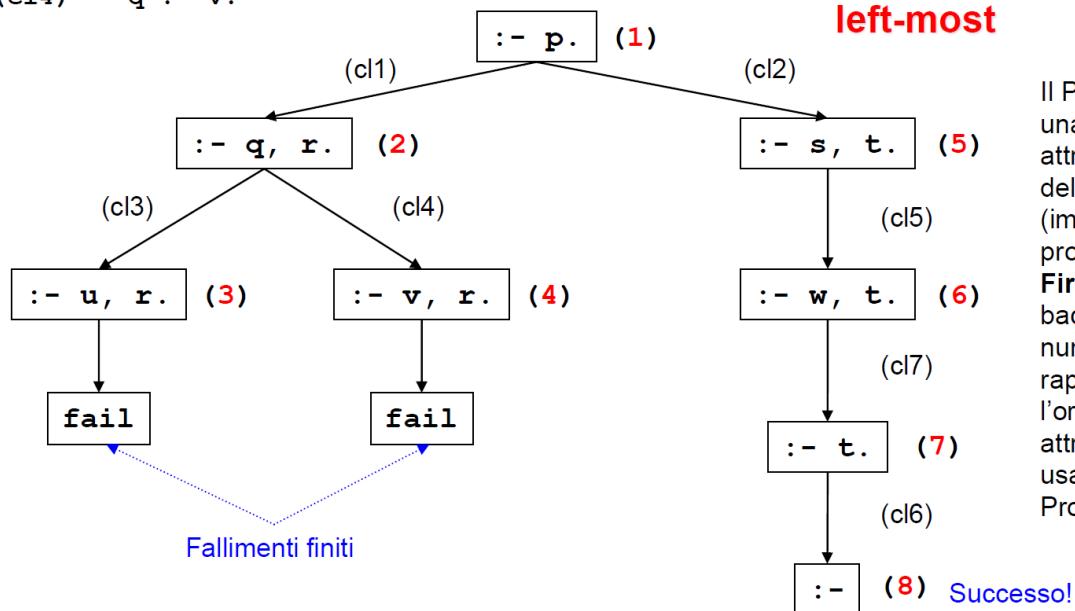
Il prolog usa la regola leftmost e dall'alto verso il basso.

```
(cl1) p :- q, r.
(cl2) p :- s, t.
(cl3) q :- u.
(cl4) q :- v.
```

```
(cl15) s :- w.
(cl16) t.
(cl17) w.
```

goal :- p.

Albero di risoluzione left-most



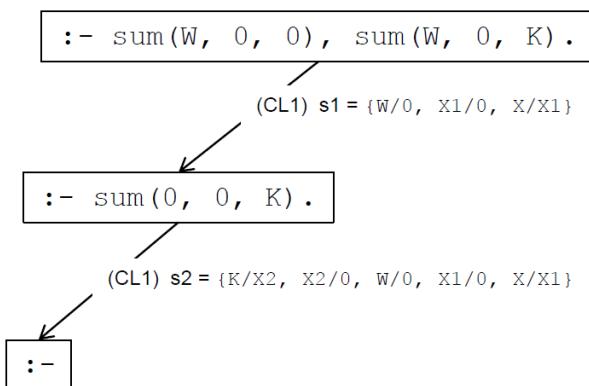
Il Prolog adotta una strategia di attraversamento dell'albero (implicito) SLD in profondità (**Depth First**) con backtracking. Il numero in **rosso** rappresenta l'ordine di attraversamento usato dal sistema Prolog

La regola di calcolo R influisce sulla struttura dell'albero ma non sulla sua correttezza e completezza, ovvero indipendentemente da R il numero di cammini di successo è invariabile.

Ad esempio:

```
sum(0, X, X). (CL1)
sum(s(X), Y, s(Z)) :- sum(X, Y, Z). (CL2)
G0 = :- sum(W, 0, 0), sum(W, 0, K).
```

Albero SLD con regola di calcolo “left-most”



Le variabili X_1 e X_2 sono il risultato dell'operazione di ridenominazione (renaming) della variabile X in CL1; appena una clausola viene presa in considerazione le sue variabili sono ridenominate.

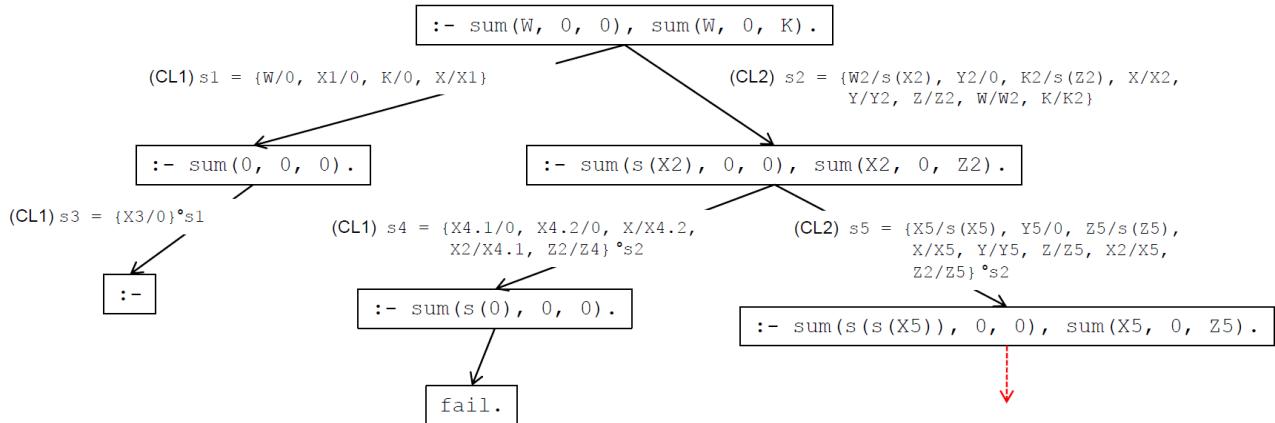
La notazione \circ indica la **composizione** di sostituzioni

```

sum(0, X, X) .                               (CL1)
sum(s(X), Y, s(Z)) :- sum(X, Y, Z) .       (CL2)
G0 = :- sum(W, 0, 0), sum(W, 0, K) .

```

Albero SLD con regola di calcolo “right-most”

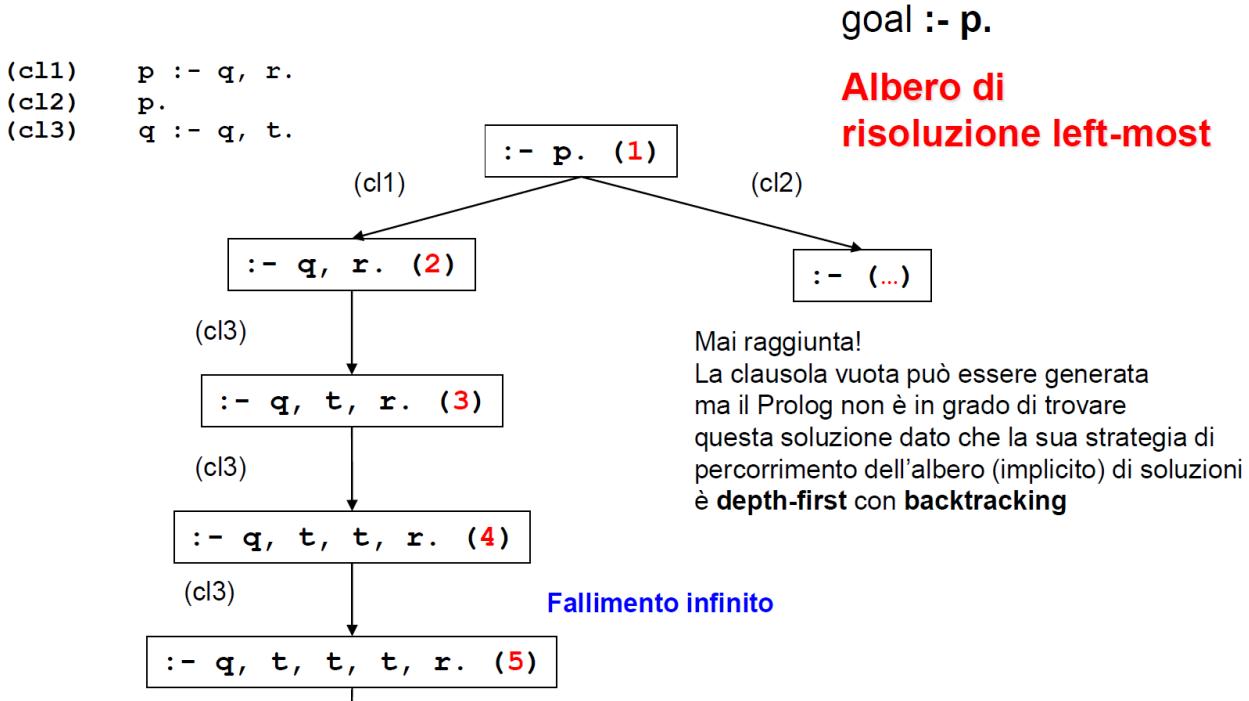


Come si nota il numero di successi è invariato mentre quello di insuccessi varia.

Quando si usano le variabili si devono tenere a mente alcune regole:

- Ad ogni passaggio le variabili vanno unificate esplicitamente e rinominate (ad esempio a $s_1 X_1/0$ e X/X_1)
- Le variabili rinominate vanno riscritte ad ogni passaggio successivo, esplicitamente o tramite l'utilizzo di s_n

Ecco un esempio di insuccesso infinito:

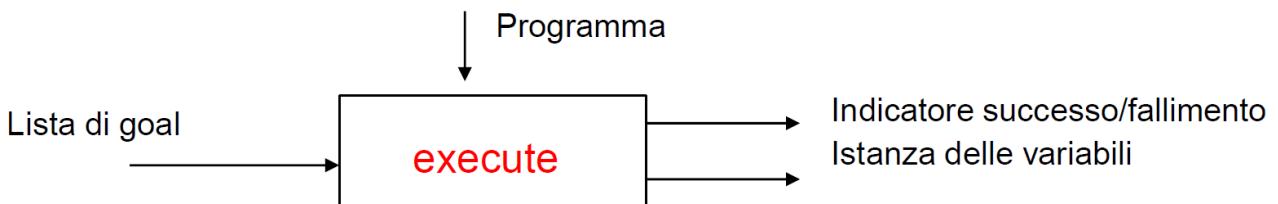


5.8.5 Modello di esecuzione Prolog

$p :- q, r.$

Interpretazione dichiarativa: p è vero se sono veri q e r .

Interpretazione procedurale: il problema p può essere scomposto nei sottoproblemi q e r .



5.8.6 Estensioni

Per rendere il Prolog un linguaggio effettivamente utilizzabile vengono aggiunti:

- Notazione per le liste
- Meccanismi per il caricamento del codice Prolog
- Meccanismi di controllo del backtracking
- Operazioni aritmetiche
- Trattamento della negazione
- Possibilità di manipolare e confrontare le strutture dei termini
- Predicati meta-logici ed extra-logici
- Predicati di input/output
- Meccanismi per modificare/accedere alla base di conoscenza

5.8.7 Il controllo di esecuzione di un programma

Come abbiamo intuito, le clausole nel database di un programma Prolog vengono considerate secondo la regola di calcolo leftmost e dall'alto verso il basso. Se un sottogoal fallisce, allora il dimostratore Prolog sceglie un'alternativa, scandendo dall'alto verso il basso la lista delle clausole.

Il prolog mette a disposizione un predicato speciale, chiamato cut (il cui simbolo è un punto esclamativo !) per controllare questa sequenza di scelte. Il cut è molto complicato da interpretare ma la sua importanza non è da sottovalutare.

5.8.8 Il predicato cut “!”

Consideriamo la seguente clausola generica con cut

$$C = a:- b_1, b_2, \dots, b_k, !, b_{k+1}, \dots, b_n.$$

L'effetto del cut è il seguente:

- Se il goal corrente G unifica con a e b_1, \dots, b_k hanno successo, allora il dimostratore si impegna inderogabilmente alla scelta di C per dimostrare G .
- Ogni clausola alternativa (successiva e in basso) per a che unifica G viene ignorata
- Se un qualche b_j con $j > k$ fallisse, il backtracking si fermerebbe al cut !
 - Le altre scelte per i b_i con $i \leq k$ sono di conseguenza rimosse dall'albero di derivazioni
- Quando il backtracking raggiunge il cut, allora il cut fallisce e la ricerca procede dall'ultimo punto di scelta prima che G scegliestesse C

ESEMPIO

Considerare il seguente programma Prolog

```
cl1  a  :- p, b.  
cl2  a  :- p, c.  
cl3  p.
```

Query: ?- a.

Lo stato interno del sistema prolog diventa il seguente.

| | | | |
|---|-------------------------|-----------------|--|
| Stack di esecuzione | (cl1) (cl2) | Scelta corrente | Si unifica a con CL1 a. → p, b. |
| Stack di esecuzione | (cl3) (cl1) (cl2) | Scelta corrente | Sullo stack prima viene messo il primo letterale, p e sulle scelte vengono messe quelle che unificano con p, ovvero CL3 a. → p, b |
| Stack di esecuzione | (cl1) (cl2) | Scelta corrente | La valutazione di p. con CL3 ha successo, quindi entrambi gli stack subiscono un pop a. → p, b |
| Stack di esecuzione | b a | Scelte per a | Si passa al secondo letterale b ma non ci sono clausole che unifichino, quindi non si aggiunge niente alle scelte a. → p, b |
| Stack di esecuzione | a | Scelta corrente | Non potendo proseguire entrambe le pile subiscono un pop, attivando così il backtracking: infatti eliminando CL1 dallo stack delle scelte si torna al ramo precedente e si inizia con CL2 a. → p, b |
| Seguono passaggi uguali con c che non può essere unificato, quindi entrambe le pile subiscono un pop, eliminando anche CL2 e lasciando solo a sullo stack di esecuzione | | | |
| A questo punto, non potendo più unificare a con alcuna clausola, la dimostrazione fallisce. | | | |

Quindi ci sono due pile (stacks):

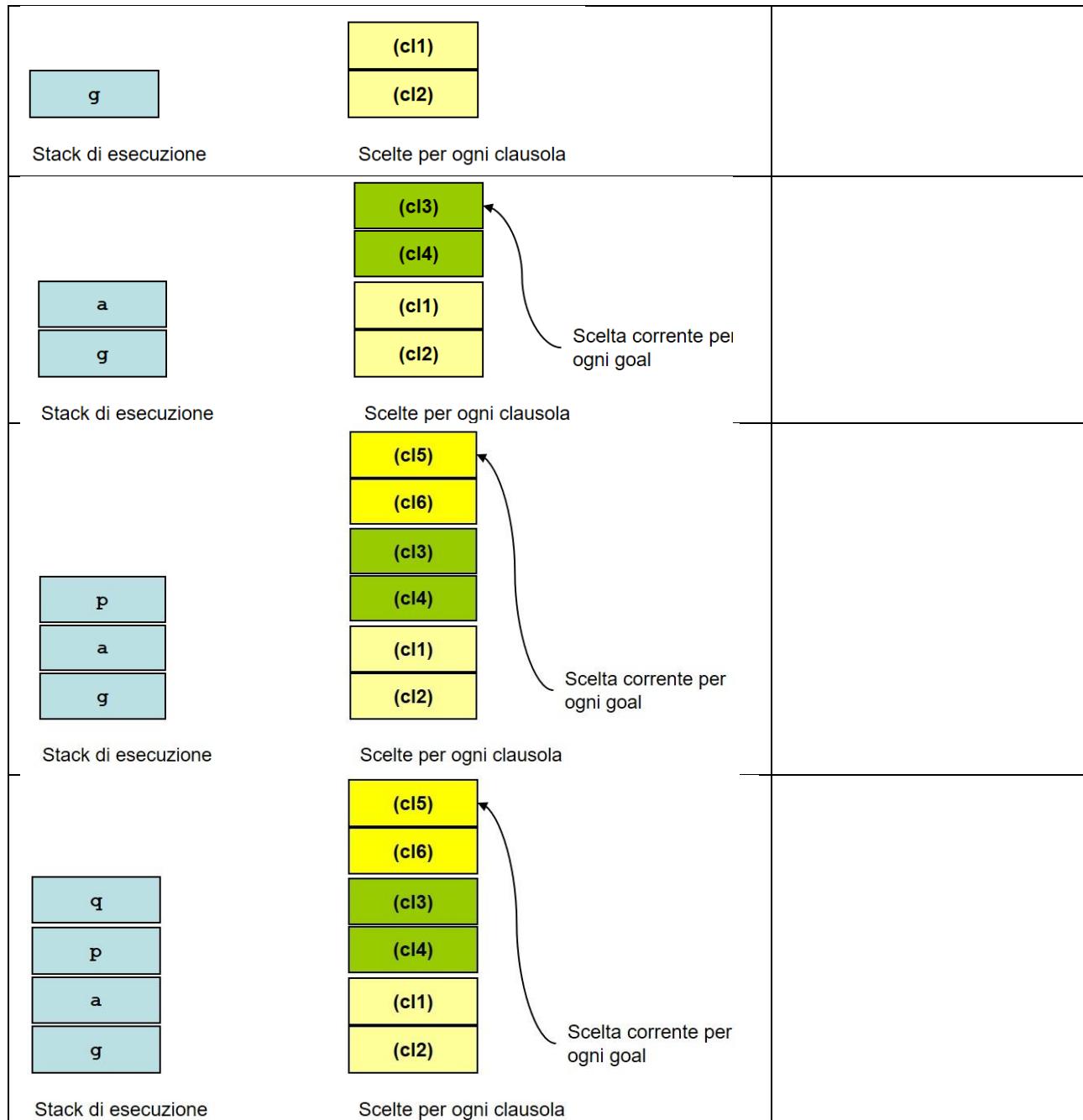
- Pila di esecuzione che contiene i record di attivazione delle varie "procedure"
- Pila di backtracking che contiene l'insieme dei "puntatori alle scelte", ad ogni fase della valutazione questa pila contiene dei puntatori alle scelte aperte nelle fasi precedenti della dimostrazione.

Ora introduciamo il cut, !.

```

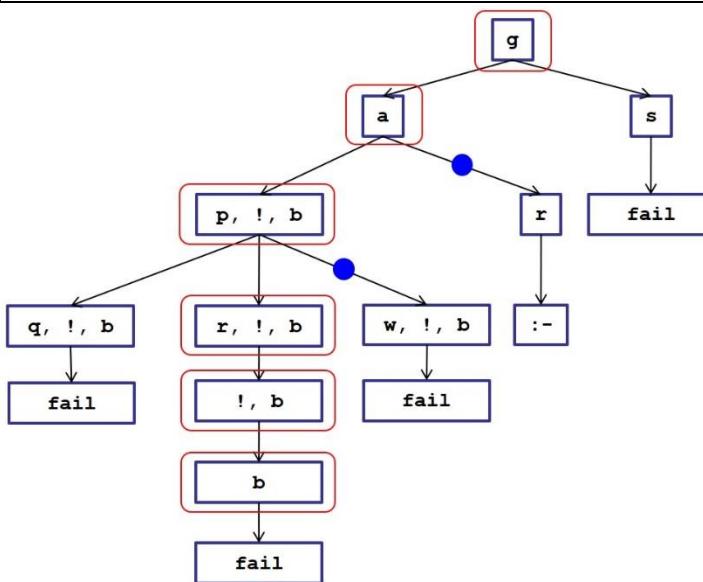
cl1  g :- a.      Consideriamo il goal
cl2  g :- s.
cl3  a :- p, !, b. ?- g.
cl4  a :- r.
cl5  p :- q.
cl6  p :- r.
cl7  r.

```





| | | |
|-------------------------|-----------------------------------|---|
| Stack di esecuzione | Scelte per ogni clausola | |
| Stack di esecuzione | Scelte per ogni clausola | |
| Stack di esecuzione | Scelte per ogni clausola | |
| Stack di esecuzione | Scelte per ogni clausola | S fallisce e lascia lo stack di scelte vuoto ma con un goal ancora da provare |
| Stack di esecuzione | Scelta corrente per ogni goal | Non siamo riusciti a generare la clausola vuota |



Si possono distinguere due tipi di cut:

- Green cuts: utili per esprimere “determinismo” e quindi rendere più efficiente il programma
- Red cuts: usati per soli scopi di efficienza, hanno per caratteristica principale quella di bloccare alcune condizioni esplicite in un programma e, soprattutto, quella di modificare la semantica del programma equivalente senza cuts. Sono tendenzialmente indesiderabili poiché rischiano di compromettere la correttezza del programma.

Consideriamo il seguente programma che serve a fare il “merge” di due liste ordinate.

```

merge([X | Xs], [Y | Ys], [X | Zs]) :-  

  X < Y,  

  merge(Xs, [Y | Ys], Zs).  
  

merge([X | Xs], [Y | Ys], [X | Zs]) :-  

  X = Y,  

  merge(Xs, Ys, Zs).  
  

merge([X | Xs], [Y | Ys], [X | Zs]) :-  

  X > Y,  

  merge([X | Xs], Ys, Zs).  
  

merge([], Ys, Ys).  

merge(Xs, [], Xs).  
  

merge([X | Xs], [Y | Ys], [X | Zs]) :-  

  X < Y,  

  !,  

  merge(Xs, [Y | Ys], Zs).  
  

merge([X | Xs], [Y | Ys], [X | Zs]) :-  

  X = Y,  

  !,  

  merge(Xs, Ys, Zs).  
  

merge([X | Xs], [Y | Ys], [X | Zs]) :-  

  X > Y,  

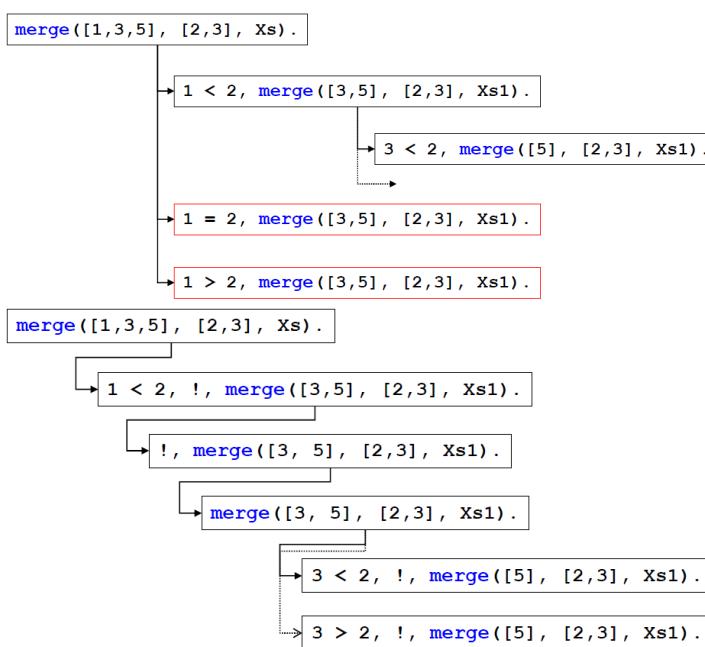
  !,  

  merge([X | Xs], Ys, Zs).  
  

merge([], Ys, Ys).  

merge(Xs, [], Xs).

```



in questo caso tutte le clausole verranno unificate indipendentemente dal successo o no di queste. Facendo l'esempio "X = 4, Y = 3", prima verrà unificato il primo merge con successo ma, nonostante questo, verranno unificati anche i successivi, trovandosi a fallire al momento del confronto numerico.

Un programma Prolog si dice deterministico quando una sola delle clausole serve (o si vorrebbe servisse) per provare un dato goal. Come già visto, i cuts che servono per esplicitare questo determinismo vengono detti green cuts.

In questo secondo esempio di merge, sono stati inseriti dei cut dopo i confronti numerici, in questo modo verranno unificate solo le clausole precedenti a quella che risulterà corretta.

Facendo l'esempio di prima, "X = 4, Y = 3", la prima clausola verrà unificata e a quel punto le altre non verranno considerate.

qua i due percorsi, si nota che nel primo c'è uno spreco di tempo durante il confronto con le altre due clausole mentre nel secondo caso la sequenza sarà più rapida e senza unificazioni inutili.

ora usiamo un altro esempio, ovvero un programma che cerca il minimo.

```
min(X, Y, X) :- X <= Y, !.
min(X, Y, Y) :- Y < X, !.
```

in questo caso il secondo cut è ridondante ma viene comunque messo per motivi di simmetria.

Ora esaminiamo lo stesso caso ma con un red cut.

```
min(X, Y, X) :- X <= Y, !.
min(X, Y, Y) .
```

in questo caso il cut è red, dato che serve solo a tagliare delle soluzioni. Non solo, il goal “min(2, 5) viene verificato come true. Qua l'esempio del perché i red cuts vanno usati con estrema cura.

5.9 PREDICATI META-LOGICI

Consideriamo il predicato:

```
celsius_fahrenheit(C, F) :- C is 5/9 * (F - 32).
%il predicato non è invertibile e l'introduzione della seconda clausola
celsius_fahrenheit(C, F) :- F is (9/5 * C) + 32.
% non aiuta, dato che il sistema si blocca (a causa di un errore) sulla
% prima
```

il problema è che dobbiamo decidere quale è l'input e quale è l'output del nostro calcolo. Per risolvere questo problema abbiamo bisogno di prediciati meta-logici.

Alcuni prediciati quindi non hanno la tipica invertibilità dei risultati di varie queries. La ragione di questo effetto sta nell'uso che abbiamo fatto di vari prediciati aritmetici nel corpo dei prediciati ($>$, $<$, \leq , is , etc.).

Ovvero, per poter usare i prediciati aritmetici che usano direttamente l'hardware abbiamo sacrificato la semantica dei nostri programmi.

I prediciati meta-logici principali trattano le variabili come oggetto del linguaggio e ci permettono di riscrivere molti programmi che usano i prediciati aritmetici di sistema come prediciati dalla semantica "corretta" e dal comportamento invertibili.

I prediciati meta-logici più importanti sono:

- **var(X)**: vero se X è una variabile logica
- **nonvar(X)**: l'opposto di var.

```
celsius_fahrenheit(C, F) :-
  var(C),
  nonvar(F),
  C is 5/9 * (F - 32).
```

introducendo questi prediciati nelle regole viste prima otteniamo le regole qua accanto.

```
celsius_fahrenheit(C, F) :-
  var(F),
  nonvar(C),
  F is (9/5 * C) + 32.
```

l'uso di var(X) ci permette di decidere quale clausola utilizzare. Quindi l'uso di questi prediciati ci permette di scrivere dei programmi efficienti ed allo stesso semanticamente "corretti".

5.10 ISPEZIONE DI TERMINI

Finora abbiamo visto come si usano dei termini per rappresentare diverse strutture dati in Prolog.

In particolare abbiamo anche intuito che esistono termini atomici (corrispondenti alle costanti in un linguaggio logico del primo ordine) e termini composti (funzioni e predici).

In Prolog abbiamo a disposizione i predicati:

- **atomic(X)**: vero se X è un numero o una costante
- **compound(X)**: vero se non è atomico.

Dato un termine Term abbiamo tre predicati che ci tornano utili per manipolarlo:

- **functor(Term, F, Arity)**: vero se Term è un termine, con Arity argomenti, il cui funtore (simbolo di funzione o di predicato) è F
- **arg(N, Term, Arg)**: vero se l'N-esimo argomento di Term è Arg
- **Term =.. L**: questo predicato, =.., viene chiamato (per motivi storici) univ; è vero quando L è una lista il cui primo elemento è il funtore di Term ed i rimanenti elementi sono i suoi argomenti

Esempi:

?- functor(foo(24), foo, 1).
true.

?- arg(3, node(x, _, [], []), X).
X = [].

?- functor(node(X, _, [], []), F, 4).
F = node.

?- arg(1, father(X, lot), haran).
X = haran.

?- functor(Term, bar, 2).
Term = bar(_528, _530).

?- father(harn, lot) =.. Ts.
Ts = [father, harn, lot].

?- father(X, lot) =.. [father, haran, lot].
X = haran.

5.11 PROGRAMMAZIONE DI ORDINE SUPERIORE

Quando si formula una domanda per il sistema Prolog, ci si aspetta una risposta che è un'istanza (individuale) derivabile dalle conoscenze base (S).

Il meccanismo di backtracking ci permette di estrarre tutte le istanze che possono essere derivate, una alla volta.

Cosa succede se vogliamo come risultato l'insieme di tutte le istanze (soluzioni) che soddisfano una certa query?

Questa richiesta non è una richiesta formulabile direttamente ad un linguaggio logico del primo ordine. La richiesta è al secondo ordine, dato che richiede un insieme di elementi che soddisfano una certa proprietà. Prolog mette a disposizione dell'utente una serie di predicati su insiemi che estendono il modello computazionale del linguaggio base.

5.11.1 Predicati su insieme

I predicati su insiemi più importanti sono tre:

- **findall(Template, Goal, Set)**:

- Vero se Set contiene tutte le istanze di Template che soddisfano Goal
- Le istanze di Template vengono ottenute tramite backtracking
- **bagof(Template, Goal, Bag) :**
 - Vero se Bag contiene tutte le alternative di template che soddisfano Goal
 - Le alternative vengono cotruite facendo backtracking solo se vi sono delle variabili libere in Goal che non appaiono in template
 - È possibile dichiarare quali variabili non vanno considerate libere al fine del backtracking grazie alla sintassi Var^G come Goal, in questo caso Var viene pensata come una variabile esistenziale
- **setof(Template, Goal, Set) :**
 - Si comporta come bagof ma non contiene soluzioni duplicate

```
?- findall(C, father(X, C), Kids).
```

```
C = _0
X = _1
Kids = [abraham, nanchor, haran, isaac, lot, milcah, yiscah]
Yes.
```

```
?- bagof(C, father(X, C), Kids).
```

```
C = _0
X = terach
KIDS = [abraham, haran, nanchor];
```

```
C = _0
X = haran
KIDS = [lot, yiscah, milcah];
```

```
C = _0
X = abraham
KIDS = [isaac];
NO.
```

```
?- bagof(C, X^father(X, C), Kids).
```

```
C = _0
X = _1
Kids = [abraham, haran, lot, yiscah, nanchor, isaac, milcah];
NO.
```

5.11.2 Predicati di ordine superiore e meta variabili

Il Prolog mette a disposizione dell'utente anche altri predicati di ordine superiore. Buona parte di questi predicati funziona fgrazie al meccanismo delle emta-variabili, ovvero variabili interpretabili come goals.

Un esempio tipico è il predicato call, che si può pensare essere definito come "call(G) :- G.".

Grazie alle meta-variabili possiamo definire il predicato applica che valuta una query composta da un funtore e da una lista di argomenti

```
applica(P, Args) :-
  P =.. PL,
  append(PL, Args, GL),
  Goal =.. GL,
  call(Goal).
```

```
?- applica(father, [X, C]).
X = terach
C = abraham;
X = terach
C = nanchor;
No
```

Applica possiamo vederlo come uno strumento utile all'utente per controllare le conoscenze prodotte a runtime. È uno strumento dato dal programmatore all'utente e può avere sia un risvolto positivo che uno negativo: il primo è permettere al programma di essere integrabile con funzioni non previste precedentemente dal programmatore, il secondo è che il codice rischia di diventare un'accozzaglia poco leggibile di conoscenze, alcune delle quali inutili.

```
?- applica(father(terach), [C]).
C = abraham;
C = nanchor;
No
```

5.12 MODIFICA DELLA BASE DI CONOSCENZE

Un programma Prolog è costituito da una base di dati che contiene fatti e regole.

Il Prolog però mette a disposizione anche altri predicati che servono a manipolare direttamente la base di dati. Ovviamente, questi predicati vanno usati con molta attenzione, dato che modifiano dinamicamente lo stato del programma. I predicati che servono a manipolare direttamente la base di dati sono:

- listing
- assert, asserta, assertz
- retract
- abolish

LISTING

elenca tutte le conoscenze base

ASSERT, ASSERTA, ASSERTZ

assert(S) ha sempre successo, la sua funzione è di aggiungere S alle conoscenze base.

asserta(S) aggiunge S all'inizio dell'elenco delle conoscenze base.

assertz(S) aggiunge S alla fine dell'elenco delle conoscenze base.

Questi comandi non controllano se la conoscenza S sia già presente all'interno delle conoscenze base, S verrà aggiunta indipendentemente da tutto.

RETRACT

retract(S) rimuove la conoscenza S dalla base di dati. S verrà rimossa solo alla sua prima occorrenza.

La manipolazione del database Prolog è una cosa molto utile, ad esempio può essere usata per memorizzare i risultati intermedi di varie computazioni, in modo da non dover rifare delle queries dispendiose in futuro: semplicemente si ricerca direttamente il fatto appena asserito.

```
addition_table(A) :-
    member(B, A),
    member(C, A),
    D is B + C,
    assert(sum(B, C, D)),
    !.
```

Questa tecnica si chiama memoization o caching.

Nell'esempio qua accanto il predicato fail serve per attivare il backtracking e aggiungere alle conoscenze di base tutte le possibili somme degli elementi B e C appartenenti ad A.

5.13 INPUT E OUTPUT IN PROLOG

I predicati principali per la gestione dell'I/O sono essenzialmente due, read e write, a cui si aggiungono i vari predicati per la gestione dei files e degli streams: open, close, seek, ecc.

Read e write sono peculiari: leggono e scrivono termini Prolog:

- Write è equivalente all'invocazione di un metodo `toString` Java su un oggetto di classe "termine"
- Read di fatto invoca il parser Prolog

```
?- write(42).
42
true.

?- foo(bar) = X, write(X).
foo(bar)
X = foo(bar).

?- read(What).
|: foo(42, bar). %|: sta per input dell'utente

What = foo(42, bar).

?- read(What), write('I just read: '), write(What).
|: read(What).
I just read: read(_714)
What = read(_714).

?- open('some/file/here.txt', write, Out),
   write(Out, foo(bar)), put(Out, 0'.), nl(Out),
   close(Out).
true
%% But file "some/file/here.txt" now contains the term 'foo(bar)'.

?- open('some/file/here.txt', read, In),
   read(In, What),
   close(In).
What = foo(bar)
```

Open e close servono per leggere e scrivere files; la versione più semplice di open ha tre argomenti:

- un atomo che rappresenta il nome del file
- una "modalità" con cui si apre il file
- un argomento a cui si associa l'identificatore del file

vi sono naturalmente molti altri predicati per I/O in Prolog, ad esempio sopra si vede il predicato `put/2`⁸ che emette un carattere sullo stream e il predicato `nl/1` che mette un “newline” sullo stream.

Inoltre, si vede che il Prolog usa la notazione `0'c` per rappresentare i caratteri come termini.

6 INTERPRETI IN “PROLOG”

Il Prolog si presta benissimo alla costruzione di interpreti per la manipolazione di linguaggi specializzati (Domain Specific Languages - DSL)

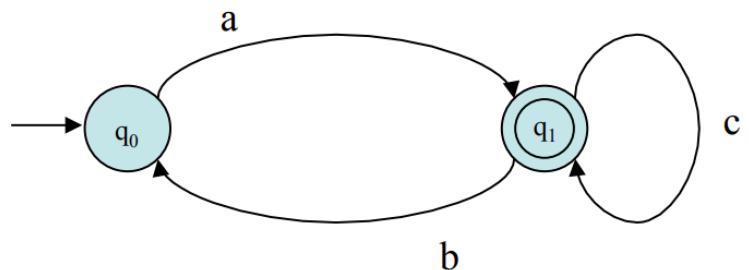
Esempi tipici sono:

- interpreti per automi (a stati finiti, a pila, macchine di Turing)
- sistemi per la deduzione automatica
- sistemi per la manipolazione del linguaggio naturale (natural Language Processing - NLP)

6.1 AUTOMI

Come costruiamo un “interprete” per riconoscere non deterministicamente dei linguaggi regolari?

```
accept([I | Is], S) :-  
    delta(S, I, N),  
    accept(Is, N).  
  
accept([], Q) :- final(Q).
```



l'automa viene codificato con:

```
initial(q0).
```

per decidere se una certa sequenza di simboli è riconosciuta dall'automa
possiamo costruire il seguente predicato:

```
final(q1).
```

```
recognize(Input) :-
```

```
    initial(S),
```

```
    accept(Input, S).
```

```
delta(q0, a, q1).
```

```
delta(q1, b, q0).
```

```
delta(q1, c, q1).
```

ESEMPIO

```
?- recognize([a, b, b, a, c, c, b, a]).
```

```
yes.
```

⁸ La notazione “predicato/2” indica il nome del predicato e la sua arietà.

```
?- recognize([a, b, a, c, b]).
```

No.

6.2 AUTOMI A PILA

Come costruiamo un “interprete” per riconoscere non deterministicamente dei linguaggi liberi da contesto (CFL)?

```
%%% accept(Input, Stato,
Pila) .  
  
accept([I | Is], Q, S) :-  
    delta(Q, I, S, Q1, S1),  
    accept(Is, Q1, S1).  
  
accept([], Q, []) :-  
    final(Q).
```

consideriamo il linguaggio $L = \{w w^R \mid w \in \{a, b, c\}^n, n \geq 0\}$. Il linguaggio non è regolare.

Scriviamo le regole necessarie per la codifica della funzione di transizione. L'automa viene codificato con:

```
initial(q0).  
  
final(q1).  
  
delta(q0, a, P, q0, [a | P]).  
delta(q0, b, P, q0, [b | P]).  
delta(q0, c, P, q0, [c | P]).  
delta(q0, r, q1, P, P).  
delta(q1, c, [c | P], q1, P).  
delta(q1, b, [b | P], q1, P).  
delta(q1, a, [a | P], q1, P).
```

Come nel caso degli automi a stati finiti, per decidere se una certa sequenza di simboli è riconosciuta dall'automa possiamo costruire il seguente predicato:

```
recognize(Input) :-  
    initial(S),  
    accept(Input, S, []).
```

ESEMPIO

```
?- recognize([a, b, a, c, r, c, a, b, a]).
```

Yes.

```
?- recognize([a, b, a, c, r, b]).
```

No.

6.3 META-INTERPRETI

Il predicato call/1 che abbiamo visto precedentemente è il più semplice meta-interprete Prolog.

I meta-interpreti sono solitamente utilizzati per aggiungere funzioni extra in Prolog.

Possiamo scrivere degli interpreti più complicati e/o specializzati se accettiamo di rappresentare i programmi con una sintassi leggermente diversa.

Considerare la base di dati seguente:

```
rule	append([], X, X)).  
rule	append([X | Xs], Ys, [X | Zs]), [append(Xs, Ys, Zs)]).
```

consideriamo ora il seguente programma:

```
solve(Goal) :-  
    solve(Goal, []).  
  
solve([], []).  
solve([], [G | Goals]) :-  
    solve(G, Goals).  
solve([A | B], Goals) :-  
    append(B, Goals, BGoals),  
    solve(A, BGoals).  
solve(A, Goals) :-  
    rule(A),  
    solve(Goals, []).  
solve(A, Goals) :-
```

```

rule(A, B),
solve(B, Goals).
```

Il programma **solve** è un meta-interprete per i predicati **rule** che compongono il nostro sistema o programma.

Ragioniamo con incertezza:

```
solve_cf(true, 1) :- !.
```

Il programma **solve_cf** è un meta-interprete per stabilire se un goal G è vero e quanto siamo certi che sia vero.

```

solve_cf((A, B), C) :-  

  !,  

  solve_cf(A, CA),  

  solve_cf(B, CB),  

  minimum(CA, CB, C).  
  

solve_cf(A, 1) :-  

  builtin(A),  

  !,  

  call(A).  
  

solve_cf(A, C) :-  

  rule_cf(A, B, CR),  

  solve_cf(B, CB),  

  C is CR * CB.
```

Ragioniamo con incertezza, ma con una soglia (indicata dalla variabile T, per “threshold”):

```

solve_cf(true, 1, T) :- !.  

solve_cf((A, B), C, T) :-  

  !,  

  solve_cf(A, CA, T),  

  solve_cf(B, CB, T),  

  minimum(CA, CB, C).  
  

solve_cf(A, 1, T) :-  

  builtin(A),  

  !,  

  call(A).  
  

solve_cf(A, C, T) :-  

  rule_cf(A, B, CR),  

  CR > T,
```

```
T1 is T/CR,  
solve_cf(B, CB, T1),  
C is CR * CB.
```

7 PARADIGMI DI PROGRAMMAZIONE

- Imperativo
 - I programmi computano modificando lo “stato” della “memoria” del sistema
- Object-oriented
 - I programmi computano mantenendo lo “stato” della “memoria” in “oggetti” che rispondono in modo particolare alle richieste che vengono fatte loro, tramite chiamate a metodi
- Funzionale
 - I programmi computano combinando “valori” (rappresentati in memoria) trasformati da chiamate a funzioni
- Dichiarativo
 - I programmi “computano” verificando che varie asserzioni su un sistema sono vere o false.
Il processo di verifica sostituisce determinati valori a variabili presenti nelle asserzioni

7.1 I LINGUAGGI IMPERATIVI

Basati sull’architettura di Von Neumann:

- Memoria costituita da celle identificate da un indirizzo, contenenti dati o istruzioni
- Unità di controllo e aritmetica
- Programma immagazzinato nella memoria centrale

Concetto di assegnamento di valori a variabili: ogni valore calcolato deve essere esplicitamente memorizzato, cioè assegnato ad una cella

Concetto di sequenza di istruzioni: ogni programma consiste nell’esecuzione di una sequenza di istruzioni con possibili salti.

7.2 EFFETTI COLLATERALI

Gli effetti collaterali sono una delle caratteristiche negative dei linguaggi imperativi. Modifiche dello stato di memoria di un programma, ad esempio tramite modifica di locazioni di memoria accessibili da variabili passate per riferimento o globali. Tipico uso è quello per la comunicazione tra diverse parti di un sistema.

Problemi:

- Distinguere tra effetti collaterali desiderati e non desiderati
- Leggibilità del programma (l’istruzione di chiamata non rivela quali variabili globali sono coinvolte)
- Risulta difficile verificare la correttezza di un programma

Ad esempio:

$$w := x + f(x, y) + z$$

la funzione $f()$ può modificare x e y se sono passati per indirizzo, e z se è globale:

- Abbiamo una leggibilità ridotta del programma
- Non ci si può fidare della commutatività dell’addizione

$$u := x + z + f(x, y) + f(x, y) + x + z$$

gli stessi problemi visti prima, più l’impossibilità per il compilatore di generare codice ottimizzato valutando una sola volta $x + z$ e $f(x, y)$.

7.3 TRASPARENZA REFERENZIALE

Un concetto matematico fondamentale è quello di trasparenza referenziale: il significato del tutto si può determinare dal significato delle parti.

Questa proprietà è valida per espressioni aritmetiche e matematiche, in particolare questo concetto rende possibile la sostituzione di espressioni con altre a patto che esse denotino gli stessi valori.

Esempio

Nell'espressione matematica $f(x) + g(x)$ si può sostituire la funzione f con la funzione h se producono valori identici. Nei linguaggi imperativi tradizionali non può essere certi della sostituzione, né che $f(x) + g(x) = g(x) + f(x)$ o che $f(x) + f(x) = 2*f(x)$.

I linguaggi funzionali (puri) hanno il concetto di trasparenza referenziale come fondamento.

7.4 LINGUAGGI DI PROGRAMMAZIONE FUNZIONALI

L'idea fondamentale dei linguaggi (e della programmazione funzionale) è il seguente:

programmi = funzioni matematiche

ovvero, un programma è costituito dalla combinazione di varie funzioni:

- Funzioni primitive
- Funzioni più complesse ottenute via composizione

La trasparenza referenziale propria della matematica viene mantenuta.

Come ci si può aspettare, programmare in un linguaggio funzionale richiede la manipolazione di funzioni. La notazione normale per denotare funzioni è la seguente:

$f(x_1, x_2, \dots, x_n)$

ad esempio $\cos(3.14)$ oppure $\text{substring}(\text{"la vita l'è bela"}, 3.7)$.

ogni funzione denota un valore ottenuto tramite una mappa a partire dagli "argomenti".

Nel paradigma funzionale vi sono oggetti di vario tipo e strutture di controllo, ma vengono raggruppati logicamente in modo diverso da come invece accade nel paradigma imperativo. In particolare è utile pensare in termini di:

- Espressioni (funzioni primitive e non)
- Modi di combinare tali espressioni per ottenerne di più complesse (composizione)
- Modi e metodi di costruzione di "astrazioni" per poter far riferimento a gruppi di espressioni per "nome" e per trattarle come unità separate
- Operatori speciali (condizionali ed altri ancora)

Definizione di funzione: regola per associare gli elementi di un insieme (dominio) a quelli di un altro insieme (codominio). Una funzione può essere applicata a un elemento del dominio (si noti che dominio e codominio possono essere il prodotto cartesiano di insiemi più semplici). L'applicazione produce (restituisce) un elemento del codominio detto valore.

Esempio

$\text{Quadrato}(x) \equiv x*x$

Dove x è un numero, detto argomento, che indica un generico elemento del dominio (ad esempio, l'insieme dei numeri reali \mathbb{R}). x è una variabile matematica, senza una precisa locazione in memoria, non ha senso pensare di modificarla. In un'implementazione di un linguaggio funzionale, x sarà mappata in particolari locazioni di memoria, ma non ci è dato modificarne il contenuto.

7.4.1 Composizione

Nei linguaggi funzionali espressioni più complesse vengono costruite mediante composizione.

Se F è definita come composizione di G e H :

$$F \equiv G \circ H$$

Applicare F equivale ad applicare G al risultato dell'applicazione di H .

7.4.2 Ricorsione e operatori speciali

Finora siete stati istruiti ad utilizzare dei linguaggi più o meno imperativi. In particolare avete visto come inserire semplici dati in un programma e di stampare dei risultati tramite quello che viene definito il sistema (o libreria) di input/output di un linguaggio. Tra questi due momenti avete visto una serie di costrutti e di oggetti raggruppabili come:

- Oggetti (numeri, strutture dati, files, ecc.)
- Strutture di controllo
 - If-then-else
 - For
 - While
 - Case
 - ...
- Assegnamenti
- Procedure, funzioni e metodi
 - Nei linguaggi imperativi le regole di trasformazione dal dominio al codominio corrispondono ai passi da eseguire in un ordine determinato dalle strutture di controllo

Le espressioni matematiche sono invece composte da composizione di funzioni spesso organizzate ricorsivamente e controllate da operatori speciali.

Esempio (non necessariamente in un linguaggio corrente)

$\text{Fattoriale}(x) \equiv \text{if } (x = 0) \text{ then } 1 \text{ else } x * \text{fattoriale}(x-1)$

Questa definizione di fattoriale utilizza un operatore speciale (if-then-else) per rappresentare valutazioni condizionali

Nella programmazione funzionale (pura) non è possibile produrre effetti collaterali.

La principale modalità di calcolo è l'applicazione di funzione. Il calcolo procede valutando espressioni, senza effetti collaterali (trasparenza referenziale).

Le funzioni sono oggetti di prima classe (non solo "puntatori a funzione"):

- Possono essere parte di una struttura dati
- Possono essere costruite durante l'esecuzione di un programma e ritornate come valore di un'altra funzione

I linguaggi funzionali consentono l'uso di funzioni di ordine superiore, cioè funzioni che prendono altre funzioni come argomenti e che possono restituirne come valore, in modo assolutamente generale.

Nei linguaggi funzionali puri non esistono strutture di controllo iterative come while e for; questi sono sostituiti da ricorsione combinata con gli operatori speciali condizionali.

8 LISP E IL PARADIGMA FUNZIONALE

LISP non è propriamente un linguaggio, ma una famiglia di linguaggi il cui acronimo sta per LISt Processing

A tutt'oggi vi sono due dialetti principali: Common lisp e Scheme (il linguaggio clojure è un nuovo dialetto molto usato di recente). Lo studio del LISP in una delle sue incarnazioni è importante dato che il linguaggio è uno dei più vecchi rappresentanti del paradigma di programmazione funzionale, altri linguaggi sono quelli della famiglia ML, Haskell, FP, F# (microsoft) e molti altri.

Le versioni minimali di LISP ammettono solo funzioni primitive su liste, un operatore speciale per creare funzioni (lambda), un operatore condizionale (cond) ed un piccolo insieme di predicati ed operatori speciali che vedremo in seguito.

Gli standard e varie implementazioni introducono diversi costrutti “Imperativi” per convenienza.

8.1 PROGRAMMAZIONE FUNZIONALE: INTERPRETI, AMBIENTI E COMPILATORI

In Java, C/C++ e vari linguaggi di programmazione la costruzione di programmi avviene via il processo di “compilazione”. In LISP e vari altri linguaggi di programmazione spesso si interagisce invece con un “ambiente” (spesso contenente il compilatore) la cui interfaccia principale è un interprete.

8.1.1 Espressioni in LISP

Una volta fatto partire l'interprete LISP si può procedere ad una esplorazione delle espressioni di base del linguaggio.

Utilizzeremo il common lisp come linguaggio.

Una prima cosa da notare è che in LISP ogni “espressione” denota un “valore”. Le espressioni più semplici sono numeri e stringhe. Possiamo vedere che succede inserendo queste espressioni nell'interprete:

```
prompt> 42
```

```
42
```

```
Prompt> "Sapete che cos'è '42'?"
```

```
"sapete che cos'è '42'?"
```

Prima di procedere facciamo la seguente osservazione: le operazioni aritmetiche elementari +, -, * e / non sono altro che funzioni.

Infatti, +(40, 2) denota il numero 42.

Questo tipo di notazione per le operazioni aritmetiche elementari si dice notazione prefissa.

In common lisp, la sintassi per le “chiamate” o “valutazioni” o “applicazione” di funzioni è molto semplice ma relativamente diversa dalla notazione tradizionale. Ogni espressione in common lisp ha la forma seguente

$$(f x_1 x_2 \dots x_N)$$

Le parentesi iniziali e finale sono obbligatorie e gli spazi (almeno uno) sono necessari per separare tra di loro la funzione e gli argomenti.

Una volta stabilita questa convenzione, la costruzione di espressioni più complicate a partire da più semplici è una cosa da ragazzi.

Le funzioni aritmetiche in LISP accettano un numero variabile di argomenti e si combinano secondo le classiche norme matematiche:

$$(+ 2 10 10 20) \equiv (+ 2 (* 2 10) 20)$$

Ovvero al posto di un valore, possiamo inserire un'espressione che lo denota.

Non vi sono ambiguità possibili nell'interpretazione poiché la funzione (anche detta operatore) è sempre il primo elemento dell'espressione.

Le espressioni possono essere complicate quanto si vuole, al fine di rendere più leggibili le espressioni più complicate di solito le si allinea in modo da avere gli argomenti (detti anche operandi) di una chiamata allineati verticalmente

Notiamo come le funzioni aritmetiche elementari + e * in common lisp rispettano i vincoli di "campo" algebrico.

```
>prompt (+)
```

```
0
```

```
>prompt (*)
```

```
1
```

Le funzioni aritmetiche elementari – e / in common lisp richiedono almeno un argomento e rappresentano in questo caso il "reciproco", sempre in senso algebrico.

LISP permette di utilizzare:

- Numeri
 - Interi: 42, -3
 - Virgola mobile: 0.5, 3.1415, 6.02E+21
 - Razionali: 3/2, -3/42
 - Complessi: #C (0 1)
- Booleani: T NIL
- Stringhe
- Operazioni su booleani: null, and, or, not
- Funzioni su numeri: +, -, /, *, mod, sin, cos, sqrt, tan, atan, plusp, <, >=, zerop

8.1.2 Ordine di valutazione

Data un'espressione LISP ($f x_1 x_2 \dots x_N$), la valutazione procede da sinistra verso destra a partire da x_1 fino a x_N producendo i valori v_1, \dots, v_N .

La funzione f viene "valutata" successivamente e viene applicata ai valori v_1, \dots, v_N .

Questa regola è inerentemente ricorsiva. Alcuni operatori speciali valutano gli argomenti in modo diverso.

8.1.3 Definizione di variabili e di funzioni

8.1.3.1 defparameter

In Common Lisp è possibile definire delle variabili usando l'operatore speciale defparameter

```
Prompt> (defparameter quarantadue 42)
```

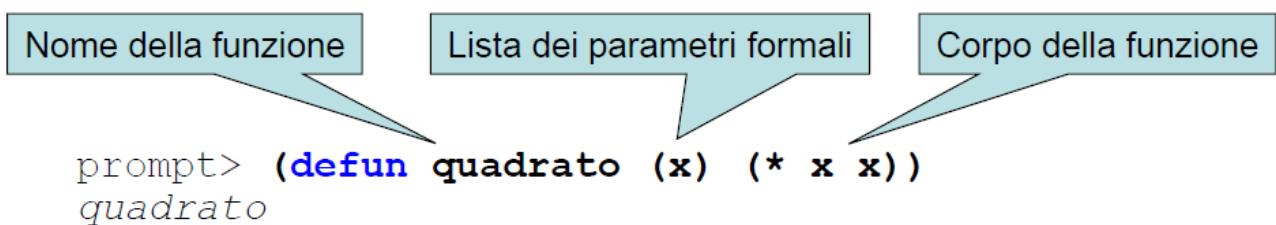
Quarantadue

Il simbolo quarantadue ha ora associato il valore 42.

La sintassi è molto semplice: l'operatore defparameter è seguito da un simbolo (un "identificatore" o "nome" e da un'espressione che viene valutata al fine di produrre un valore che viene associato al simbolo.

8.1.3.2 defun

Le funzioni si possono definire usando l'operatore speciale defun



La sintassi è molto semplice: l'operatore defun è seguito dal nome della funzione e da una lista di simboli – ovvero delimitati da parentesi – che rappresentano i nomi dei parametri formali della funzione; infine troviamo un'espressione (o una sequenza di espressioni) che costituisce il "corpo" della funzione.

L'operatore defun associa il corpo della funzione al nome nell'ambiente globale del sistema common lisp e restituisce come valore il nome della funzione.

Una volta definita, una funzione viene eseguita (o chiamata) usando la regola descritta precedentemente

```

prompt> (quadrato quarantadue)
1764
  
```

```

prompt> (- (quadrato (+ 20 22)) 10)
1754
  
```

```

prompt> (defun somma-di-quadrati (x y)
           (+ (quadrato x) (quadrato y)))
somma-di-quadrati
  
```

```

prompt> (- (somma-di-quadrati 6 3) 3)
42
  
```

8.1.4 Nomi in common lisp

In common lisp i nomi possono contenere il carattere ‘-‘, questo perché non c’è ambiguità con il segno ‘-‘, ovvero con la funzione ‘-‘.

In generale i simboli (nomi) LISP sono sequenze alfanumeriche aumentate con qualunque carattere, esclusi (,), #, ‘, “, : e la virgola

8.1.5 Valutazione di funzioni

La valutazione di funzioni avviene mediante la costruzione di activation frames.

I parametri formali di una funzione vengono associati ai valori (nb: si passa tutto per valore, non esistono effetti collaterali).

Il corpo della funzione viene valutato ricorsivamente tenendo conto di questi legami in maniera statica, ovvero bisogna tener presente cosa accade con variabili che risultano “libere” in una sotto-espressione.

Ad ogni sotto espressione del corpo si sostituisce il valore che essa denota.

Il valore restituito dalla funzione è il valore del corpo della funzione (che non è altro che una sotto-espressione).

```
Prompt> (defun doppio (n) (* 2 n))
```

```
Doppio
```

Estende l’ambiente globale con il legame tra doppio e la sua definizione

```
Prompt> (doppio 3)
```

```
6
```

Per la valutazione di una funzione F, l’ambiente deve eseguire i seguenti sei passi:

- 1) Mettere i parametri in un posto dove la procedura possa recuperarli
- 2) Trasferire il controllo della procedura
- 3) Allocare le risorse necessarie
- 4) Effettuare la computazione della procedura
- 5) Mettere i risultati in un posto accessibile al chiamante
- 6) Restituire il controllo al chiamante

L’activation frame contiene:

- Spazio per i registri da salvare dalla chiamata di una sotto funzione
- Spazio per l’indirizzo di ritorno
- Spazio per le variabili, definizioni locali e spazio per valori di ritorno
- Spazio per i valori degli argomenti
- Spazio per il riferimento statico
- Spazio per il riferimento dinamico
- Altri spazi eventuali

La gestione dello static link in common lisp è in realtà più complicato dato che il linguaggio ammette la creazione a runtime di funzioni che si devono ricordare il valore delle loro variabili libere al momento della loro creazione. Queste funzioni sono implementate con particolari strutture dati chiamate chiusure. Questa caratteristica del common lisp ci permette di creare delle funzioni anonime (operatore LAMBDA).

8.1.6 Funzioni anonime: espressioni LAMBDA

Una delle caratteristiche dei linguaggi funzionali (e del lisp in particolare) è la capacità di costruire funzioni anonime, l'operatore che ci permette di costruire funzioni anonime si chiama lambda.

L'operatore labda (il cui nome deriva dal λ -calcolo) denota una funzione anonima la cui sintassi è la seguente:

`(lambda (x1 x2 ... xn) <e>)`

Dove x₁ x₂ ... x_n sono dei simboli che rappresentano i parametri formali ed <e> è un'espressione. Queste espressioni sono chiamate lambda-espressioni.

Esempio

`(lambda (x) (+ 2 x))`

È la funzione che aggiunge 2 a x

`((lambda (x) (+ 2 x)) 40)`

È l'applicazione con x = 40.

Una lambda-expression può essere usata ovunque possiamo usare un nome di una funzione.

8.1.7 Operatori speciali: condizionali

Supponiamo di voler definire una funzione chiamata valore_assoluto che trasformi un numero nel suo valore assoluto, ovvero matematicamente

$$\text{valore_assoluto}(x) = \begin{cases} x & \text{se } x > 0 \\ 0 & \text{se } x = 0 \\ -x & \text{se } x < 0 \end{cases}$$

Questo tipo di definizione matematica viene detta “per casi”, ovvero si danno una serie di casi dove il valore finale della funzione dipende dalla verità o meno della condizione preliminare.

In lisp questo tipo di costrutto è disponibile tramite l'operatore speciale cond.

Cond ha la seguente sintassi:

`(cond (c1 e1) (c2 e2) ... (cM eM))`

Dove c₁, ..., c_M ed e₁, ..., e_M sono normali espressioni dove c rappresenta la condizione ed e la

(defun valore-assoluto (x) (cond ((> x 0) x) ((= x 0) 0) ((< x 0) (- x)))) conseguenza. Le parentesi sono obbligatorie che

delimitano le coppie di espressioni.

Date le funzioni <, =, >, la funzione valore-assoluto può quindi essere definita nel modo seguente:

`prompt> (= quarantadue 0)
NIL`

le funzioni <, =, > sono dei “predicati” che ritornano i valori “vero” o “falso”; la costante T rappresenta il valore di verità “vero”, la costante NIL rappresenta il valore di verità “falso”.

`prompt> (> quarantadue 0)
T`

L'operatore `cond` viene valutato in maniera speciale. Ogni coppia $(c_j \ e_j)$ viene considerata in ordine:

- Se c_i ha valore T allora il valore ritornato dalla valutazione dell'operatore `cond` è il valore ottenuto dalla valutazione di e_j
- Altrimenti la valutazione considera la coppia successiva
- Se non vi sono più coppie allora `cond` produce il valore `NIL`

```
(defun valore-assoluto (x) Naturalmente la funzione valore_assoluto può essere
  (cond ((> x 0) x)           definita anche come espresso qua accanto.
        (T (- x))))
```

Questo tipo di definizioni è così comunque che il common lisp offre l'abbreviazione `if`, con la sintassi

```
(if c e1 e2)
```

dove e_1 è la conseguenza per il valore T e e_2 per `NIL`.

```
(defun valore-assoluto (x) Come in ogni linguaggio, anche il common lisp ha a
  (if (> x 0) x (- x))) disposizione i soliti operatori booleani, che appaiono a
                                         tutti gli effetti come delle funzioni
```

| | |
|---|----------------------------------|
| (and c ₁ c ₂ ... c _K) | prompt> (and (> 42 0) (< -42 0)) |
| (or d ₁ d ₂ ... d _M) | T |
| (not e) | prompt> (not (> 42 0)) |
| | NIL |

La loro semantica è intuibile ed è espressa qua accanto.

8.1.8 Operatori speciali: booleani

`prompt> (and)` Anche in questo caso vengono rispettate alcune relazioni fondamentali
 T

`prompt> (or)`
`NIL`

8.2 FUNZIONI RICORSIVE

```
(defun fattoriale (n)
  (if (= n 0)
      1
      (* n (fattoriale (- n 1)))))
```

Il fattoriale è un classico esempio di funzione ricorsiva.

Questa funzione calcola il fattoriale di un numero utilizzando una sequenza di valori intermedi che devono essere salvati da qualche parte (ovvero sullo "stack" di attivazione di ogni chiamata ricorsiva).

```
(defun fib (n)
  (cond ((= n 0) 1)
        ((= n 1) 1)
        (T (+ (fib (- n 2)) (fib (- n 1))))))
))
```

Un altro tipico caso è la serie di Fibonacci e il calcolo dell' n -esima cifra.

Ci sono più modi per scrivere una funzione ricorsiva:

```
(defun fatt-ciclo (n acc)
  (if (= n 0)
      acc
      (fatt-ciclo (- n 1) (* n acc)))))

(defun fattoriale (n)
  (fatt-ciclo n 1))
```

un nuovo record di attivazione. Queste funzioni vengono dette funzioni “tail-ricorsive”.

La funzione fib non può essere direttamente riscritta in modo tail-ricorsivo poiché richiede la combinazione di due chiamate ricorsive.

8.3 STRUTTURE DATI E FUNZIONI

Supponiamo di voler costruire una piccola libreria per altri calcoli con numeri razionali. Innanzitutto, assumiamo di aver a disposizione una funzione che costruisce una rappresentazione di un numero razionale: (crea_razionale n d) → <il razionale n/d> dove n è il numeratore e d il denominatore.

Assumiamo anche di avere due funzioni, numer e denom, che estraggono rispettivamente il numeratore n ed il denominatore d dalla rappresentazione di un numero razionale.

Costruire la libreria a questo punto è semplice.

La libreria è la seguente alla quale mancano alcune funzioni.

```
(defun somma-raz (r1 r2)
  (crea-razionale (+ (* (numer r1) (denom r2))
                     (* (numer r2) (denom r1)))
                  (* (denom r1) (denom r2)))))

(defun molt-raz (r1 r2)
  (crea-razionale (* (numer r1) (numer r2))
                  (* (denom r1) (denom r2))))

(defun -=raz (r1 r2)
  (= (* (numer r1) (denom r2))
     (* (numer r2) (denom r1))))
```

8.4 LE CONS-CELLS E LA FUNZIONE CONS

Una delle strutture dati più importanti in LISP è la cosiddetta con-cell. Una cons-cell è semplicemente una coppia di puntatori a due elementi.

in questo caso la funzione fattoriale riutilizza uno degli argomenti come un accumulatore. La funzione fatt-ciclo non è altro che dei cicli in incognito. Questo tipo di funzioni ricorsive è particolare poiché un compilatore può ottimizzare la chiamata ricorsiva con un'operazione di JUMP, senza creare

Le cons-cells sono create dalla funzione cons, che si preoccupa di allocare la memoria necessaria al mantenimento della struttura.⁹

```
prompt> (defparameter c (cons 40 2))  Cons : <oggetto LISP> x <oggetto LISP> → <cons-cell>
c
```

```
prompt> (car c)
40
```

```
prompt> (cdr c)
2
```

```
(defun crea-razionale (n d)
  (cons n d))
```

Data la primitiva cons, le funzioni crea-razionale, numer e denom diventano molto semplicemente come indicate qua accanto.

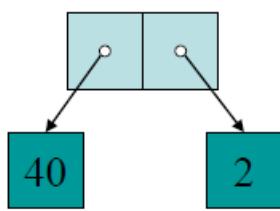
```
(defun numer (r) (car r))
```

Esempio

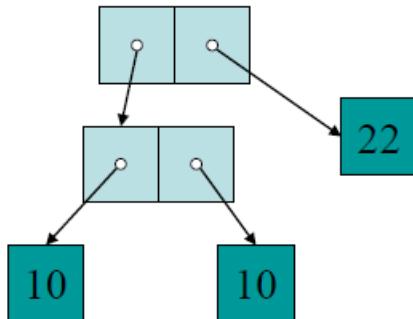
```
(defun denom (r) (cdr r))
```

(denom (crea-razionale 42 7)) → 7

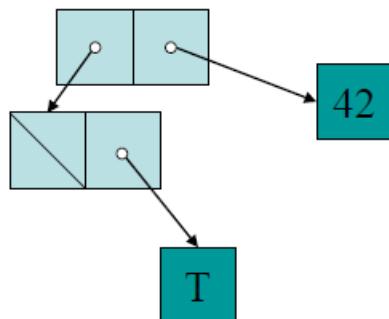
```
(cons 40 2)
```



```
(cons (cons 10 10) 22)
```



```
(cons (cons NIL T) 42)
```



La funzione cons genera in memoria dei grafi di puntatori arbitrariamente complessi. Questi grafi vengono rappresentati in una tradizionale notazione detta box-and-pointer.

I valori T, NIL e stringhe sono valori quanto altri.

notare la rappresentazione

Come risponde il sistema LISP quando si richiama la funzione cons?

```
prompt> (cons NIL T)
(NIL . T) ; Notazione "dotted-pair"
; ("coppia-puntata"). Gli spazi
; sono significativi.
```

```
prompt> (cons 4 2)
(4 . 2)
```

```
prompt> (cons "foo" 42)
("foo" . 42)
```

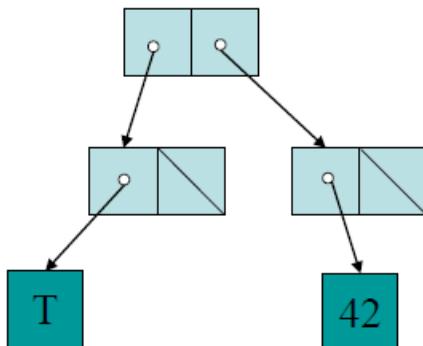
La notazione “dotted pair” è l’unica “irregolarità sintattica infissa” in LISP.

Quando si parla di liste in LISP ci si riferisce a particolari configurazioni di cons-cells dove l’ultimo puntatore è NIL, la costante NIL è equivalente alla lista vuota ().

⁹ Cons sta a LISP come new a Java

```
(cons (cons T NIL) (cons 42 NIL))
```

una lista LISP corrispondente a questa configurazione di cons-cells viene rappresentata tipograficamente come:



((T) 42)

che è equivalente a

((T . NIL) . (42 . NIL))

In altre parole una cons-cell con NIL come secondo elemento (ovvero come cdr) viene stampata senza punto ed il NIL.

Inoltre una cons-cell con una cons-cell come secondo elemento è stampata in modo "abbreviato"

```
prompt> (cons 42 (cons 123 666))
(42 123 . 666)      ; Equivalente a (42 . (123 . 666))
```

```
prompt> (cons 42 (cons "foo bar" nil))
(42 "foo bar")
;;; To', una 'lista'.
```

8.5 LISTE E LA FUNZIONE LIST

La funzione cons può quindi essere usata per rappresentare sequenze (liste) di oggetti.

La definizione

```
(defparameter L (cons 1 (cons 2 (cons 3 (cons 4 NIL)))))
```

genera in memoria una sequenza di cons-cells tale che (car (cdr (cdr L))) → 3.

La costruzione è così utile da meritare un nome particolare:

```
(defparameter L (list 1 2 3 4))
```

La funzione list accetta un numero variabile di argomenti, le liste vengono stampate in maniera abbreviata:

```
(list -1 0 1 2 3) → (-1 0 1 2 3)
```

Attenzione! L'espressione (list 1 2 3) e la lista (1 2 3) non vanno confuse: la prima è un'espressione a tutti gli effetti, la seconda è la rappresentazione tipografica di una struttura dati in memoria, che non ha un operatore nella posizione canonica.

8.6 LISTE

```
(defun list-ref (n list)
  (if (<= n 0)
      (car list)
      (list-ref (- n 1) (cdr list))))
```

Ora che possiamo costruire varie liste abbiamo anche la possibilità di manipolarle. Come si estraе l'n-esimo elemento da una lista? Tramite una funzione ricorsiva, così come possiamo

```
(defun lunghezza (l)
  (if (null l)
      0
      (+ 1 (lunghezza (cdr l)))))
```

calcolare la lunghezza della lista dove la funzione null ritorna il valore T se l'argomento passatole è il valore NIL.

8.6.1 Elementi di una lista

- nth: La funzione list-ref è definita nello standard common lisp con il nome nth.
- cdr/rest: La funzione cdr ritorna di fatto il “resto” di una lista e lo standard common lisp offre il sinonimo rest.
- car/first: La funzione car ritorna il primo elemento di una lista e lo standard common lisp offre il sinonimo first.
- Lo standard common lisp ha in libreria le funzioni second, third, ..., tenth.
- Per concatenare due liste si utilizza la funzione “append”

Appendi è una funzione ricorsiva e presenta una tipica forma di ricorsione “strutturale” sul resto di una lista (detta anche “cdr-recursion” o “rest-recursion”).

8.7 DATI SIMBOLICI E OPERAZIONE QUOTE

Tutti gli esempi che abbiamo visto fino ad ora hanno utilizzato numeri (soprattutto interi) e stringhe. Il lisp ci permette di costruire delle liste come

(a b c d e f g)

((gino 10) (ugo 10) (maria 2) (ettore 20))

Come possiamo costruire una lista che contiene i due simboli A e B?

Date le regole di valutazione degli argomenti dell'espressione (list A B), il sistema cerca di trovare il valore associato all'identificatore A, non lo trova e segnala un errore. Abbiamo bisogno di un operatore che dica al LISP di non procedere alla valutazione di una (sotto)espressione. L'operatore in questione è quote.

prompt> (quote A)
A

Quote è un altro operatore speciale (come cond, if, ecc.) la cui sintassi è

prompt> (quote (1 B 3))
(1 B 3)

(quote <e>)

L'espressione <e> non viene valutata e viene ritornata letteralmente.

prompt> (quote (1 2 (3 4) (5 6 "foo")))
(1 2 (3 4) (5 6 "foo"))

Quote è talmente importante che il LISP fornisce una comoda abbreviazione che utilizza un singolo carattere:

prompt> 'A
A

'<e> equivale a (quote <e>)

prompt> '(1 B 3)
(1 B 3)

L'operatore quote non sembra agire su numeri e stringhe.

prompt> '(1 2 (3 4) (5 6 "foo"))
(1 2 (3 4) (5 6 "foo"))

Ciò accade perché numeri e stringhe sono autovalutanti, ovvero essi sono espressioni

il valore ha la stessa rappresentazione tipografica dell'espressione tipografica che li denota.

```
prompt> (list 'A 'B)
(A B)
```

```
prompt> '(A B)
(A B)
```

```
prompt> '(defun media (x y) (/ (+ x y) 2.0))
(DEFUN MEDIA (X Y) (/ (+ X Y) 2.0))
```

```
prompt> (defun media (x y) (/ (+ x y) 2.0))
MEDIA
```

```
prompt> (defparameter m
          '(defun media (x y) (/ (+ x y) 2.0)))
M
```

```
prompt> (third m) ; (car (cdr (cdr m)))
(X Y)
```

Conseguenze:

- Il programma Hello World è cortissimo.

```
prompt> "Hello world!"
"Hello world!"
```

- In LISP i programmi ed i dati sono esattamente la stessa cosa.

8.8 SIMBOLI, NUMERI, LISTE E “ATOMI”

```
prompt> (atom 3)
T
```

```
prompt> (atom 'un-simbolo)
T
```

```
prompt> (atom "una stringa")
T
```

```
prompt> (atom (cons -42 42))
NIL
```

```
prompt> (atom (list 1 2 3))
NIL
```

- Cons-cells (quindi liste)
- Altri oggetti di base

In LISP abbiamo una ripartizione degli oggetti principali in due categorie: atomi e cons-cells:

- Gli atomi sono simboli, numeri, stringhe, ecc.
- I non-atomi sono le cons-cells (quindi le liste)

Esiste un predicato che controlla se il suo argomento è un atomo o meno: atom.

8.8.1 Simboli e liste (e altri elementi), ovvero le symbolic expressions

Dunque in Lisp abbiamo:

- Numeri
- Simboli
- Stringhe

Le cons-cell, i numeri, i simboli e le stringhe (ma non solo) costituiscono le symbolic expressions, dette anche Sexp's.

I simboli (o identifieri) denotano invece i valori che sono loro associati (ad esempio tramite defparameter).

Le liste, se non quotate, rappresentano invece delle espressioni da valutare, ergo la necessità di avere l'operatore di quote.

8.9 VALUTAZIONE DI ESPRESSIONI E FUNZIONI RICORSIVE

8.9.1 Dettagli e funzione eval

Dato che programmi e sexp's in lisp sono equivalenti, possiamo dare le seguenti regole di valutazione ed implementarle nella funzione eval.

Data una sexp:

- Se è un atomo
 - se è un numero ritorna il suo valore
 - se è una stringa la ritorna così com'è
 - se è un simbolo
 - estrae il suo valore dall'ambiente corrente e lo ritorna
 - se non esiste un valore associato allora la segna come un errore
- se è una cons-cell ($O A_1 A_2 \dots A_n$):
 - se O è un operatore speciale, allora la lista ($O A_1 A_2 \dots A_n$) viene valutata in modo speciale
 - se O è un simbolo che denota una funzione nell'ambiente corrente, allora questa funzione viene applicata (apply) alla lista ($VA_1 VA_2 \dots VA_n$) che raccoglie i valori delle valutazioni delle espressioni $A_1 A_2 \dots A_n$
 - se O è una lambda expression la si applica alla lista ($VA_1 VA_2 \dots VA_n$) che raccoglie i valori delle valutazioni delle espressioni $A_1 A_2 \dots A_n$
 - altrimenti si segnala un errore

eval applica la funzione al suo argomento, definendo i legami di N “in cascata” e costruendo una espressione che alla fine verrà valutata, a partire dalla sotto-espressione più annidata.

```
(defun fatt (n)
  (if (zerop n) 1 (* n (fatt (- n 1)))))

prompt> (fatt 3)
          n diventa associato a 3 (“bound to” 3):

(eval '(fatt 3))
→ (eval '(if (zerop 3) 1 (* 3 (fatt (- 3 1)))))

... → (eval '(* 3 (fatt 2)))
      ... → (eval '(*3 (* 2 (fatt 1))))
          ... → (* 3 (* 2 (* 1 (fatt 0))))
              ... → (* 3 (* 2 (* 1 1)))
... → ... 6
```

8.9.2 Esempi con liste

La struttura ricorsiva delle liste si presta molto bene alla programmazione ricorsiva.

Metodo:

- Scrivere il valore della funzione nel caso banale (usualmente la lista vuota)
- Ridursi ricorsivamente al caso base operando su un argomento ridotto

Specificare la funzione:

- Sommatoria degli elementi di una lista (si assume che gli atomi siano numeri interi)
 - Sum: list → integer

Specificare la soluzione in modo ricorsivo:

- Se la lista è vuota: lista = () → 0
- Altrimenti lista = L → first(L) + sum(rest(L))

Implementazione:

```
(defun sum (l)
  (if (null l)
      0
      (+ (first l) (sum (rest l)))))
```

Esempio: lunghezza di una lista

(lung '(a b c)) → 3

(lung '(((a) b) c) → 2

(lung nil) → 0

Se la lista è vuota allora la lunghezza è 0, altrimenti è 1 + la lunghezza del resto della lista

```
(defun lung (l)
  (if (null l)
      0
      (+ 1 (lung (rest l)))))
```

8.9.3 Last

Last è una funzione che ritorna l'ultimo elemento di una lista. In common lisp ha una semantica leggermente diversa.

```
(defun last-l (l)
  (cond ((null l) nil)
        ((atom l) ; Notare questo caso.
         (error "L'argomento non e` una lista"))
        ((null (rest l)) (first l))
        (t (last-l (rest l)))))
```

8.9.4 Ricorsione semplice e doppia

Ricorsione semplice (“ricorsione cdr”): la ricorsione è sempre definita sul resto di una lista.

Non è sempre sufficiente, in certi casi serve una ricorsione doppia (“ricorsione car-cdr”), ovvero anche sul primo elemento di una lista.

8.9.4.1 Count-atoms

Conta gli atomi di una lista

(count-atoms '((a b c) 1 (xyz d))) → 6

Base:

- $() \rightarrow 0$
- Atomo $\rightarrow 1$

Due casi, perché in questo caso sexp può essere sia lista sia atomo.

Ipotesi ricorsiva:

- $(\text{count-atoms}(\text{rest } L)) \rightarrow \text{numero atomi di rest di } L$

Passo:

- Se $(\text{first } L)$ è un atomo $\rightarrow (+ 1 (\text{count-atoms}(\text{rest } L)))$
- Altrimenti...

Utilizziamo invece la ricorsione doppia.

$(\text{first } L)$ e $(\text{rest } L)$ sono sottostrutture di L , possiamo usare l'ipotesi ricorsiva:

- $(\text{count-atoms}(\text{first } L))$ e $(\text{count-atoms}(\text{rest } L))$ contano correttamente i loro atomi

Passo:

- $(+ (\text{count-atoms}(\text{first } L)) (\text{count-atoms}(\text{rest } L)))$

Quindi si ricorre sia sul car che sul cdr della lista:

```
(defun count-atoms (x) ; 'x' è una sexp.
  (cond ((null x) 0)
        ((atom x) 1)
        (t (+ (count-atoms (first x))
               (count-atoms (rest x))))))
```

8.9.4.2 Profondità

Profondità massima di annidamento di una sexp (ovvero numero massimo di parentesi aperte e non chiuse):

- Atomo $\rightarrow 0$
- $() \rightarrow 1$
- $(a (b ((c)) (d))) \rightarrow 4$

Base:

- Ovvia (caso banale di atomo o di lista vuota)

Ipotesi ricorsiva:

- $(\text{prof} (\text{first } x)), (\text{prof} (\text{rest } x))$

Passo:

- Prof di x è il massimo (max) fra
 $(\text{prof} (\text{first } x)) + 1$

e
 (prof (rest x))

```
(defun prof (x)
  (cond ((null x) 1)
        ((atom x) 0)
        (t (max (+ 1 (prof (first x)))
                  (prof (rest x)))))))
```

La funzione max è predefinita in common lisp.

8.9.4.3 *Flatten*

Serve ad appiattire una lista.

```
(flatten '(((a (b (c d)) e) f)) → (a b c d e f)
(flatten 'a) → (a)
```

Base:

- Lista vuota, non cambia
- Atomo → (list atomo)

Ipotesi ricorsiva:

- L = (e1 ... en): (mirror (first L)), (mirror (rest L))

Passo:

append di

- (mirror (rest l)) con
- (list (mirror (first x)))

8.9.4.4 *Mirror*

Immagine speculare di una sexp

```
(defun mirror (x) ; x è` una sexp
  (if (atom x)
      x
      (append (mirror (rest x))
              (list (mirror (first x)))))))
```

NB: senza list, append toglierebbe una parentesi al mirror del first se questo è una lista, darebbe errore al parametro se fosse un atomo.

8.9.4.5 *Inverti*

Inversione di una lista

Data L = (e1 ... en), inverti L → (en ... e1)

Base:

- (inverti NIL) → NIL

- $(\text{inverti} (\text{rest } L)) \rightarrow (\text{en} \dots e2)$
- $e1$ va posto in coda a $(\text{en} \dots e2)$

supponiamo che ci sia una funzione cons-end che faccia quest'ultima operazione:

$(\text{cons-end } S \ L) \rightarrow (e1 \dots en \ S)$ e poi definiamo cons-end (approccio alla programmazione top-down).

```
(defun inverti (x)
  (cond ((null x) x)
        ((atom x) x) ; anche Sexp...
        (T (cons-end (first x)
                      (inverti (rest x))))))
```

8.9.4.6 circulate

scrivere la funzione circulate in modo che operi come segue al primo livello:

```
prompt> (circulate '(1 2 3 4) 'left)
(2 3 4 1)
```

```
prompt> (circulate '(1 2 3 4) 'right)
(4 1 2 3)
```

```
(defun circulate (lst direction)
  (cond ((atom lst) lst)
        ((null lst) nil)
        ((eq direction 'left)
         (append (cdr lst) (list (car lst)))))
        ((eq direction 'right)
         (cons (last-1 lst) (but-last lst)))
        (T lst) ))
```

8.10 FUNZIONI DI UGUAGLIANZA

```
prompt> (eq 42 42)
T
```

```
prompt> (eq 42 3)
NIL
```

```
prompt> (eq 'quarantadue 'quarantadue)
T
```

```
prompt> (eq 'quarantadue quarantadue)
NIL
```

```
prompt> (eq 'quarantadue 'fattoriale)
NIL
```

```
prompt> (eq '(42) '(42))
NTI,
```

Un problema preliminare che si pone in LISP è quello di controllare se due oggetti sono uguali, il common lisp mette a disposizione vari predicati di uguaglianza con semantica diversa.

Vediamo i due più importanti: eql e equal.

8.10.1 Eq1

Il predicato eq1 viene usato per controllare l'uguaglianza di simboli e numeri.

8.10.2 Equal

Il predicato equal si comporta come eq1. Ma è in grado di controllare se due liste sono uguali; in

pratica non fa altro che applicare eq1 ricorsivamente a tutti gli atomi di una lista: se un'applicazione di eq1 ritorna il valore NIL allora equal fa lo stesso, altrimenti viene ritornato il valore T.

```
prompt> (equal 42 42)
T
```

```
prompt> (equal 42 3)
NIL
```

```
prompt> (equal 'quarantadue 'quarantadue)
T
```

```
prompt> (equal '(1 2 3 (a s d) 4) '(1 2 3 (a s d) 4))
T
```

```
prompt> (equal '(1 due tre (a s d) 4) '(1 2 3 (a s d) 4))
NIL
```

8.11 LISTE E FUNZIONI

Ora che abbiamo modo di costruire quello che è un contenitore basilare (le liste) ed abbiamo l'operazione di quote, possiamo cominciare a vedere veramente quali sono i vantaggi di un paradigma funzionale.

Prendiamo la lista (defparameter pari (list 2 4 6 8 10))

Supponiamo di voler moltiplicare tutti gli elementi della lista per un certo valore e di ritornare una nuova lista con i nuovi elementi; questa funzione è semplicemente

```
(defun scala-lista (l fattore)
  (if (null l)
      nil
      (cons (* fattore (car l))
            (scala-lista (cdr l) fattore))))
```

Si noti che la funzione scala-lista ripete la struttura di appendi. L'operazione fatta dalla funzione scala-lista si può astrarre se astraiamo il concetto di valore funzionale.

L'astrazione "applica la funzione f a tutti gli elementi della lista L e ritorna una lista dei valori" è nota come "map"; in common lisp la funzione mapcar svolge questo compito.

La funzione mapcar è predefinita, può essere scritta come

```
(defun mapcar* (funzione lista)
  (if (null lista)
      nil
      (cons (funcall funzione (car lista))
            (mapcar* funzione (cdr lista)))))
```

L'asterisco in mapcar* viene utilizzato per evitare errori nell'ambiente common lisp.

La funzione funcall serve invece per chiamare una funzione con un certo argomento.

Supponiamo di avere una serie di funzioni chiamate scala-4, scala-10, scala-pi, ecc.

```
(defun scala-4 (x) (* x 4))
(defun scala-10 (x) (* x 10))
(defun scala-pi (x) (* x pi))
```

La funzione scala-lista-10 può essere scritta come

```
(defun scala-lista-10 (lista)
  (mapcar 'scala-10 lista))
```

Ovviamente questo ci dice che possiamo astrarre la funzione scala-lista-x in modo abbastanza semplice:

```
(defun scala-lista (lista funzione-scalante)
  (mapcar funzione-scalante lista))
```

Il parametro funzione-scalante è associato alla funzione che ci interessa; ad esempio per scalare di un fattore 10 ora possiamo scrivere:

```
prompt> (scala-lista (list 1 2 3) 'scala-10)
(10 20 30)
```

8.12 FUNZIONI ANONIME ED OPERATORE LAMBDA

L'esempio precedente è semplice ma è interessante per un motivo: sarebbe bene poter costruire delle funzioni ausiliarie ogni volta che ce ne fosse bisogno

Come abbiamo già visto, in LISP è possibile definire delle funzioni anonime a questo scopo utilizzando l'operatore speciale lambda.

Con l'operatore lambda possiamo creare tutte le funzioni che vogliamo senza assegnare loro un nome.

```
prompt> (lambda (x) (+ x 42))
#<funzione>
```

```
prompt> ((lambda (x) (+ x 42)) 42)
84
```

```
prompt> (scala-lista '(1 2 3)
                      (lambda (x) (* x 3)))
(3 6 9)
```

```
prompt> (defun adder-x (x)
                      (lambda (y) (+ x y)))
adder-x
```

```
prompt> (defparameter adder-42 (adder-x 42))
adder-42
```

```
prompt> adder-42
```

```
prompt> (funcall adder-42 42)
84
```

Possiamo anche creare funzioni che costruiscono delle funzioni e le ritornano come valori.

Dato l'operatore lambda, possiamo riscrivere la funzione scala-lista in maniera più elegante.

8.13 OPERATORE LAMBDA ED OPERATORE LET

Consideriamo la seguente funzione

$$F(x, y) = x(1 + xy)^2 + y(1 - y) + (1 - y)(1 + xy)$$

Usando l'operatore lambda possiamo costruire una serie di valori intermedi da riutilizzare.

```
(defun f (x y)
  ((lambda (a b)
    (+ (* x (quadrato a))
       (* y b)
       (* a b)))
   (+ 1 (* x y))
   (- 1 y)))
```

ovvero la funzione anonima viene chiamata con due argomenti che rappresentano i valori intermedi da riutilizzare.

Questo tipo di chiamate a funzioni anonime è così utile da essere stato ri-codificato con un nuovo operatore speciale: let.

Usando l'operatore let, la funzione precedente diventa:

```
(defun f (x y)
  (let ((a (+ 1 (* x y)))
        (b (- 1 y)))
    )
  (+ (* x (quadrato a))
     (* y b)
     (* a b))))
```

ovvero, l'operatore let ci permette di introdurre dei nuovi nomi (variabili) locali da poter riutilizzare all'interno di una procedura; la sua sintassi è:

`(let ((n1 e1) (n2 e2) ... (nk ek)) espressione)`

Ad esempio

```
(let ((a 40) (b (+ 1 1))) (+ a b)) → 42
```

8.14 TIPICHE FUNZIONI DI ORDINE SUPERIORE

Le funzioni che prendono una o più funzioni come argomenti sono dette funzioni di ordine superiore.

La loro esistenza è al cuore del paradigma funzionale di programmazione.

Finora abbiamo visto e vedremo:

- Mapcar
- Compose
- Filter (in common lisp varianti di remove e delete)
- Fold (in common lisp reduce)
- Complement

8.14.1 Funzione compose

```
(defun compose (f g)
  (lambda (x)
    (funcall f (funcall g x))))
prompt> (funcall (compose 'first 'rest)
                  '(1 2 3 4 5))
```

2

La funzione compose corrisponde alla nozione matematica di composizione di funzioni.

La semantica della funzione è la seguente: date due funzioni (di un solo argomento) f e g come argomenti, ritorna una nuova funzione che corrisponde alla composizione f(g(x)).

8.14.2 Funzione filter

La funzione filter rimuove gli elementi della lista che non soddisfano il predicato

```
(defun filter (predicato lista)
  (cond ((null lista) nil)
        ((funcall predicato (car lista))
         (cons (car lista)
               (filter predicato (cdr lista))))
        (T (filter predicato (cdr lista))))))
```

```
prompt> (filter 'oddp '(1 2 3 4 5))
(1 3 5)
```

Dove oddp è una funzione built-in di LISP che riconosce i numeri dispari.

8.14.3 Funzione accumula

La funzione accumula (detta anche fold o reduce) applica una funzione ad un elemento di una lista ed al risultato (ricorsivo) dell'applicazione di accumula al resto della lista.

```
(defun accumula (f iniziale lista)
  (if (null lista)
      iniziale
      (funcall f (car lista)
                (accumula f iniziale (cdr lista)))))
```

```
prompt> (accumula '+ 0 '(1 2 3))
6
```

```
prompt> (accumula '* 1 '(1 2 3 4))
24
```

```
prompt> (accumula 'cons NIL '(1 2 3))
(1 2 3)
```

Consideriamo la seguente funzione:

```
(defun iota (n)
  (if (= n 0)
      nil
      (cons n (iota (- n 1)))))
```

quindi
 $(\text{iota } 4) \rightarrow (4 \ 3 \ 2 \ 1).$

Grazie alla funzione accumula possiamo riscrivere la funzione

fattoriale (quasi).

```
(defun fattoriale (n)
  (if (zerop n) 1 (accumula '* 1 (iota n))))
```

8.15 UTILI VARIAZIONI SUL TEMA

```
cl-prompt> :foo  
:foo
```

Certi simboli in common lisp hanno un'interpretazione particolare. I simboli i cui nomi iniziano con due punti (:) sono detti keywords ed hanno sé stessi come valore.

```
cl-prompt> :forty-two  
:forty-two
```

Le keywords sono usate estensivamente in common lisp e ci servono essenzialmente per definire delle funzioni con una sintassi di chiamata più interessante di quella semplice.

Come abbiamo visto esistono funzioni in common lisp che prendono una sequenza variabile di argomenti.

Ovviamente si possono definire delle funzioni con questo comportamento: basta usare la seguente sintassi nella lista di argomenti con cui si definisce una funzione. Questa lista di argomenti è detta lambda-list.

Indicatore di lista variabile di argomenti **Parametro contenente la lista degli argomenti**

```
(defun foo (a b c &rest l) (append l (list a b c)))
```

In common lisp vi sono anche funzioni che prendono dei parametri opzionali. Ovviamente si possono definire delle funzioni con questo comportamento: basta usare la seguente sintassi nella lista degli argomenti con cui si definisce una funzione.

Indicatore di argomento opzionale

Parametro opzionale

```
(defun foo (a &optional o) (cons a o))
```

Esempio

```
(subseq "qwerty" 2) → "erty"
```

```
(subseq "qwerty" 1 4) → "wer"
```

I parametri opzionali possono essere inizializzati con un valore di default

```
(defun fattoriale (n &optional (acc 1))
  (if (zerop n) acc (fattoriale (- n 1) (* n acc))))
```

```
cl-prompt> (fattoriale 4)
24
```

```
cl-prompt> (fattoriale 4 2)
48
```

In common lisp si possono definire delle funzioni che utilizzano i loro parametri associandoli a dei nomi, ovvero delle keywords. Molte funzioni standard hanno questo comportamento.

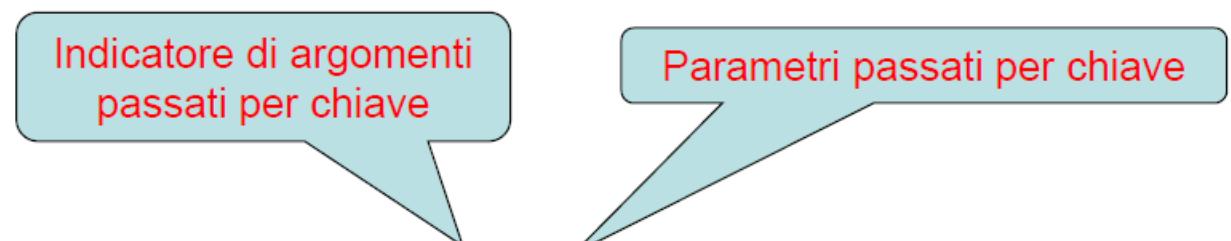
```
cl-prompt> (find 2 '(1 2 3 4 5 6))
2
```

```
cl-prompt> (find 2 '(1 2 3 4 5 6) :start 3)
NIL
```

```
cl-prompt> (find 2 '(1 2 3 4 5 6) :end 3 :start 1)
2
```

```
cl-prompt> (sort '((a 1) (q 2) (d 3) (f 4)) 'string>
                  :key 'first)
((Q 2) (F 4) (D 3) (A 1))
```

Ovviamente si possono definire delle funzioni che accettano parametri a “chiave”: basta usare la seguente sintassi nella lista di argomenti con cui si definisce una funzione:



```
(defun make-point (&key x y)
  (list x y))
```

```
cl-prompt> (make-point)
(nil nil)
```

```
cl-prompt> (make-point :y 42)
(nil 42)
```

```
cl-prompt> (make-point :y 42 :x -123)
(-123 42)
```

Ovvero ogni parametro passato a chiave diventa una keyword da poter utilizzare al momento della chiamata (anche i parametri passati “a chiave” possono avere un valore di default).

Parametri opzionali, a chiave e variabili vanno sempre dichiarati dopo quelli obbligatori.

8.16 INPUT/OUTPUT IN COMMON LISP

Le due funzioni principali del common lisp per la gestione dell’I/O sono READ e PRINT.

Ad esse va associata la gestione dei files e streams di I/O.

8.16.1 Read

La funzione read fa molto di più di una semplice lettura. L’esempio seguente dovrebbe essere convincente:

```
prompt> (read)
(foo 41 (42) 43 45) ; READ aspetta un input.
(FOO 41 (42) 43 45) ; Valore ritornato da READ
```

```
prompt> (third (read))
(foo 41 (42) 43 45) ; READ aspetta un input.
(42)
```

Ovvero read legge un intero oggetto LISP, riconoscendone la sintassi.

8.16.2 Print

```
prompt> (print 42)
42    ; PRINT stampa la rappresentazione di 42
      ; (preceduta da un a-capo).
42    ; L'ambiente Common Lisp stampa a video il
      ; valore dell'applicazione della funzione
      ; PRINT al valore 42.
```

```
prompt> (print "HELLO WORLD!")
```

```
"HELLO WORLD!"
"HELLO WORLD!"
```

```
prompt> (print (sqrt -1))
```

```
#C(0.0 1.0)
#C(0.0 1.0)
```

La funzione print stampa un oggetto LISP rispettandone la sintassi. Il valore ritornato da print è il valore dell'oggetto la cui rappresentazione tipografica è appena stata stampata.

8.16.3 Output

Ovviamente è bene avere a disposizione dei metodi per poter stampare un po' più agevolmente. Il common lisp mette a disposizione la funzione format (simile alla printf di C). FORMAT è complessa, alcuni esempi semplici sono:

```
prompt> (format t "Il fattoriale di ~D e` ~D~%" 3 (fact 3))
Il fattoriale di 3 e` 6
NIL
```

la prima linea è la stampa della stringa “formattata”; il NIL è il valore ritornato da FORMAT

```
prompt> (format t "~S + ~S = ~S~%" '(1) '(2) (append '(1) '(2)))
(1) + (2) = (1 2)
NIL
```

Le direttive nella stringa da formattare sono introdotte dal carattere ~.

| | |
|----|---|
| ~D | Interi |
| ~% | A capo |
| ~S | Stampa un oggetto LISP secondo la sua sintassi standard |
| ~A | Stampa un oggetto LISP secondo una sintassi esteticamente piacevole |

```
prompt> (format t "~S e` una stringa!~%" "foo")
"foo" e` una stringa!
NIL
```

```
prompt> (format t "~A forse non e` una stringa!~%" "foo")
foo forse non e` una stringa!
NIL
```

8.16.4 Output formattato

Che cosa è il T che appare come primo argomento a format? È l'indicazione di “dove” andare a stampare; nella fattispecie su “standard output”.

8.16.5 Streams common lisp

Il common lisp ha sempre a disposizione almeno tre streams standard:

- Standard input
- Standard output
- Standard error

I tre streams sono i valori associati alle tre variabili:

- *standard-input*
- *standard-output*
- *error-output*

Esempio

```
(format *standard-output* "~S e' non una stringa!~%" "foo") → "foo" non è una stringa! NIL
```

Il T passato al posto di *standard-output* è una comodità.

```
prompt> (read *standard-input*)
qwerty
QWERTY

prompt> (print '(42 + 2 = 44 gatti) *error-output*)
(42 + 2 = 44 gatti)
(42 + 2 = 44 gatti)
```

Le funzioni READ, PRINT e FORMAT accettano un numero variabile di argomenti. Uno di questi è uno stream (di output per FORMAT e PRINT e di input per READ).

La manipolazione dei files in common lisp è relativamente complicata ma simile a quella di Java.

Per leggere e scrivere da e su un file si usa la macro “with-open-file”¹⁰.

```
(with-open-file (<var> <file> :direction :input) <codice>

(with-open-file (<var> <file> :direction :output) <codice>)
```

La variabile <var> viene associata allo stream aperto sul <file> e può venire utilizzata all'interno di <codice>.

Ora alcuni empi di scrittura e lettura sul file “foo.lisp” (nella cartella corrente).

¹⁰ Le macro in common lisp sono un utile strumento che permette di “estendere” il linguaggio; defun è solitamente implementata come una macro.

```
(with-open-file (out "foo.lisp"
                      :direction :output
                      :if-exists :supersede
                      :if-does-not-exist :create)
  (mapcar (lambda (e)
             (format out "~S" e))
          ' ((1 . A) (2 . B) (42 . QD) (3 . D)))))

(with-open-file (in "foo.lisp"
                      :direction :input
                      :if-does-not-exist :error)
  (read-list-from in))

(defun read-list-from (input-stream)
  (let ((e (read input-stream nil 'eof)))
    (unless (eq e 'eof)
      (cons e (read-list-from input-stream)))))
```

Il secondo argomento a read stabilisce che nessun errore debba essere generato quando si incontra la fine del file. In questo caso va invece ritornato il valore passato come terzo elemento (ovvero il simbolo EOF).

```
CL-USER 13 > (defun read-list-from (input-stream)
                  (let ((e (read input-stream nil 'eof)))
                    (unless (eq e 'eof)
                      (cons e (read-list-from input-stream)))))

READ-LIST-FROM

CL-USER 14 > (with-open-file (out "foo.lisp"
                                      :direction :output
                                      :if-exists :supersede
                                      :if-does-not-exist :create)
  (mapcar (lambda (e)
            (format out "~S" e))
          ' ((1 . A) (2 . B) (42 . QD) (3 . D))))
(NIL NIL NIL NIL)
```

```
CL-USER 15 > (with-open-file (in "foo.lisp"
                                      :direction :input
                                      :if-does-not-exist :error)
  (read-list-from in))
((1 . A) (2 . B) (42 . QD) (3 . D))
```

8.16.6 Interazione con l'ambiente common lisp

L'ambiente lisp, o meglio la sua command-line, esegue tre operazioni fondamentali, ed ora che sappiamo qualcosa di più su I/O possiamo spiegarle.

- Legge (read) ciò che viene presentato in input
 - Ciò che viene letto viene rappresentato internamente in strutture dati appropriate
- La rappresentazione interna viene valutata (eval) al fine di produrre un valore (o più valori)

- Il valore ottenuto viene stampato.

Questo il read-eval-print loop.

8.17 VALUTAZIONE DI ESPRESSIONI E FUNZIONI

Dato che programmi e sexp's in lisp sono equivalenti, possiamo dare le seguenti regole di valutazione (ed implementarle nella funzione eval).

Data una sexp:

- Se è un atomo
 - se è un numero ritorna il suo valore
 - se è una stringa la ritorna così com'è
 - se è un simbolo
 - estrae il suo valore dall'ambiente corrente e lo ritorna
 - se non esiste un valore associato allora la segna come un errore
- se è una cons-cell ($O A_1 A_2 \dots A_n$):
 - se O è un operatore speciale, allora la lista $(O A_1 A_2 \dots A_n)$ viene valutata in modo speciale
 - se O è un simbolo che denota una funzione nell'ambiente corrente, allora questa funzione viene applicata (apply) alla lista $(VA_1 VA_2 \dots VA_n)$ che raccoglie i valori delle valutazioni delle espressioni $A_1 A_2 \dots A_n$
 - se O è una lambda expression la si applica alla lista $(VA_1 VA_2 \dots VA_n)$ che raccoglie i valori delle valutazioni delle espressioni $A_1 A_2 \dots A_n$
 - altrimenti si segnala un errore

8.17.1 apply

la funzione apply è definita come:

`apply : funzione list → sexp`

ovvero prende un designatore di funzione (ovvero un simbolo, una lambda-expression o una funzione) e ritorna un valore

8.17.2 eval

la funzione eval costruisce il valore denotato da una sexp

`eval : sexp env → sexp`

le funzioni apply e eval possono essere scritte direttamente in lisp, ovvero, dato che il lisp i dati e i programmi sono la stessa cosa, è possibile scrivere facilmente un interprete lisp in lisp.

Questi interpreti sono detti meta-circolari.

La costruzione di varianti di interpreti meta-circolari è uno dei metodi con cui si procede ad esplorare nuove modalità di programmazione.

Costruiamo la funzione valuta (eval è standard) a partire dalle regole di valutazione definite precedentemente.

La funzione valuta prende una S-expression sexp ed un "ambiente" env.

La funzione valuta procede nel seguente modo:

- 1) `sexp` è un'espressione autovalutante?
(self-evaluating-p `sexp`)
 - se sì, allora ritorna il suo valore
 - se no, allora...
- 2) `sexp` è una variabile?
(variable-p `sexp`)
 - se sì allora recuperarne il valore associato in `env`
(var-value `sexp env`)
 - se no, allora....
- 3) `sexp` è una espressione quotata della forma ('<e>)?
(quoted-exp-p `sexp`)
 - Se sì allora ritorna <e>
 - Se no, allora...
- 4) Sexp è una lambda-expressione? Ovvero una lista della forma (`lambda (...) ...`)?
(lambda-exp-p `sexp`)
 - Se sì allora crea una chiusura ricordando l'ambiente in cui questa espressione viene valutata (ovvero ricordando static link)


```
(make-fun (lambda-exp-vars   sexp)
            (lambda-exp-body   sexp)
            env)
```
 - Se no, allora...
- 5) Sexp è un'applicazione di una funzione a degli argomenti?
(application-exp-p `sexp`)
 - Se sì allora applica (`apply`) l'operatore alla lista dei valori ottenuti valutando ogni argomento


```
(apply (eval (operator sexp) env)
            (list-of-values (operands sexp) env))
```
 - Se no, allora...

8.17.3 Sequenza di valutazioni

(`progn <e1> <e2> ... <eN>`)

`Progn` crea un ordine di valutazione delle varie espressioni date come argomento.

Quando `progn` è l'espressione principale di una `defun` allora la si può elidere senza problemi; il corpo della `defun` si dice essere un `progn` implicito.

8.17.4 Funzioni utili per la valutazione delle sexp

- **self-evaluating-p**

```
(defun self-evaluating-p (x)
  (and (atom p) (not (symbolp x))))
```

- **quoted-exp-p**

```
(defun quoted-exp-p (x)
  (and (consp x) (eq (first x) 'quote)))
```

- **lambda-exp-p**

```
(defun lambda-exp-p (x) (and (consp x) (eq (first x) 'lambda)))
```

- **valuta-seq** (e, similaremente, **list-of-values**)

```
(defun valuta-seq (seq env)
  (mapcar (lambda (s) (valuta s env)) seq))
```

La funzione valuta si può quindi costruire a partire dalle regole di valutazione definite precedentemente:

```
(defun valuta (sexp &optional (env *the-global-environment*))
  (cond ((self-evaluating-p sexp) sexp)
        ((variable-p sexp) (var-value sexp env))
        ((quoted-exp-p sexp) (exp-of-quotation sexp))
        ((if-exp-p sexp) (valuta-if sexp env))
        ((lambda-exp-p sexp) (make-fun (lambda-exp-vars sexp)
                                         (lambda-exp-body sexp)
                                         env))
        ((sequence-exp-p sexp)
         (valuta-seq (sequence-expressions sexp) env))
        ((cond-exp-p sexp) (valuta (cond-to-if sexp) env))
        ((definition-exp-p sexp) (valuta-def sexp env))
        ((application-exp-p sexp)
         (applica (valuta (operator sexp) env)
                  (list-of-values (operands sexp) env)))
        (t (error ";; Non so come valutare : ~S." sexp))))
```

Ora costruiamo la funzione applica a partire dalle regole di valutazione definite precedentemente:

```
(defun applica (fun arguments)
  (cond ((primitive-fun-p fun)
         (apply-primitive-fun fun arguments))
        ((fun-p fun)
         (valuta-seq (fun-body fun)
                     (extend-environment (fun-parameters fun)
                                         arguments
                                         (fun-environment fun))))
        (t
         (error "Funzione ~S sconosciuta in APPLICA." fun))))
```

8.18 LA RAPPRESENTAZIONE INTERNA DI “FUNZIONI”

La valutazione di una espressione lambda genera una funzione che viene rappresentata nell’ambiente come una struttura particolare detta “chiusura”.

Questa struttura contiene il corpo dell’espressione lambda, la lista dei parametri formali e l’ambiente in cui l’espressione lambda è stata costruita, ovvero la struttura contiene lo static link all’ambiente di valutazione dove recuperare i valori delle variabili libere nel corpo dell’espressione lambda.

La funzione applica usa lo static link contenuto nella chiusura.

8.19 AMBIENTI (ENVIRONMENTS)

Le funzioni eval ed apply si appoggiano sull’implementazione degli ambienti, ovvero sulla manipolazione di mappe di associazioni tra simboli e valori. Un ambiente è una sequenza di frames.

La funzione var-value non è nient’altro che una “get” di una chiave in una mappa.

Come possiamo implementare le funzioni di manipolazione di un ambiente in common lisp?

- Make-frame
- Extend-env
- Var-value
- Var-value-in-frame

8.19.1 Make-frame

Un frame viene rappresentato come una lista di coppie prefissa dal simbolo frame:

```
(defun make-frame (vars values)
  (cons 'frame (mapcar 'cons vars values)))
;; mapcar agisce su più liste!
```

ESEMPIO

```
cl-prompt> (make-frame '(x y z) '(0 1 0))
(FRAME (X . 0) (Y . 1) (Z . 0))
```

```
cl-prompt> (make-frame '(q w) '(il-simbolo-q "W"))
(FRAME (Q . IL-SIMBOLO-Q) (W . "W"))
```

8.19.2 Extend-env

Un ambiente viene quindi esteso nella seguente maniera:

```
(defun extend-env (vars vals &optional (base-env *the-empty-env*))
  (if (= (length vars) (length vals))
      (cons (make-frame vars vals) base-env)
      (if (< (length vars) (length vals))
          (error "Too many arguments supplied")
          (error "Too few arguments supplied"))))
```

Come base definiamo anche

```
(defparameter *the-empty-env* '((frame (nil . nil) (t . t))))
```

Esempio

```
cl-prompt> (defparameter env1
  (extend-env '(x y z) '(0 1 0) ()))
ENV1

cl-prompt> (extend-env '(q w) '(il-simbolo-q "W") env1)
((FRAME (Q . IL-SIMBOLO-Q) (W . "W")) (FRAME (X . 0) (Y . 1) (Z . 0)))
```

8.20 RISCRITTURA DI ESPRESSIONI

Una delle operazioni più importanti che un interprete/compilatore fa è di riscrivere un'espressione in un'altra (più semplice) al fine di riutilizzare del codice già scritto.

Le strutture dati lisp usate per le espressioni da valuta ed applica rendono questa operazione particolarmente semplice:

- Cond viene riscritta in if
- Let viene riscritta nella corrispondente operazione lambda

L'espressione riscritta viene poi ripassata al valutatore per completarne l'esecuzione.

9 FUNZIONI LISP UTILI¹¹

- Caratteristiche imperative
 - Assegnamenti: setf (da non usare)
 - Costrutti di iterazione: dolist, dotimes, do, loop
- Caratteristiche object oriented
 - CLOS
 - Multimethods
- I/O
 - Open, close, write
- Macro
- Gestione eccezioni:
 - Error, handler-case, invoke-restart

9.1 COMMENTI

;;;; commento per descrivere il programma

;;; commento normale

;; commento identato col codice

; commento dopo una linea di codice

#|| [...] ||# commento multilinea

9.2 FORMAT, FORCE-OUTPUT

```
(defun dump-db ()
  (dolist (cd *db*)
    (format t "~{~a:~10t~a~%~}~%" cd)))
```

Common Lisp provides a couple ways to emit output, but the most flexible is the FORMAT function. FORMAT takes a variable number of arguments, but the only two required arguments are the place to send the output and a string.

(format t "Hello world")

- ~a: The ~a directive is the aesthetic directive; it means to consume one argument and output it in a human-readable form. This will render keywords without the leading : and strings without quotation marks.
- ~t: The ~t directive is for tabulating. The ~[numero]t tells FORMAT to emit enough spaces to move to the n column before processing the next.
- ~{ and ~}: When FORMAT sees ~{ the next argument to be consumed must be a list. FORMAT loops over that list, processing the directives between the ~{ and ~}, consuming as many elements of the list as needed each time through the list
- ~%: newline

The call to FORCE-OUTPUT is necessary in some implementations to ensure that Lisp doesn't wait for a newline before it prints the prompt.

¹¹ “Per il Lisp evitate la SET, la SETQ e la SETF (a meno che non sia strettamente necessario).” MA

```
(defun prompt-for-cd ()
  (make-cd
    (prompt-read "Title")
    (prompt-read "Artist")
    (or (parse-integer (prompt-
      read "Rating") :junk-allowed
      t) 0)
    (y-or-n-p "Ripped [y/n]: ")))
```

9.3 READ-LINE E LETTURA VALORI

Then you can read a single line of text with the aptly named READ-LINE function.

- **Prompt-read:** legge una stringa
- **Parse-integer:** interpreta una stringa come un numero
 - **:junk-allowed t:** permette che tra i numeri ci siano valori non numerici
- **Y-or-n-p:** accetta come risposta Y/y, N/n

9.4 DEFUN

Functions are one of the basic program building blocks in Lisp and can be defined with a DEFUN expression.

```
(defun hello-world (format t "Hello world"))
```

La forma classica è:

```
(defun name (parameter*)
  "Optional documentation string."
  body-form*)
```

- **&optional:** To define a function with optional parameters, after the names of any required parameters, place the symbol &optional followed by the names of the optional parameters
- **&rest:** Obviously, you could write functions taking a variable number of arguments by simply giving them a lot of optional parameters. To do it properly, you'd have to have as many optional parameters as the number of arguments that can legally be passed in a function call. This number is implementation dependent but guaranteed to be at least 50. Lisp lets you include a catchall parameter after the symbol &rest. If a function includes a &rest parameter, any arguments remaining after values have been doled out to all the required and optional parameters are gathered up into a list that becomes the value of the &rest parameter.
- **&key:** To give a function keyword parameters, after any required, &optional, and &rest parameters you include the symbol &key and then any number of keyword parameter specifiers, which work like optional parameter specifiers.

Si possono combinare i vari parametri. You can safely combine &rest and &key parameters, but the behavior may be a bit surprising initially. Normally the presence of either &rest or &key in a parameter list causes all the values remaining after the required and &optional parameters have been filled in to be processed in a particular way. If both &rest and &key appear in a parameter list, then both things happen—all the remaining values, which include the keywords themselves, are gathered into a list that's bound to the &rest parameter, and the appropriate values are also bound to the &key parameters.

```
(defun foo (&rest rest &key a b c) (list rest a b c))

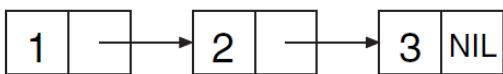
(foo :a 1 :b 2 :c 3) → ((:A 1 :B 2 :C 3) 1 2 3)
```

9.5 LIST

You can make a list with the LIST function, which, appropriately enough, returns a list of its arguments.

```
(list 1 2 3)
```

The key to understanding lists is to understand that they're largely an illusion built on top of objects that are instances of a more primitive data type. Those simpler objects are pairs of values called cons cells, after the function CONS used to create them. CONS takes two arguments and returns a new cons cell containing the two values. The two values in a cons cell are called the CAR and the CDR after the names of the functions used to access them.

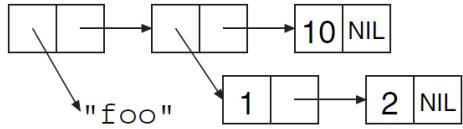


The box on the left represents the CAR, and the box on the right is the CDR. The values stored in a particular cons cell are either drawn in the appropriate box or represented by an arrow from the box to a representation of the referenced value.

```
(defparameter *list* (list 1 2 3 4))
(first *list*) → 1
(rest *list*) → (2 3 4)
(first (rest *list*)) → 2
```

Similarly, when you're thinking in terms of lists, you don't have to use the meaningless names CAR and CDR; FIRST and REST are synonyms for CAR and CDR that you should use when you're dealing with cons cells as lists.

```
(list "foo" (list 1 2) 10) → ("foo"
(1 2) 10)
```



Because cons cells can hold any kind of values, so can lists. And a single list can hold objects of different types.

the function APPEND takes any number of list arguments and returns a new list containing the elements of all its arguments.

REVERSE, the nondestructive function that returns a reversed version of a sequence, to

NREVERSE, a recycling version of the same function.

NREVERSE allows you to do exactly that. The N stands for non-consing, meaning it doesn't need to allocate any new cons cells. The exact side effects of NREVERSE are intentionally not specified—it's allowed to modify any CAR or CDR of any cons cell in the list—but a typical implementation might walk down the list changing the CDR of each cons cell to point to the previous cons cell.

However, not all do, including several of the more commonly used recycling functions such as NCONC, the recycling version of APPEND, and DELETE, DELETE-IF, DELETE-IF-NOT, and DELETE-DUPLICATES, the recycling versions of the REMOVE family of sequence functions.

| Function | Description |
|-----------|---|
| LAST | Returns the last cons cell in a list. With an integer, argument returns the last <i>n</i> cons cells. |
| BUTLAST | Returns a copy of the list, excluding the last cons cell. With an integer argument, excludes the last <i>n</i> cells. |
| NBUTLAST | The recycling version of BUTLAST; may modify and return the argument list but has no reliable side effects. |
| LDIFF | Returns a copy of a list up to a given cons cell. |
| TAILP | Returns true if a given object is a cons cell that's part of the structure of a list. |
| LIST* | Builds a list to hold all but the last of its arguments and then makes the last argument the CDR of the last cell in the list. In other words, a cross between LIST and APPEND. |
| MAKE-LIST | Builds an <i>n</i> item list. The initial elements of the list are NIL or the value specified with the :initial-element keyword argument. |
| REVAPPEND | Combination of REVERSE and APPEND; reverses first argument as with REVERSE and then appends the second argument. |
| NRECONC | Recycling version of REVAPPEND; reverses first argument as if by NREVERSE and then appends the second argument. No reliable side effects. |
| CONSP | Predicate to test whether an object is a cons cell. |
| ATOM | Predicate to test whether an object is <i>not</i> a cons cell. |
| LISTP | Predicate to test whether an object is either a cons cell or NIL. |
| NULL | Predicate to test whether an object is NIL. Functionally equivalent to NOT but stylistically preferable when testing for an empty list as opposed to boolean false. |

9.6 PLIST

A plist is a list where every other element, starting with the first, is a symbol that describes what the next element in the list is. You can use a particular kind of symbol, called a keyword symbol. A keyword is any name that starts with a colon (:).

```
(list :a 1 :b 2 :c 3)
```

9.7 GETF

The thing that makes plists a convenient way to represent the records in a database is the function GETF, which takes a plist and a symbol and returns the value in the plist following the symbol.

```
(getf (list :a 1 :b 2 :c 3) :b) → 2
```

9.8 DEFVAR, DEFPARAMETER, VARIABILI GLOBALI E PUSH

```
CL-USER> (add-record (make-cd "Roses"
  "Kathy Mattea" 7 t))
(:TITLE "Roses" :ARTIST "Kathy
Mattea" :RATING 7 :RIPPED T)
CL-USER> (add-record (make-cd "Fly"
  "Dixie Chicks" 8 t))
(:TITLE "Fly" :ARTIST "Dixie
Chicks" :RATING 8 :RIPPED T)
(:TITLE "Roses" :ARTIST "Kathy
Mattea" :RATING 7 :RIPPED T)
```

You can use a global variable, *db*, which you can define with the DEFVAR macro. The asterisks (*) in the name are a Lisp naming convention for global variables.

```
(defvar *db* nil)
```

You can use the PUSH macro to add items to *db*.

```
(defun add-record (cd)
  (push cd *db*))
```

```
(defvar *count* 0
  "Count of widgets made so far.")
(defparameter *gap-tolerance* 0.001
  "Tolerance to be allowed in
  widget gaps.")
```

Common Lisp provides two ways to create global variables: DEFVAR and DEFPARAMETER. Both forms take a variable name, an initial value, and an optional documentation string. After it has been DEFVARed or DEFPARAMETERed, the name can be used anywhere to refer to the current binding of the global variable. Global variables

are conventionally named with names that start and end with *.

The difference between the two forms is that DEFPARAMETER always assigns the initial value to the named variable while DEFVAR does so only if the variable is undefined.

After defining a variable with DEFVAR or DEFPARAMETER, you can refer to it from anywhere.

Practically speaking, you should use DEFVAR to define variables that will contain data you'd want to keep even if you made a change to the source code that uses the variable.

9.9 LOOP

```
(defun add-cds ()
  (loop (add-record (prompt-for-
    cd))
    (if (not (y-or-n-p "Another?
[y/n]: "))
      (return))))
```

You can use the simple form of the LOOP macro, which repeatedly executes a body of expressions until it's exited by a call to RETURN.

9.10 WITH-OPEN-FILE

```
(defun save-db (filename)
  (with-open-file (out filename
    :direction :output
    :if-exists :supersede)
    (with-standard-io-syntax
      (print *db* out))))
```

The WITH-OPEN-FILE macro opens a file, binds the stream to a variable, executes a set of expressions, and then closes the file. It also makes sure the file is closed even if something goes wrong while evaluating the body.

It contains the name of the variable that will hold the file stream to which you'll write within the body of WITH-OPEN-FILE, a value that must be a file name, and then some options that control how the file is opened.

Here you specify that you're opening the file for writing with :direction :output and that you want to overwrite

an existing file of the same name if it exists with :if-exists :supersede.

9.11 PRINT

PRINT prints Lisp objects in a form that can be read back in by the Lisp reader. The macro WITH-STANDARD-IO-SYNTAX ensures that certain variables that affect the behavior of PRINT are set to their standard values.

9.12 REMOVE-IF-NOT

The function REMOVE-IF-NOT takes a predicate and a list and returns a list containing only the elements of the original list that match the predicate. It creates a new list, leaving the original list untouched.

```
CL-USER> (remove-if-not #'evenp '(1 2 3 4 5 6 7 8 9 10))
(2 4 6 8 10)
```

The funny notation #' is shorthand for “Get me the function with the following name.” Without the #' , Lisp would treat evenp as the name of a variable and look up the value of the variable, not the function.

You can also pass REMOVE-IF-NOT an anonymous function (lambda expression).

9.13 LAMBDA

Note that lambda isn't the name of the function—it's the indicator you're defining an anonymous function.⁵ Other than the lack of a name, however, a LAMBDA expression looks a lot like a DEFUN: the word lambda is followed by a parameter list, which is followed by the body of the function.

```
(lambda (parameters) body)
```

Ora, se volessimo estrarre solo i pezzi di “Dixie chicks”, useremmo:

| | | |
|---|---|--|
| <pre>CL-USER> (remove-if-not #'(lambda (cd) (equal (getf cd :artist) "Dixie Chicks")) *db*) ((:TITLE "Home" :ARTIST "Dixie Chicks" :RATING 9 :RIPPED T) (:TITLE "Fly" :ARTIST "Dixie Chicks" :RATING 8 :RIPPED T))</pre> | → | <pre>(defun select-by-artist (artist) (remove-if-not #'(lambda (cd) (equal (getf cd :artist) artist)) *db*))</pre> |
|---|---|--|

Anonymous functions can be useful when you need to pass a function as an argument to another function and the function you need to pass is simple enough to express inline.

The other important use of LAMBDA expressions is in making closures, functions that capture part of the environment where they're created.

9.14 FUNZIONI CON SELETTORI VARI

```
CL-USER> (select (artist-selector "Dixie Chicks"))

((:TITLE "Home" :ARTIST "Dixie Chicks" :RATING 9 :RIPPED T)
 (:TITLE "Fly" :ARTIST "Dixie Chicks" :RATING 8 :RIPPED T))

(defun select (selector-fn)
  (remove-if-not selector-fn *db*))
  (defun artist-selector (artist)
    #'(lambda (cd) (equal (getf cd :artist) artist)))
```

Artist-selector is a function that returns a function and one that references a variable that—it seems—won't exist after artist-selector returns.

Now you just need some more functions to generate selectors. But just as you don't want to have to write select-by-title, select-by-rating, and so on, because they would all be quite similar, you're not going to want to write a bunch of nearly identical selector-function generators, one for each field. Why not write one general-purpose selector-function generator, a function that, depending on what arguments you pass it, will generate a selector function for different fields or maybe even a combination of fields? You can write such a function, but first you need a crash course in a feature called keyword parameters.

Sometimes you may want to write a function that can be called with varying numbers of arguments.

```
(defun foo (a b c) (list a b c)) → (defun foo (&key a b c) (list a b c))

(foo :a 1 :b 2 :c 3) → (1 2 3)
(foo :c 3 :b 2 :a 1) → (1 2 3)
(foo :a 1 :c 3) → (1 NIL 3)
(foo) → (NIL NIL NIL)
```

The only difference is the &key at the beginning of the argument list.

As these examples show, the value of the variables a, b, and c are bound to the values that follow the corresponding keyword. And if a particular keyword isn't present in the call,

the corresponding variable is set to NIL.

This variable (c-supplied-p in the example) will be bound to true if the caller actually supplied an argument for this parameter and NIL otherwise.

By convention, these variables are usually named the same as the actual parameter with a “-supplied-p” on the end. For example

```
(defun foo (&key a (b 20) (c 30 c-supplied-p)) (list a b
c c-supplied-p))

(foo :a 1 :b 2 :c 3) → (1 2 3 T)
(foo :c 3 :b 2 :a 1) → (1 2 3 T)
(foo :a 1 :c 3) → (1 20 3 T)
(foo) → (NIL 20 30 NIL)
```

Ora, se volessimo includere i vari selettori come chiavi:

```
(defun where (&key title artist rating (ripped nil ripped-p))
#'(lambda (cd)
  (and
    (if title (equal (getf cd :title) title) t)
    (if artist (equal (getf cd :artist) artist) t)
    (if rating (equal (getf cd :rating) rating) t)
    (if ripped-p (equal (getf cd :ripped) ripped) t))))
```

(select (where :rating 10 :ripped nil))

9.15 EVENP, ODDP

- evenp: controlla se il valore è pari
- oddp: controlla se il valore è dispari

9.16 MAPCAR

The main new bit is the use of a function MAPCAR that maps over a list, * db* in this case, and returns a new list containing the results of calling a function on each item in the original list.

MAPCAR is the function most like MAP. Because it always returns a list, it doesn't require the result-type argument MAP does. Instead, its first argument is the function to apply, and subsequent arguments are the lists whose elements will provide the arguments to the function.

MAPLIST is just like MAPCAR except instead of passing the elements of the list to the function, it passes the actual cons cells.

(mapcar #'(lambda (x) (* 2 x)) (list 1 2 3)) → (2 4 6)

(mapcar #'+ (list 1 2 3) (list 10 20 30)) → (11 22 33)

9.17 IF, CASE

The rule for IF is pretty easy: evaluate the first expression. If it evaluates to non- NIL, then evaluate the next expression and return its value. Otherwise, return the value of evaluating the third expression or NIL if the third expression is omitted.

(if test-form then-form [else-form])

Nel caso si abbia una serie di if-if else si può utilizzare cond in due maniere:

| casi separati | Casi riferiti ad una variabile |
|--|---|
| (cond (p ...) (q ...) (r ...) ... (t ...)) | (cond *variabile* (*valore variabile* ...) (*valore variabile* ...) (*valore variabile* ...) ... (t ...)) |

| | |
|--|--|
| | |
|--|--|

9.18 QUOTE

QUOTE, which takes a single expression as its “argument” and simply returns it, unevaluated. For instance, the following evaluates to the list (+ 1 2), not the value 3:

```
(quote (+ 1 2)) → (+ 1 2)
(quote (+ 1 2)) equivale a `(+ 1 2)
```

9.19 EQ, EQL, EQUAL, EQUALP

EQ tests for “object identity”—two objects are EQ if they’re identical. you should never use EQ to compare values that may be numbers or characters.

Thus, Common Lisp defines EQL to behave like EQ except that it also is guaranteed to consider two objects of the same class representing the same numeric or character value to be equivalent.

EQUAL loosens the discrimination of EQL to consider lists equivalent if they have the same structure and contents, recursively, according to EQUAL. EQUAL also considers strings equivalent if they contain the same characters.

EQUALP is similar to EQUAL except it’s even less discriminating. It considers two strings equivalent if they contain the same characters, ignoring differences in case. Numbers are equivalent under EQUALP If they represent the same mathematical value.

9.20 FUNCTION

In fact, you’ve already used FUNCTION, but it was in disguise. The syntax #' , which you used in Chapter 3, is syntactic sugar for FUNCTION, just the way ' is syntactic sugar for QUOTE.

| | | |
|--|----------|---|
| CL-USER> (function foo) #<Interpreted Function FOO> | Uguale a | CL-USER> #'foo #<Interpreted Function FOO> |
|--|----------|---|

9.21 FUNCALL, APPLY

Once you’ve got the function object, there’s really only one thing you can do with it— invoke it. Common Lisp provides two functions for invoking a function through a function object: FUNCALL and APPLY. They differ only in how they obtain the arguments to pass to the function.

FUNCALL is the one to use when you know the number of arguments you’re going to pass to the function at the time you write the code. The first argument to FUNCALL is the function object to be invoked, and the rest of the arguments are passed onto that function. It accepts a function object as an argument and plots a simple ASCII-art histogram of the values returned by the argument function when it’s invoked on the values from min to max, stepping by step.

Like FUNCALL, the first argument to APPLY is a function object. But after the function object, instead of individual arguments, it expects a list. It then applies the function to the values in the list.

As a further convenience, APPLY can also accept “loose” arguments as long as the last argument is a list.

APPLY doesn't care about whether the function being applied takes &optional, &rest, or &key arguments—the argument list produced by combining any loose arguments with the final list must be a legal argument list for the function with enough arguments for all the required parameters and only appropriate keyword parameters.

9.22 LET

Another form that introduces new variables is the LET special operator. The skeleton of a LET form looks like this:

```
(let (variable*)
  body-form*)
```

where each variable is a variable initialization form.

The following LET form, for example, binds the three variables x, y, and z with initial values 10, 20, and NIL:

```
(let ((x 10) (y 20) z)
  ...)
```

When the LET form is evaluated, all the initial value forms are first evaluated.

The scope of function parameters and LET variables is delimited by the form that introduces the variable. This form is called the binding form. As you'll see in a bit, the two types of variables use two slightly different scoping mechanisms, but in both cases the scope is delimited by the binding form.

| | |
|---|---|
| <pre>(defun foo (x) (format t "Parameter: ~a~%" x) (let ((x 2)) (format t "Outer LET: ~a~%" x) (let ((x 3)) (format t "Inner LET: ~a~%" x)) (format t "Outer LET: ~a~%" x)) (format t "Parameter: ~a~%" x))</pre> | <pre>CL-USER> (foo 1) Parameter: 1 Outer LET: 2 Inner LET: 3 Outer LET: 2 Parameter: 1 NIL</pre> |
|---|---|

Another binding form is a variant of LET, LET*. The difference is that in a LET, the variable names can be used only in the body of the LET—the part of the LET after the variables list—but in a LET*, the initial value forms for each variable can refer to variables introduced earlier in the variables list.

| Sì | No |
|---|---|
| <pre>(let* ((x 10) (y (+ x 10))) (list x y))</pre> | <pre>(let ((x 10) (y (+ x 10))) (list x y))</pre> |

9.23 WHEN, UNLESS

"doesn't Lisp provide a way to say what I really want, namely, 'When x is true, do this, that, and the other thing'?"

This is exactly what macros provide. In this case, Common Lisp comes with a standard macro, WHEN, which lets you write this:

```
(when (spam-p current-message)
```

```
(file-in-spam-folder current-message)
(update-spam-database current-message) )
```

A counterpart to the WHEN macro is UNLESS, which reverses the condition, evaluating its body forms only if the condition is false.

9.24 COND

```
(cond
```

```
(test-1 form*)
```

```
.
```

```
.
```

```
(test-N form*))
```

Each element of the body represents one branch of the conditional and consists of a list containing a condition form and zero or more forms to be evaluated if that branch is chosen. The conditions are evaluated in the order the branches appear in the body until one of them evaluates to true. At that point, the remaining forms in that branch are evaluated, and the value of the last form in the branch is returned as the value of the COND as a whole. If the branch contains no forms after the condition, the value of the condition is returned instead.

By convention, the branch representing the final else clause in an if/else-if chain is written with a condition of T.

9.25 AND, OR, NOT

NOT is a function so strictly speaking doesn't belong in this chapter, but it's closely tied to AND and OR. It takes a single argument and inverts its truth value, returning T if the argument is NIL and NIL otherwise.

Thus, AND stops and returns NIL as soon as one of its subforms evaluates to NIL. If all the subforms evaluate to non-NIL, it returns the value of the last subform.

OR, on the other hand, stops as soon as one of its subforms evaluates to non-NIL and returns the resulting value. If none of the subforms evaluate to true, OR returns NIL.

9.26 Do, DOLIST, DOTIMES

```
CL-USER> (dolist (x '(1 2 3))
  (print x) (if (evenp x)
    (return)))
1
2
NIL
```

DOLIST loops across the items of a list, executing the loop body with a variable holding the successive items of the list.

```
(dolist (var list-form)
  body-form*)
```

When the loop starts, the list-form is evaluated once to produce a list. Then the body of the loop is evaluated once for each item in the list with the variable var holding

the value of the item.

If you want to break out of a DOLIST loop before the end of the list, you can use RETURN.

```
(dotimes (x 20)
  (dotimes (y 20)
    (format t "~3d " (* (1+ x) (1+ y))))
  (format t "~%"))
```

DOTIMES is the high-level looping construct for counting loops. The basic template is much the same as DOLIST's. The count-form must evaluate to an integer. Each time through the loop

var holds successive integers from 0 to one less than that number.

```
(dotimes (var count-form)
  body-form*)
```

As with DOLIST, you can use RETURN to break out of the loop early.

Because the body of both DOLIST and DOTIMES loops can contain any kind of expressions, you can also nest loops.

While DOLIST and DOTIMES are convenient and easy to use, they aren't flexible enough to use for all loops.

Where DOLIST and DOTIMES provide only one loop variable, DO lets you bind any number of variables and gives you complete control over how they change on each step through the loop.

```
(do (variable-definition*)
  (end-test-form result-form*)
  statement*)
```

Each variable-definition introduces a variable that will be in scope in the body of the loop. The full form of a single variable definition is a list containing three elements:

```
(var init-form step-form)
```

- The init-form will be evaluated at the beginning of the loop and the resulting values bound to the variable var.
- Before each subsequent iteration of the loop, the step-form will be evaluated and the new value assigned to var. The step-form is optional; if it's left out, the variable will keep its value from iteration to iteration unless you explicitly assign it a new value in the loop body.

As with the variable definitions in a LET, if the init-form is left out, the variable is bound to NIL. Also as with LET, you can use a plain variable name as shorthand for a list containing just the name.

At the beginning of each iteration, after all the loop variables have been given their new values, the end-test-form is evaluated. As long as it evaluates to NIL, the iteration proceeds, evaluating the statements in order.

When the end-test-form evaluates to true, the result-forms are evaluated, and the value of the last result form is returned as the value of the DO expression.

9.27 LOOP

Well, it turns out a handful of looping idioms come up over and over again, such as looping over various data structures: lists, vectors, hash tables, and packages. Or accumulating values in various ways while

looping: collecting, counting, summing, minimizing, or maximizing. If you need a loop to do one of these things (or several at the same time), the LOOP macro may give you an easier way to express it.

The LOOP macro actually comes in two flavors—simple and extended.

9.27.1 Forma semplice

```
(loop
  (when (> (get-universal-time)
    *some-future-date*)
    (return))
  (format t "Waiting ...~%")
  (sleep 1))
```

(loop
body-form*)

this is an infinite loop that doesn't bind any variables.

The forms in body are evaluated each time through the loop, which will iterate forever unless you use RETURN to break out.

9.27.2 Forma estesa

It's distinguished by the use of certain loop keywords that implement a special-purpose language for expressing looping idioms.

| | | |
|--|---------|---|
| (do ((nums nil) (i 1 (1+ i))) ((> i 10) (nreverse nums)) (push i nums)) → (1 2 3 4 5 6 7 8 9 10) | diventa | (loop for i from 1 to 10 collecting i) → (1 2 3 4 5 6 7 8 9 10) |
|--|---------|---|

The symbols across, and, below, collecting, counting, finally, for, from, summing, then, and to are some of the loop keywords whose presence identifies these as instances of the extended LOOP.

9.28 DEFMACRO

The basic skeleton of a DEFMACRO is quite similar to the skeleton of a DEFUN:

```
(defmacro name (parameter*)
```

"Optional documentation string."

body-form*)

Like a function, a macro consists of a name, a parameter list, an optional documentation string, and a body of Lisp expressions. the job of a macro isn't to do anything directly—its job is to generate code that will later do what you want.

The job of a macro is to translate a macro form—in other words, a Lisp form whose first element is the name of the macro—into code that does a particular thing.

9.29 STRING E CHARS

| Numeric Analog | Case-Sensitive | Case-Insensitive |
|----------------|----------------|-------------------|
| = | CHAR= | CHAR-EQUAL |
| /= | CHAR/= | CHAR-NOT-EQUAL |
| < | CHAR< | CHAR-LESSP |
| > | CHAR> | CHAR-GREATERP |
| <= | CHAR<= | CHAR-NOT-GREATERP |
| >= | CHAR>= | CHAR-NOT-LESSP |

| Numeric Analog | Case-Sensitive | Case-Insensitive |
|----------------|----------------|---------------------|
| = | STRING= | STRING-EQUAL |
| /= | STRING/= | STRING-NOT-EQUAL |
| < | STRING< | STRING-LESSP |
| > | STRING> | STRING-GREATERP |
| <= | STRING<= | STRING-NOT-GREATERP |
| >= | STRING>= | STRING-NOT-LESSP |

the string comparators can compare only two strings. That's because they also take keyword arguments that allow you to restrict the comparison to a substring of either or both strings. The arguments :start1, :end1, :start2, and :end2 specify the starting (inclusive) and ending (exclusive) indices of substrings in the first and second string arguments.

```
(string= "stringa1" "stringa2" :start1 n :end1 n :start2 n :end2 n)
```

The comparators that return true when their arguments differ—that is, all of them except STRING= And STRING- -EQUAL— return the index in the first string where the mismatch was detected.

```
(string/= "lisp" "lissome") → 3
```

If the first string is a prefix of the second, the return value will be the length of the first string, that is, one greater than the largest valid index into the string.

```
(string< "lisp" "lisper") → 4
```

9.30 COLLEZIONI

9.30.1 Vettori e array

Vectors are Common Lisp's basic integer-indexed collection.

you should always use VECTOR or the more general function MAKE-ARRAY to create vectors you plan to modify.

MAKE-ARRAY is more general than VECTOR since you can use it to create arrays of any dimensionality as well as both fixed-size and resizable vectors. The one required argument to MAKE-ARRAY is a list containing the dimensions of the array. Since a vector is a one-dimensional array, this list will contain one number, the size of the vector.

MAKE-ARRAY will also accept a plain number in the place of a one-item list. With no other arguments, MAKE-ARRAY will create a vector with uninitialized elements that must be set before they can be accessed. To create a vector with the elements all set to a particular value, you can pass an :initial-element argument.

ESEMPIO

```
(make-array 5 :initial-element nil) → (NIL NIL NIL NIL NIL)
```

MAKE-ARRAY is also the function to use to make a resizable vector. A resizable vector also keeps track of the number of elements actually stored in the vector. This number is stored in the vector's fill pointer, so called because it's the index of the next position to be filled when you add an element to the vector.

To make a vector with a fill pointer, you pass MAKE-ARRAY a :fill-pointer argument.

Esempio

```
(make-array 5 :fill-pointer 0) → #()
```

To add an element to the end of a resizable vector, you can use the function VECTOR-PUSH. It adds the element at the current value of the fill pointer and then increments the fill pointer by one.

Per eliminare l'ultimo elemento dell'array si usa VECTOR-POP.

```
*x* → #(A B C)
```

```
(vector-pop *x*) → C
```

```
*x* → #(A B)
```

However, even a vector with a fill pointer isn't completely resizable. The vector *x* can hold at most five elements. To make an arbitrarily resizable vector, you need to pass MAKE-ARRAY another keyword argument: :adjustable.

```
(make-array 5 :fill-pointer 0 :adjustable t) → #()
```

This call makes an adjustable vector whose underlying memory can be resized as needed. To add elements to an adjustable vector, you use VECTOR-PUSH-EXTEND, which works just like VECTOR-PUSH except it will automatically expand the array if you try to push an element onto a full vector.

| Name | Required Arguments | Returns |
|------------|------------------------------|--|
| COUNT | Item and sequence | Number of times item appears in sequence |
| FIND | Item and sequence | Item or NIL |
| POSITION | Item and sequence | Index into sequence or NIL |
| REMOVE | Item and sequence | Sequence with instances of item removed |
| SUBSTITUTE | New item, item, and sequence | Sequence with instances of item replaced with new item |

```
(count 1 #(1 2 1 2 3 1 2 3 4)) → 3
(remove 1 #(1 2 1 2 3 1 2 3 4)) → #(2 2 3 2 3 4)
(remove 1 '(1 2 1 2 3 1 2 3 4)) → (2 2 3 2 3 4)
(remove #\a "foobarbaz") → "foobrbz"
(substitute 10 1 #(1 2 1 2 3 1 2 3 4)) → #(10 2 10 2 3 10 2 3 4)
(substitute 10 1 '(1 2 1 2 3 1 2 3 4)) → (10 2 10 2 3 10 2 3 4)
(substitute #\x #\b "foobarbaz") → "fooxarxaz"
(find 1 #(1 2 1 2 3 1 2 3 4)) → 1
(find 10 #(1 2 1 2 3 1 2 3 4)) → NIL
(position 1 #(1 2 1 2 3 1 2 3 4)) → 0
```

To limit the effects of these functions to a particular subsequence of the sequence argument, you can provide bounding indices with :start and :end arguments. Passing NIL for :end or omitting it is the same as specifying the length of the sequence. If a non -NIL :from-end argument is provided, then the elements of the sequence will be examined in reverse order. By itself :from-end can affect the results of only FIND and POSITION.

However, the :from-end argument can affect REMOVE and SUBSTITUTE in conjunction with another keyword parameter, :count, that's used to specify how many elements to remove or substitute. If you specify a :count lower than the number of matching elements, then it obviously matters which end you start from:

```
(find 'a #((a 10) (b 20) (a 30) (b 40)) :key #'first) → (A 10)
(find 'a #((a 10) (b 20) (a 30) (b 40)) :key #'first :from-end t) → (A 30)
(remove #\a "foobarbaz" :count 1) → "foobrbaz"
(remove #\a "foobarbaz" :count 1 :from-end t) → "foobarbz"
```

| Argument | Meaning | Default |
|-----------|---|---------|
| :test | Two-argument function used to compare item (or value extracted by :key function) to element. | EQL |
| :key | One-argument function to extract key value from actual sequence element. NIL means use element as is. | NIL |
| :start | Starting index (inclusive) of subsequence. | 0 |
| :end | Ending index (exclusive) of subsequence. NIL indicates end of sequence. | NIL |
| :from-end | If true, the sequence will be traversed in reverse order, from end to start. | NIL |
| :count | Number indicating the number of elements to remove or substitute or NIL to indicate all (REMOVE and SUBSTITUTE only). | NIL |

For each of the functions just discussed, Common Lisp provides two higher-order function variants that, in the place of the item argument, take a function to be called on each element of the sequence: -IF e -IF-NOT.

```
(count-if #'evenp #(1 2 3 4 5)) → 2
(count-if-not #'evenp #(1 2 3 4 5)) → 3
(position-if #'digit-char-p "abcd0001") → 4
(remove-if-not #'(lambda (x) (char= (elt x 0) #\f))
  #("foo" "bar" "baz" "foom"))
  → #("foo" "foom")
```

9.31 SUBSEQUENCE MANIPULATIONS

Another set of functions allows you to manipulate subsequences of existing sequences. The most basic of these is SUBSEQ, which extracts a subsequence starting at a particular index and continuing to a particular ending index or the end of the sequence.

SUBSEQ is also SETFable, but it won't extend or shrink a sequence; if the new value and the subsequence to be replaced are different lengths, the shorter of the two determines how many characters are actually changed.

If you need to find a subsequence within a sequence, the SEARCH function works like POSITION except the first argument is a sequence rather than a single item.

```
(position #\b "foobarbaz") → 3
```

```
(search "bar" "foobarbaz") → 3
```

9.32 I/O

You obtain a stream from which you can read a file's contents with the OPEN function. By default OPEN returns a character-based input stream you can pass to a variety of functions that read one or more characters of text: READ-CHAR reads a single character; READ-LINE reads a line of text, returning it as a

string with the end-of-line character(s) removed; and READ reads a single s-expression, returning a Lisp object. When you're done with the stream, you can close it with the CLOSE function.

```
(open "/some/file/name.txt")
```

You can use the object returned as the first argument to any of the read functions. For instance, to print the first line of the file, you can combine OPEN, READ -LINE, and CLOSE as follows:

```
(let ((in (open "/some/file/name.txt")))
  (format t "~a~%" (read-line in))
  (close in))
```

If you want to open a possibly nonexistent file without OPEN signaling an error, you can use the keyword argument :if-does-not-exist to specify a different behavior. The three possible values are :error, the default; :create, which tells it to go ahead and create the file and then proceed as if it had already existed; and NIL, which tells it to return NIL instead of a stream.

```
(let ((in (open "/some/file/name.txt" :if-does-not-exist nil)))
  (when in
    (loop for line = (read-line in nil)
          while line do (format t "~a~%" line))
    (close in)))
```

9.33 NTH

(nth [indice] '[una lista])

Inizia da 0

10 INTRODUZIONE A C E C++

Il C è uno dei linguaggi fondamentali:

- Linguaggio a base di UNIX
- Linguaggio di livello relativamente basso
- Richiede molta disciplina e aderenza a varie convenzioni

Il C++ nasce come estensione ad oggetti del C.

10.1 COMPILAZIONE ED ESECUZIONE

Assunzioni fondamentali:

- Si usano gli strumenti richiamabili dalla linea di comando
- L'interprete di comandi è una shell UNIX o il cmd di windows
- È possibile avere un sottosistema UNIX sotto windows installando cygwin

Il compilatore C e/o C++ ha diversi nomi su diverse piattaforme.

- Gcc (g++) è il compilatore su molti sistemi UNIX
- Cl è il compilatore microsoft

Ora, supponiamo di aver salvato il programma C o C++ in un file chiamato hello.c.

Per compilare il programma si invoca il compilatore nel prompt:

gcc hello.c

se non ci sono errori, l'eseguibile sarà un file chiamato (convenzionalmente) a.out che può essere richiamato direttamente:

a.out

è possibile richiedere un nome specifico per l'eseguibile utilizzando l'opzione -o:

g++ -o ciao hello.cc

ciao

c++ è di fatto un C esteso, quindi i tipi di dati fondamentali si comportano (essenzialmente) allo stesso modo.

In C/C++ i vari elementi del linguaggio sono denotati da nomi (o identificatori) che possono essere composti da lettere, numeri e _. La regola è che devono iniziare necessariamente con una lettera o _.

10.2 TIPI FONDAMENTALI

Tipi semplici:

- Int
- Char
- Float
- Bool
- Puntatori e riferimenti

Tipi aggregati:

- Array (es. int a [10] è un array di 10 interi, float m[2][2] è una matrice 2x2 di numeri in virgola mobile)
- Strutture

Il C ha anche un tipo particolare che denota la mancanza di informazione: void

Infine ha a disposizione un **tipo “enumerazione” di costanti**: `enum [nome] {<c1>, <c2>, ..., <cn>}`

Il C++ ha a disposizione la nozione di classe come base della programmazione ad oggetti.

10.3 VARIABILI

Un programma C deve manipolare valori che vanno associati a “nomi”, ovvero a variabili e/o funzioni. Tutti questi nomi vengono introdotti in un programma C/C++ per mezzo di dichiarazioni (come in java).

Le dichiarazioni solitamente hanno la forma

```
<tipo> <modificatori> <nome> <modificatori> [= <inizializzazione>]
```

ESEMPIO

```
Char a = 'a';
float x;
int qd = 42;
int f(float, bool);    f è una funzione che prende un float e un booleano e restituisce un intero
char* ps;   ps è un puntatore che punta ad una locazione di memoria che contiene un carattere
float** m; m è un puntatore che punta ad un puntatore che punta ad un float
float v[]; v è un array di dimensione non specificata
```

Solo in C++:

```
int x = 42
int& rx = x;      una referenza all'intero x
```

10.3.1 Arrays

Gli array sono una componente importante del C/C++, ma il loro comportamento è molto sofisticato e, allo stesso tempo, di “basso livello”.

Gli array sono da considerarsi come un blocco di memoria fisica. La dichiarazione `unsigned char s[80];` di fatto riserva un blocco di 80 char (ogni singolo char prende 8 bit) nella RAM.

Si noti come una zona di memoria, ad esempio 512 Mb, potrebbe essere dichiarata in un programma C come `unsigned char total_memory[512*1024*1024];`.

Gli array non hanno attributi associati oltre al tipo. Un array può essere inizializzato anche con dei valori iniziali:

```
int un_due_tre[3] = {1, 2, 3}
```

in questo caso non è necessario dichiarare la dimensione dell'array:

```
float an_array[] = {3.0, 42}
```

però bisogna stare attenti, **gli array non si espandono automaticamente quando serve**, quindi “`int fa[2] = {3, 42, 0}`” è un errore.

Le stringhe in C non hanno un tipo specifico, sono degli array terminati con un carattere nullo '\0' implicitamente inserito dal compilatore, ad esempio:

```
char stringa[] ) "Dylan dog";
```

equivale a

```
char stringa[] = {'D', 'y', 'l', 'a', 'n', ' ', 'D',
'D', 'o', 'g', '\0'};
```

quindi nonostante la stringa sia composta da 9 caratteri, la lunghezza di stringa è 10.

La convenzione di terminare una stringa con un carattere '\0' è essenziale per il funzionamento in C.

10.3.2 Strutture

Le strutture in C/C++ sono aggregazioni di tipi diversi.

Vengono create tramite la parola chiave “struct”

```
Struct polar_complex {
    float magnitude;
    float angle; } * c = {42, 3.14}
```

I campi di una struttura si estraggono con la notazione punteggiata come in java.

`c.magnitude → 42.`

In C++ è possibile usare dei “costruttori” per inizializzare le strutture.

10.3.3 Puntatori

Il concetto di “Puntatore” è fondamentale per qualunque tipo di programmazione. Ogni sistema di programmazione ha, a qualche livello, un concetto equivalente:

- nella programmazione dei database relazionali le strutture di chiavi incrociate, non sono altro che una particolare realizzazione di questo concetto
- in Java, di fatto, è quasi tutto un puntatore.

In C/C++ i puntatori e le referenze sono esplicativi.

Dato un qualunque tipo T, il tipo associato T* è il tipo “puntatore a T”, ovvero, una variabile di tipo T* contiene l'indirizzo in memoria di un oggetto di tipo T.

```

int x = 42;

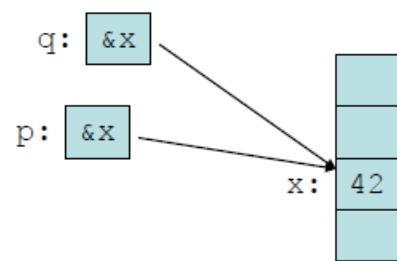
int* p = &x;      // p contiene l'indirizzo di x ottenuto
                  // tramite

                  //l'operatore &

int* q = p;      // q punta a p che punta ad x

int y = *q;      // y prende il valore puntato da q,
                  //quindi da p, quindi 42

```



10.3.4 Puntatori ed array

Un array è un puntatore sono molto simili in C.

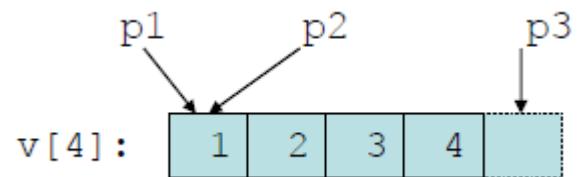
Il nome di un array può essere usato come un puntatore al suo primo elemento.

```

Int v[] = {1, 2, 3, 4};

int* p1 = v;
int* p2 = v[0];
int* p3 = v[4]    // attenzione, v[4] è un
                  //elemento successivo
                  //all'ultimo.

```



Tutto ciò può essere causa di problemi, soprattutto di sicurezza.

10.3.5 Puntatori, array e aritmetica

In C è possibile eseguire operazioni aritmetiche su puntatori. Ovvero, dato un puntatore p, l'espressione p+2 ha un senso.

```

Int v[] = {1, 2, 3, 4};

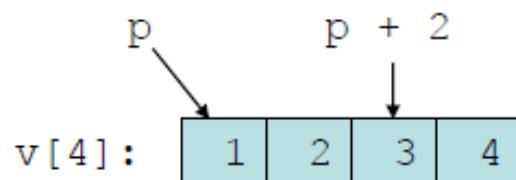
int* p = v;    //puntatore al primo elemento,
              // v[0]

int x = *(p + 2); //puntatore al terzo
                  //elemento

int y = v[2];

bool xy = (x == y); // xy è vero

```



10.3.6 Puntatori, array, aritmetica e stringhe

Dato che in C/C++ una stringa non è altro che un array terminato da un carattere nullo, e dato che il valore di falsità è pari al valore 0, la funzione dalla libreria `strlen` calcola la lunghezza di una stringa. Può avere il seguente ciclo:

```
char s[] = "Catarella";
char* p = 0;      // punta a nulla
int l = 0;
for (p = s; *p; p++)
l++;
```

alla fine del ciclo `l` conterrà la lunghezza di `s`.

`p = s` equivale a far puntare `p` al primo elemento di `s`.

```
const int * ptr;
```

significa che i dati puntati sono costanti e immutabili ma il puntatore non lo è.

```
int * const ptr;
```

significa che il puntatore è costante e immutabile ma i dati puntati non lo sono.

10.4 BLOCCHI E OPERATORI CONDIZIONALI E DI ITERAZIONE

```
if (<espressione>
    <blocco o statement>
[else <blocco o statement>]

while (<espressione>
    <blocco o statement>

do <blocco o statement>
while (<espressione>

for (<inizio>; <espressione>; <espressione>)
    <blocco o statement>
```

In C/C++ vi è il concetto di blocco di operazioni racchiuso tra graffe {}. Ogni blocco introduce un ambito ("scope").

Il C/C++ ha i soliti operatori condizionali e di iterazione con la seguente sintassi, uguale a quella in Java.

If sfrutta anche la vecchia forma "condizione ? conseguenza se vero : conseguenza se falso"

```
switch (<espressione> {
case <valore letterale 1>:
    <statement>* [break;]
case <valore letterale 2>:
    <statement>* [break;]
...
case <valore letterale k>:
    <statement>* [break;]
[default:
    <statement>* [break;]]
}
```

10.5 ALTRI “STATEMENTS”

Infine, abbiamo:

`return <espressione>;`

`return` serve a fine funzione per restituire i valori alla funzione chiamante, uguale a Java.

`goto <label>;`

`<label> : <statement>`

attenzione, `goto` può introdurre vari problemi se usato male.

In pratica **prima delle istruzioni possono essere inserite delle etichette seguite da ":"**. Un'istruzione `goto` seguita da un'etichetta porterà il program counter direttamente a quell'istruzione (è un branch sotto mentite spoglie).

```
try { <statement>* }
catch (<dichiarazione di eccezione>) {
    <statement>*
}
...
```

10.6 ESPRESSIONI

In C/C++ le espressioni sono costruite a partire da vari operatori.

| | | | | |
|--------------------------------|--|--------------------------------------|---|--|
| Accesso | <code>. -> [] &(prefisso)</code> | Chiamata di funzione | <code>()</code> | |
| Dimensionamento | <code>Sizeof</code> | Aritmetici | <code>+ - * / %</code> | |
| Incrementi e decrementi | <code>++ -- (prefissi e postfissi)</code> | Shift | <code><< >></code> | |
| Logici e booleani | <code>~ ! < <= > >= == != & ^ && </code> | Assegnamento | <code>= *= /= %= += -= <=>= &= = ^=</code> | |
| Sequenza | <code>,</code> | In c++ abbiamo anche altri operatori | | |
| Informazioni sui tipi | <code>Typeid</code> | Cambiamenti di tipi | <code>Dynamic_cast static_cast reinterpret_cast const_cast</code> | |
| Eccezioni | <code>throw</code> | | | |

10.7 FUNZIONI

Un programma C/C++ è costituito da un insieme di funzioni. Una di queste funzioni, main, ha il ruolo particolare di rappresentare il punto di inizio di un programma.

Una funzione C/C++ viene definita nel seguente modo:

```
<tipo di ritorno> <nome funzione> (<argomenti con tipo di variabile>) {
    <dichiarazioni di altre variabili locali>
    <codice>
}
```

Ad esempio:

```
int plus (int a, int b) {
    return a + b;
};
```

| esempio di funzione ricorsiva: fattoriale |
|--|
| <pre>int fact (int n) { If (n == 0) return 1; else if (n>0) return n * fact (n-1); else println("n < 0, valore invalido"); }</pre> |

10.7.1 Funzioni e variabili locali

Abbiamo visto delle funzioni che accettano dei parametri e come i loro tipi vengono dichiarati.

Le variabili locali vengono dichiarate in maniera simile, la loro comodità è che quando non sono più necessarie vengono rimosse dalla memoria.

Ad esempio:

| esempio di funzione iterativa con valori locali: fattoriale |
|---|
| <pre>int fact_i (int n) { int acc = 1, i; // acc e i sono locali for (i = 1; i < n; i++) acc *= i; return acc; }</pre> |

10.7.2 Funzioni e passaggio di parametri

Ad eccezione degli array, tutti i valori vengono passati per valore.

Linguaggio: C

```
Int x = 3;
int y = 42;

void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

swap (&x, &y);
```

Linguaggio: C++

```
Int x = 3;
int y = 42;

void swap(int& a, int& b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

swap (x, y);
```

I puntatori ci permettono di simulare ciò che in altri linguaggi viene chiamato “passaggio di parametri per riferimento”.

Nell'esempio qua accanto la funzione **swap** prende in ingresso due puntatori. Chiamarla ponendo il prefisso **&** alle variabili serve proprio a indicare che si sta passando il loro indirizzo e non il valore.

In questo modo qualsiasi modifica all'interno della funzione swap avrà effetto anche all'esterno di essa.

In C++ la funzione swap può essere scritta usando il tipo “reference”, ovvero ponendo all'interno della dichiarazione della funzione **&** e poi chiamandola in maniera classica. Il secondo esempio è scritto in c++.

```
Int i[3] = {1, 0, 0};
int v[3] = {1, 2, 3};

int vsmult(int a[], int b[]) {
    int result = 0;
    for (i = 0; i < 3; i++)
        result += a[i] * v[i];
    return result;
}

Int s = vsmult(i, v);
```

Gli array vengono passati ad una funzione per riferimento, ovvero un parametro di tipo array converte automaticamente un'espressione di tipo T[] ad un puntatore al primo elemento.

La chiamata

```
Int s = vsmult(i, v);
```

deposita il valore 1 in s, senza copiare in a e b gli interi array i e v.

Passare parametri per riferimento è molto utile anche per le strutture come mostrato dall'esempio successivo.

P è una variabile di tipo struct persona. Se vogliamo scrivere una funzione che manipoli il valore della variabile p, sarebbe bene usare dei puntatori.

```
Struct persona {
    Char nome[80];
    int età;
    char assistente_fidato[80];
}
Persona p = {"Salvo montalbano",
42, "Catarella"};
```

```
Void stampa_persona(struct
persona* x) {
    printf("<Persona '%s',
'%s' %d>\n",
(*x).nome,
x -> assistente_fidato,
x -> età);
}
```

L'operatore -> è un'abbreviazione, serve per indicare di prendere da x la variabile che ha come nome quello indicato successivamente.

L'output della chiamata “stampa_persona(&p); produce “<Persona ‘Salvo Montalbano’ ‘Catarella’ 42>.

Il più recente standard C permette il passaggio di strutture per valore.

10.8 LA COMPILAZIONE DEI PROGRAMMI IN C/C++

Consigliato: installare Codeblocks con il compilatore C integrato

<http://www.codeblocks.org/downloads/26/codeblocks-xx.yymingw-setup.exe>

dopo averlo installato all'interno della cartella di installazione ci sarà la cartella MinGW, aggiungere \$\$\backslash CodeBlocks\MinGW\bin alla variabile di sistema path.

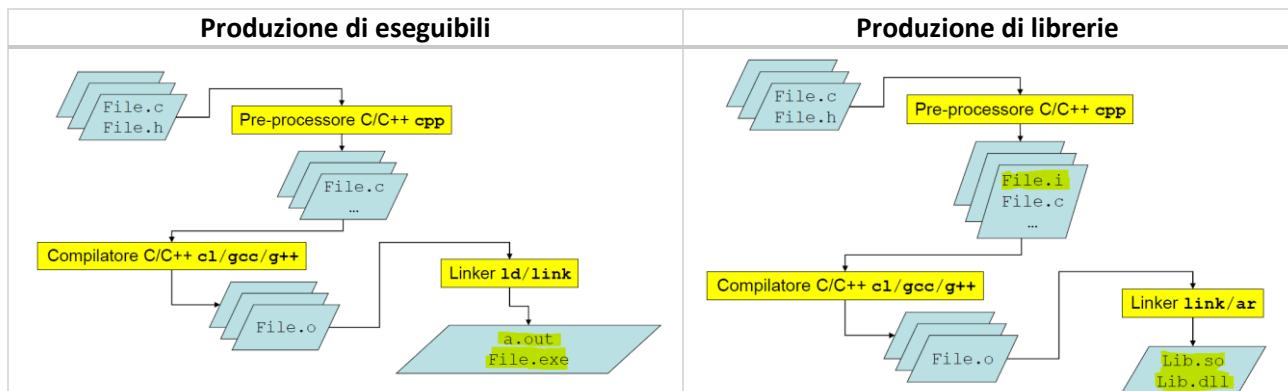
Dopo aver scritto i programmi potranno essere compilati dal prompt di sistema.

La chiamata alla compilazione del file.c è semplice se c'è un solo file, nel caso invece che il codice sorgente sia distribuito su un insieme di files e directories serve che la loro organizzazione sia costruita correttamente.

In C/C++, i programmi si basano sulla distinzione tra:

- **Header files** (estensione .h o .hpp)
- **File di implementazione** (estensione .c, .cc, .C, .cpp)

Questa distinzione fa leva sul pre-processore C/C++.



La compilazione di default chiama prima di tutto il preprocessore. Ci sono funzioni aggiuntive che permettono di richiamare anche solo il compilatore.

Chiamare il compilatore dalla linea di comando fa sì che venga chiamato anche il linker.

10.8.1 Preprocessore

Il preprocessore è un programma che trasforma testo, non necessariamente il testo deve essere in C/C++.

Il preprocessore opera sulla base di direttive. Le più utili sono di tre tipi:

- Inclusione di testo
- Definizione di “macro”
- Condizionali

Tutte queste direttive funzionano assieme per permetterci di costruire programmi modulari.

| Direttiva | Forma | Scopo | |
|-----------------------|---|--|---|
| <i>Di inclusione</i> | #include "file.h" #include <file.h> | Include tutto il file nel sorgente | |
| <i>Di definizione</i> | #define PI 3.14L #define max(x, y) ((x) < (y) ? (y) : (x)) | Tutte le istanze della stringa PI o di stringhe max(a, b) vengono sostituite | |
| <i>Condizionali</i> | #ifdef PI ... #else ... #endif | #ifndef PI ... #else ... #endif | Il testo compreso tra #ifdef e #endif viene incluso o meno se la macro è definita |

Usare il preprocessore era l'unico modo di introdurre “costanti” simboliche in C. Buona parte degli header contengono parecchie definizioni di costanti che vengono processate dal preprocessore C/C++.

Esempio: di solito su una piattaforma simil.unix, il file limits.h viene incluso con la direttiva “#include <limits.h>” e contiene varie definizioni, come ad esempio “#define CHAR_MAX 127, #define INT_MAX 2147483647”.

10.8.2 Compilazione separata e “header” files

Ogni programma di una certa dimensione dovrebbe essere modularizzato in maniera appropriata. La modularizzazione di un programma corrisponde all’operazione di compilazione separata.

Un compilatore C/C++ agisce (di solito) su un solo file; il risultato è un file “oggetto” contenente dei riferimenti irrisolti a codice non direttamente disponibile.

```
use-foo.c
#include <stdio.h>
Int foo(int);

int main() {
    printf("foo(42) == %d\n", foo(42));
}
```

Il linker ha il compito di risolvere questi riferimenti e, nel caso, segnalare degli errori qualora qualche riferimento rimanga irrisolto.

Prendiamo ad esempio il seguente file use-foo.c qua accanto.

Se proviamo a compilarla il linker ci avviserà che foo non è definita ma solo dichiarata.

```
foo.c
Int foo(int x) {
    Return x == 42 ? 1 : 0;
}
```

Ora definiamola nel file foo.c. cercando di compilare foo.c però ci verrà riferito che manca la funzione main.

Per evitare questi errori si usa la funzione -c in fase di compilazione:

```
prompt$ gcc -c foo.c
prompt$ gcc -c use-foo.c
```

l'ordine è ininfluente. Il risultato è la creazione di due file, use-foo.o e foo.o ma ciò non permette comunque il loro uso in quanto file separati.

La creazione di un eseguibile richiede un passo ulteriore: i files use-foo.o e foo.o devono essere linkati assieme:

```
prompt$ gcc use-foo.o foo.o → a.out
```

regole per definizioni e dichiarazioni distribuite su files multipli:

- **Tutte le dichiarazioni di variabili e funzioni devono essere consistenti per tipo**
- Ogni oggetto può essere definito una volta sola, altrimenti il linker restituisce un errore

Use-foo.c

```
#include <stdio.h>
#include "foo.h"

Int foo(int);

int main() {
    printf("foo(42) == %d\n", foo(42));
}
```

Un modo di separare l'interfaccia di un “Modulo” (foo in questo caso) dalla sua implementazione, evitando al tempo stesso il pericolo di ridefinizioni incontrollate, consiste nell’usare attentamente gli header files.

Riscriviamo il file use-foo.c, riprendiamo foo.c e scriviamo foo.h.

Il file foo.h contiene l'interfaccia del modulo foo, la compilazione di use-foo.c include foo.h e quindi rende disponibile la dichiarazione. Il file foo.h

rimane a disposizione per essere incluso in altri files. Per modificare l'interfaccia del modulo foo dobbiamo modificare un solo file.

Foo.c

```
Int foo(int x) {
    Return x == 42 ? 1 : 0;
}
```

Foo.h

```
extern int foo(int);
```

Questo è un esempio fin troppo semplice. Consideriamo un caso con due files che vogliono “usare” dei numeri complessi.

```
/* complex-use-1.c */
Struct complex {float im; float re; }

Struct complex c_mult (struct complex*, struct complex*);
```

```
/* complex-use-2.c */
Struct complex {float angle; float magnitude};

Struct complex c_mult (struct complex* cl, struct complex* c2) { ... }
```

In questi due casi abbiamo un conflitto nel contenuto delle due strutture (desumibile dai nomi dei campi).

La concentrazione di queste definizioni in un solo file complex.h evidenzierebbe questo problema specifico, per questo è comodo utilizzare i file header, permette di riassumere le varie intestazioni dei file all'interno di un solo posto, permettendo così di vedere subito se un metodo è stato definito in maniera differente più volte.

```
Foo.c
#include "foo.h"

Int foo(int x) {
    Return x == 42 ? 1 : 0;
}
```

Cosa accade se viene modificato il file al quale si riferisce il file header? Dipende dal grado di sofisticazione del compilatore. **Abbiamo bisogno di imporre all'implementazione di un modulo il rispetto della sua interfaccia pubblica.** La cosa più semplice da fare è di includere l'header nel file di implementazione stesso.

10.8.2.1 Inclusioni multiple

La prossima parte è abbastanza complicata.

Se vogliamo includere foo.h in un header file baz.h per poi includere sua foo.h che baz.h in un file use-foo-baz.c dobbiamo mettere in conto la possibilità di definizioni multiple: strutture, tipi.

Osservando l'esempio nell'immagine successiva, **i due file header (foo e baz) includerebbero entrambi foo.c, il primo includendolo tramite la dichiarazione extern, il secondo perché include foo.h e quindi, di rimando, foo.c.**

L'inserimento di direttive condizionali (#ifdef, #ifndef) e la creazione tramite #define di token specifici per controllare se un'interfaccia sia già stata inclusa è comodo.

Per spiegare meglio questo concetto vediamo l'esempio con i file baz.h, foo.h e come vengono modificate le loro intestazioni.

Spiegando cosa accade nell'immagine:

- use-foo-baz.c include foo.h
 - foo.h controlla se _FOO_H è stato già definito
 - no, quindi genera il token _FOO_H e include successivamente la funzione foo
- use-foo-baz.c include baz.h
 - baz.h controlla se _BAZ_H è già stato definito
 - no, quindi genera il token _BAZ_H e include successivamente l'header foo.h
 - foo.h controlla se _FOO_H è stato già definito
 - sì, quindi salta il codice definito nel blocco #ifndef ... #endif
 - baz.h include le funzioni zot e baz

perché tutto questo sforzo aggiuntivo? Per ottimizzare i tempi di esecuzione, infatti così ogni file viene incluso una singola volta.

```

use-foo-baz.c
1 /* use-foo-baz.c */
2
3 #include "foo.h"      //include foo.h
4 #include "baz.h"      //include zot, baz.
5           //foo.h non viene re-incluso
6
7 /* codice che usa sia foo che baz */
8
9 /* spiegazione: foo definisce già _FOO_H,
10    quindi quando baz.h cerca di definirlo
11    non lo definisce una seconda volta */
12

baz.h
1 /* baz.h */
2
3 #ifndef _BAZ_H //se il token _BAZ_H non è stato già definito
4     #define _BAZ_H //lo definisce, in questo modo non vengono
5           // definite più volte certe parti
6
7     include "foo.h" // qua include foo.h, controllando se _FOO_H
8           // sia già stato generato da una chiamata
9           // precedente
10
11     struct zot {char* zut; double qd[42]; };
12     extern void baz(int z, struct zot*);
13
14 #endif
15

foo.c
1 #include "foo.h"
2
3 int foo(char x){
4     return x == 'x' ? 42 : 0; //definizione di foo
5 }

foo.h
1 /* foo.h */
2
3 #ifndef _FOO_H // se il token _FOO_H non è stato creato
4     #define _FOO_H // lo definisce, in modo da segnalare che
5           // l'header è già stato incluso
6           // almeno una volta
7
8     extern int foo(int);
9
10 #endif
11
12 //include foo

```

10.9 COME ORGANIZZARE LE LIBRERIE IN C

Che cos'è una libreria? È un file in un particolare formato che può essere manipolato dal linker (sia staticamente che dinamicamente). Una libreria dinamica ha l'estensione .dll in windows.

Una libreria è essenzialmente una collezione di file oggetto con un indice associato che permette al linker (od al programma in esecuzione) di andare a caricare il codice corrispondente ad un dato “entry point”, ovvero ad una funzione.

```

QUPCP.h

#ifndef _QUIPCP_H
#define _QUIPCP_H

extern const int N;

    //inizializzazione non
    desiderata

extern bool find(int, int);
extern void unite(int, int);
extern void init_quick_find_pcp();

#endif

```

Ad esempio, consideriamo una libreria chiamata QUICK-UNION-PCP (che implementa l'algoritmo UNION-FIND). Potremmo mettere tutto il codice in un solo file con la funzione main (o altre) che usa le funzioni find e unite.

Invece per prima cosa scorporiamo l'interfaccia della libreria QUICK-UNION-PCP e la mettiamo in un file QUPCP.h

Questa definizione non è ancora ottimale, ma dà un'idea della struttura della libreria.

Note:

- Il parametro N non è visibile all'esterno dell'implementazione
- L'unico elemento non parametrico è il tipo (int) della rappresentazione usata per codificare gli elementi

A questo punto potremmo cambiare completamente l'implementazione nel file QUPCP.cc (il file contenente la definizione di funzioni, costanti e parametri).

QUPCP.h

```
#ifndef _QUIPCP_H
#define _QUIPCP_H

extern const int qupcp_N;

//inizializzazione non desiderata
extern bool qupcp_find(int, int);
extern void qupcp_unite(int, int);
extern void
qupcp_init_quick_find_pcp();

#endif
```

La libreria QUPCP “esporta” una costante chiamata N. La scelta di questo nome non è particolarmente felice, dato che il nome N potrebbe essere specificato da molte altre librerie, nonché dal programma principale.

In C è assolutamente necessario essere molto disciplinati nella scelta dei nomi. La soluzione più semplice consiste nel prefissare ogni nome della libreria visibile all'esterno con un identificatore univoco. In questo caso basterebbe usare il prefisso “qupcp_” per distinguerne l'origine.

Per costruire le librerie si usano differenti programmi in base alla piattaforma, per windows è link.

10.10 MEMORIA DINAMICA IN C/C++

Finora abbiamo visto solo dichiarazioni e definizioni di oggetti C/C++ che vengono allocati sullo stack di sistema.

```
Int *p = (int*) malloc(10 * sizeof (int));
int i;
*p = 1;
*(p + 1) = 42;
*(p + 2) = 3;
for (i = 0; i < 10; i++)
    printf("%d\n", *(p + i));
free(p);
```

L'esempio UNION-FIND, di fatto utilizzava una “memoria statica” (l'array di dimensioni fisse) per rappresentare in memoria degli insiemi e le loro relazioni di appartenenza.

C e C++ ci obbligano a gestire esplicitamente la memoria dinamica (free store o heap). Non esiste la nozione di “garbage collection” come in altri linguaggi.

In C la memoria dinamica viene

allocata e de-allocata usando la coppia di funzioni malloc e free (e derivati).

Questo frammento di codice alloca un puntatore a 10 interi nel free store, assegna i numeri 1, 42 e 3 e poi stampa i 10 elementi.

Alla fine, la memoria viene riposta nel free store con la chiamata free. Il frammento contiene un errore di prassi che poi vedremo.

La funzione malloc restituisce un puntatore di tipo void* ad una zona di memoria, quindi è necessario inserire tutte le necessarie conversioni di tipo per evitare problemi con il compilatore.

Le dimensioni del blocco di memoria ritornato dipendono dal parametro passato alla funzione.

Se non vi è memoria disponibile, malloc restituisce un puntatore nullo; quindi bisogna sempre controllare il risultato di un'allocazione nel free store

```

Int *p = (int*) new int[10];
int i;
*p = 1;
*(p + 1) = 42;
*(p + 2) = 3;
for (i = 0; i < 10; i++)
    printf("%d\n", *(p + i));
delete [] p;

```

In C++ si usano gli operatori `new` e `delete` per manipolare il free store. Il frammento precedente diventa come indicato qua accanto.

Si noti la presenza di [] per denotare la deallocazione di un array. Anche in questo caso è necessario controllare il valore restituito da `new`.

10.11 MODIFICATORI DI DICHIARAZIONE

- **Extern:** la dichiarazione seguente ha una definizione non locale, ovvero la definizione dell'oggetto dichiarato si trova più in là nel file o in un altro file
- **Static:** la dichiarazione o definizione seguente viene “fissata” nello spazio di memoria globale e non è visibile al di fuori del file in cui essa appare

```

#include <stdio.h>

void foo() {
    int a = 10;
    static int sa = 10;
    a += 5;
    sa += 5;
    printf("a = %d, sa = %d\n", a, sa);
}

int main() {
    int i;
    for (i = 0; i < 5; ++i)
        foo();
}

a = 15, sa = 15
a = 15, sa = 20
a = 15, sa = 25
a = 15, sa = 30
a = 15, sa = 35

```

Quindi `extern` si usa essenzialmente per dichiarazioni da mettere in file di interfaccia (header), mentre `static` si usa soprattutto per definizioni globali in un file.

Il modificatore `static` di fatto fissa la dichiarazione o definizione seguente nello “scope” che la racchiude. **Variabili dichiarate static mantengono il loro valore tra una chiamata di funzione e la successiva.**

10.12 COSTANTI

“Const” rende non cambiabile l’attributo di una dichiarazione. **Le costanti devono essere inizializzate per forza, a meno che non siano dichiarate extern.** Cercare di cambiare una costante porta un errore.

Quando si cominciano a considerare i puntatori e le funzioni, le complicazioni aumentano.

Quando si usa un puntatore vi sono due oggetti da considerare:

- Il puntatore
- L’oggetto puntato

La dichiarazione const deve poter distinguere tra i due.

Un importante uso di const è il seguente:

```
char* strcpy (char* p, const char* q);
```

in questo caso q non si può modificare.

Vi è un altro uso di const in congiunzione con i metodi delle classi C++ che non vedremo.

Nonostante la complessità derivante dalle combinazioni di const con puntatori di varia natura, è senz'altro molto utile usare costanti quando ve ne fosse la necessità (ovvero, in alternativa a #define).

10.13 “STREAMS”, INPUT, OUTPUT E FILES

C e C++ hanno librerie molto sofisticate per la gestione di input, output e files.

Le librerie di I/O in C/C++ sono molto legate alla piattaforma sottostante, ed in particolare al “file system”.

In C tutte le operazioni di I/O coinvolgono streams, solitamente associati a files.

Nella libreria C (in stdio.h) esistono tre stream fondamentali:

- Stdin: standard input
- Stdout: standard output
- Stderr: standard error

Altri streams vengono creati ed associati a strutture di tipo FILE (anch'esso definito in stdio.h) tramite le funzioni di libreria.

10.13.1 Output

Le funzioni di output più semplici sono le seguenti:

- **Int fputc(int c, FILE* ostream)**
scrive il carattere c (notare la conversione da int a char) su ostream
- **Int fputs(const char* s, FILE* ostream)**
scrive la stringa s su ostream
- **Int fprintf(FILE* ostream, const char* format, ...)**
scrive la stringa format su ostream dopo averla “interpretata” sulla base degli argomenti forniti. Funziona esattamente come printf, semplicemente lo riversa sul FILE.

La funzione printf che abbiamo già visto è equivalente a fprintf con il primo parametro pari a stdout.

La funzione fprintf interpreta la stringa format sostituendo alle direttive % delle stringhe che dipendono dal parametro corrispondente:

- \n stampa una “newline”
- \t stampa un carattere di tabulazione
- %d stampa un intero int
- %f stampa un float
 - %x.yf dove x è il numero delle cifre intere e y il numero delle cifre frazionarie
 - %g per la rappresentazione normalizzata
- %s stampa una stringa char

10.13.2 Streams e file handlers in C++

In C++ le operazioni di I/O coinvolgono streams che sono istanze di classi di libreria.

Nella libreria C++ (in iostream) esistono tre stream fondamentali:

- `Std::cin`: standard input
- `Std::cout`: standard output
- `Std::cerr`: standard error

Altri streams vengono creati ed associati a files nel file system creando istanze delle classi `ifstream` (input file stream) e `ofstream` (output file stream).

In C++ si usa l'operatore << (detto “put”) per scrivere qualcosa su un output stream. L’operatore è associativo a destra e prende due parametri di input:

- Un output stream
- Un valore di un qualche tipo riconosciuto

ESEMPIO

```
cout << "il numero " << 42 << endl;
cout << 3.14L << endl;
```

la stringa `endl` è di fatto un puntatore ad una funzione, che l’operatore `<<` riconosce e richiama; il suo effetto è di scrivere un carattere newline sullo stream e di assicurarsi che esso sia “svuotato”.

L’operatore `<<` può essere specializzato a seconda dei casi, così come il metodo Java `Object.toString()`.

```
struct person {
    char name[80];
    int age;
    char trusted_assistant[80];
} p = {"Salvo Montalbano", 42, Catarella"};

cout << p << endl
<Person "Salvo Montalbano" "Catarella" 42>
```

Ad esempio, consideriamo una struttura e supponiamo di avere a disposizione una versione specializzata dell’operatore `<<`.

Il richiamare `cout` può stampare direttamente tutti i campi della struttura.

La specializzazione dell’operatore `<<` ha come prerequisito la definizione delle nozioni di `operator` e di `overloading` in C++.

L’overloading consiste nel definire più volte la stessa funzione conferendo però diversi tipi in entrata.

10.13.3 Input in C

In C le funzioni principali dedicate alla gestione di input sono le seguenti:

- **`int fgetc(FILE* istream)`**
legge un carattere da `istream` e lo restituisce
- **`char* fgets(char* s, int n, FILE* istream)`**
legge al più `n` caratteri da `istream` nella stringa `s` e la restituisce; se c’è un newline o se `istream` è alla fine (EOF) l’operazione di lettura si ferma, un puntatore nullo viene restituito in caso di errore. Quindi bisognerebbe sempre controllare il risultato di una chiamata a `fgets`

- **int fscanf (FILE* istream, const char* format, ...)**
legge l'input da istream sotto il controllo del contenuto della stringa format. I parametri passati devono essere dei puntatori ad aree di memoria dove è possibile depositare il valore letto. Fscanf è in grado di interpretare le stringhe lette in input e di tradurle nel formato interno corretto

La funzione scanf è simmetrica rispetto a printf: scanf ("%d", &x) === fscanf (stdin, "%d", &x).

10.13.4 Input in C++

In C++ si usa l'operatore **>>** (detto "get") per leggere qualcosa da un input stream. L'operatore è associativo a destra e prende due parametri di input:

- Un input di stream
- Un puntatore (o una referenza) ad un oggetto di qualche tipo riconosciuto

ESEMPIO

```
Int x;
cin >> x;
cout << "il numero è " << x << endl;
```

```
istream& operator >> (istream& s,
complex& a) {
    double re = 0, im = 0;
    char c = 0;

    s >> c;
    if (c == '(') {
        s >> re >> c;
        if (c == ' ') s >> im c;
        if (c != ')')
s.clear(ios_base::badbit);
    } else {
        s.putback(c);
        s >> re;
    }
    If (s) a = complex(re, im);

    return s;
}
```

ora un caso più complicato. Costruiamo un operatore speciale per la lettura di numeri complessi della forma:

- f
- (f)
- (f, f)

Dove f è un float.

Assumiamo:

- l'esistenza di una classe complex con relativi costruttori
- di conoscere la semantica della creazione di operatori specializzati tramite la dichiarazione operator

10.14 FILE I/O IN C

Le funzioni di input e output agiscono su stream che possono essere associati a files. Per associare uno stream ad un file si usa la funzione **fopen**.

Per rompere questa associazione (ovvero per chiudere lo stream) si usa la funzione **fclose**.

Un file può essere distrutto usando la funzione **remove**.

La funzione fopen:

```
FILE* fopen(const char* filename, const char* mode);
```

il parametro filename è il nome completo del file da “aprire”.

Il parametro mode controlla come il file viene aperto:

- “r”: apre un file di testo in sola lettura
- “w”: apre azzerando un file già esistente o crea un file in scrittura
- “a”: “append”, apre o crea un file di testo in scrittura (alla fine del file)

fopen ritorna un puntatore ad uno stream FILE od il puntatore nullo se viene segnalato un qualche errore.

La funzione fclose:

```
int fclose(FILE* stream);
```

semplicemente segnala al file system che il file associato a stream non verrà più usato dal programma a meno che non venga riaperto.

Il valore ritornato è 0 se la chiamata va a buon termine, EOF in caso contrario.

10.14.1 File I/O in C++

Supponiamo che bat.txt contenta la stringa 42

```
ifstream bar("bar.txt");
int qd;

bar >> qd;
bar.close();
cout << qd;

// viene stampato 42 a schermo

ofstream("bar.txt", "a");
bar << '\n' << "Vogons" << endl
bar.close();

// ora il file contiene anche "Vogons" preceduta da un
newline
```

In C++ l'input e l'output su files si basa sulla costruzione di istanze delle classi

ifstream ed ofstream (“input file stream” e “output file stream”), definite in iostream.h.

In quanto streams, tali istanze possono essere manipolate tramite gli operatori >> e <<.

10.15 ABBREVIAZIONI: TYPEDEF

Il nome FILE in <stdio.h> in realtà è un'abbreviazione di una struttura più complessa. Queste abbreviazioni si creano usando la direttiva typedef con una sintassi che ricorda le dichiarazioni.

ESEMPIO

```
typedef char buffer[1024];
buffer x;
```

x è dichiarata di fatto come char x[1024]

11 CODE DI PRIORITÀ (PRIORITY QUEUES) E C

11.1 CODE CON PRIORITÀ

Unica assunzione: gli elementi che sono parte dell'input sono confrontabili

Operazioni fondamentali:

- Operazioni proprie dell'ADT¹²
 - Inserimento
 - Rimozione del massimo/minimo
- Operazioni generiche su ADT
 - Copia
 - Concatenazione
 - Creazione
 - Distruzione
 - Controllo se vuoto
- Applicazioni
 - simulazione "event-driven"
 - Es. clienti in coda
 - "number crunching"
 - Es. riduzione di errori di arrotondamento
 - Compressione di dati
 - Es. codici di Huffman¹³
 - Ricerca su grafi
 - Es. Algoritmi di Dijkstra¹⁴ e Prim¹⁵
 - Teoria dei numeri
 - Es. Somma di potenze
 - Intelligenza artificiale
 - Es. algoritmo A*¹⁶
 - Statistica
 - Es. Manutenzione degli M valori più grandi
 - Sistemi operativi
 - Es. "load balancing", gestione interruzioni
 - Ottimizzazione discreta
 - Es. bin packing¹⁷, scheduling

¹² Un tipo di dato astratto o ADT (Abstract Data Type) è un tipo di dato le cui istanze possono essere manipolate con modalità che dipendono esclusivamente dalla semantica del dato e non dalla sua realizzazione. È un modo per ragionare senza concentrarsi sul tipo di dato ma sulla sua implementazione più generica.

¹³ Per Codifica di Huffman si intende un algoritmo di codifica dei simboli usato per la compressione di dati, basato sul principio di trovare il sistema ottimale per codificare stringhe basato sulla frequenza relativa di ciascun carattere

¹⁴ L'algoritmo di Dijkstra è un algoritmo utilizzato per cercare i cammini minimi in un grafo con o senza ordinamento, ciclico e con pesi non negativi sugli archi.

¹⁵ L'algoritmo di Prim è un algoritmo ottimo utilizzato in teoria dei grafi, informatica e ricerca operativa per determinare gli alberi di supporto minimi di un grafo non orientato e con pesi non negativi.

¹⁶ Nell'informatica, A* (pronunciato "A star" in inglese) è un algoritmo di ricerca su grafi che individua un percorso da un dato nodo iniziale verso un dato nodo goal (o che passi un test di goal dato). Utilizza una "stima euristica" che classifica ogni nodo attraverso una stima della strada migliore che passa attraverso tale nodo.

¹⁷ Oggetti di differente dimensione devono essere immagazzinati in una serie di "cestini" di volume V in modo che venga minimizzato l'uso di cestini. È un problema NP-completo

- Filtri spam
 - Filtri bayesiani¹⁸
- Generalizza
 - Pile, code, code randomizzate

11.2 ESEMPI

11.2.1 Ricerca degli elementi più grandi

Problema:

- Abbiamo un flusso di n dati in arrivo con N molto più grande della disponibilità di memoria del nostro calcolatore
- Dobbiamo recuperare gli M elementi più grandi
- Esempi di utilizzo:
 - Rilevamento frodi: isolare le transazioni più grandi
 - Manutenzione file system: trovare i files o le directories più grandi

Soluzione: si usa una coda con priorità

```
int main() {
    pq_priority_queue pq = pq_make_priority_queue(str_less, M+1);
    //assumiamo di avere questa funzione

    char in[80];

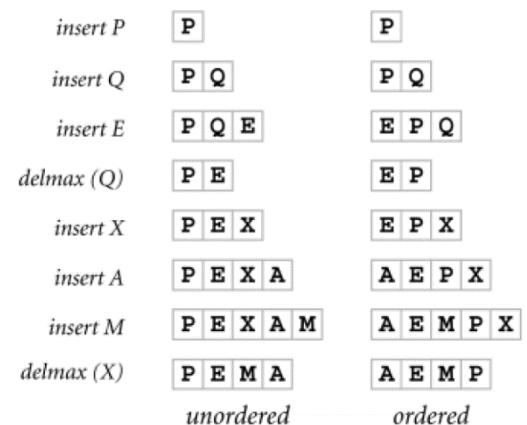
    while((scanf("%s", in) != EOF) {
        pq_insert(pq, in);
        if(pq_count(pq) > M)
            pq_extract(pq);
    }
    while (! pq_is_empty(pq))
        printf("%s", (char*) pq_extract(pq));
    free(pq);
}
```

11.2.2 Implementazioni elementari

Sfida: implementare entrambe le operazioni base in modo efficiente

Implementazione elementare con un array ordinato o no.

| Implementazione | Insert | Extract |
|---|--------|---------|
| Array non ordinato | O(1) | O(N) |
| Array ordinato | O(N) | O(1) |
| Caso asintotico peggiore con N elementi | | |



¹⁸ Il filtro bayesiano applica all'analisi delle email un teorema, espresso per l'appunto da Bayes, secondo il quale ogni evento cui è attribuita una probabilità è valutabile in base all'analisi degli eventi già verificatisi.

```

typedef int (* pq_compare) (void*, void*);

typedef struct _pq {
    void** pq;
    int size;
    int count;
    pq compare compare;
} * pq priority queue; //pq priority queue è il nome del tipo

//costruttore
pq_priority_queue pq_make_priority_queue(pq_comare cd, int size){
    pq_priority_queue pq
    = (pq_priority_queue) malloc(sizeof(struct _pq));
    /* controllo eventuali errori di allocazione memoria */
    pq -> pq = malloc(size * sizeof(void *));
    /* controllo evenutali errori di allocazione memoria */
    pq -> compare = cf;
    pq -> count = 0; //estrae da pq il valore count e gli associa 0
    pq -> size = size;
    return pq;
}
//metodi ausiliari
void pq_insert(pq_priority_queue pq, void* item){
    pq -> pq[(pq -> count)++] = item;
}

void pq_count(pq_priority_queue pq){
    return pq -> count;
}

bool pq_is_empty(pq priority queue pq){
    return 0 == pq -> count;
}

void* pq_extract(pq_priority_queue pq){
    int e = 0;
    int i;
    for(i = 0; i < pq->count; i++)
        if(pq -> compare(pq -> pq[i], pq -> pq[e]))
            e = i;
    exchange(pq -> pq, e, pq -> count-1);
    return pq -> pq[--(pq->count)];
}

```

12 LA STRUTTURA DATI HEAP BINARIO

Gli heaps ci permettono di implementare le operazioni pq_extract e pq_insert in tempo logaritmico.

Uno heap viene implementato (di solito) con un array il cui contenuto è interpretabile come un albero binario completo in “heap-order”.

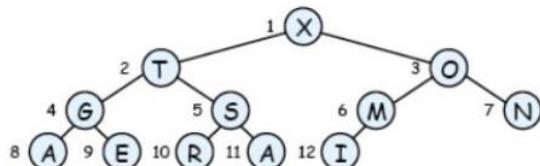
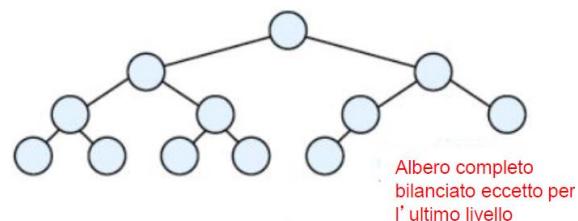
- Albero binario
 - Vuoto, oppure
 - Nodo con puntatori a due sottoalberi (destro e sinistro)
- Albero binario in heap order
 - Elementi nei nodi (chiavi)
 - L’elemento nel nodo genitore deve essere maggiore degli elementi nei nodi figli
- Albero binario di ricerca
 - Albero binario in heap order dove il nodo figlio sinistro è minore o uguale a quello a destra

Rappresentazione con un array: si considerano i nodi per livello, nessun puntatore è necessario dato che l’albero è completo.

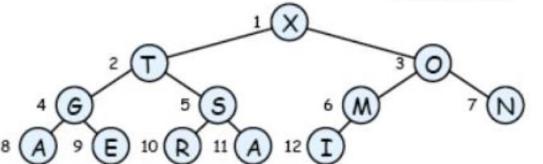
12.1 PROPRIETÀ DEGLI HEAP (MUCCHI) BINARI

Consideriamo per le spiegazioni un albero binario max-heap (ovvero il valore più alto è alla radice dell’albero). Nel caso di min-heap il massimo diventa il minimo.

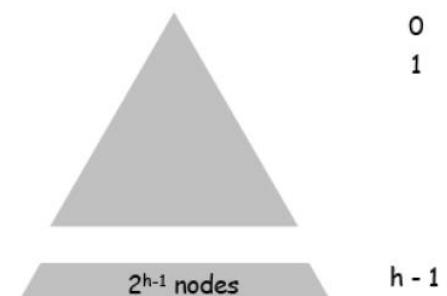
- Proprietà A: la chiave più grande è alla radice dell’albero
- Proprietà B: gli indici dell’array si possono usare per muoversi nell’albero
 - Gli indici partono da 1
 - Il genitore del nodo si trova in posizione N, il figlio sinistro a $2N$ e quello destro a $2N+1$
- Proprietà C: l’altezza di un heap è: $h = 1 + \log(N)$



| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|----|----|----|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| X | T | O | G | S | M | N | A | E | R | A | I |



| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|----|----|----|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| X | T | O | G | S | M | N | A | E | R | A | I |



Gli heap vengono utilizzati ad esempio per l'

13 FUNZIONI C UTILI (APPUNTI ECONOMIA)

13.1 INCLUSIONE LIBRERIE

- #define <stdio.h>
- #define <stdlib.h>
- #define <math.h>

```
Int main () {
    ...
}
```

13.2 DICHIARAZIONE TIPI

- **Int**
 - Numeri interi
 - Chiamata: %d
 - **Float**
 - Numeri reali
 - Chiamata:
 - %f per il numero completo
 - %x.yf dove x è il numero delle cifre intere e y il numero delle cifre frazionarie
 - %g per la rappresentazione normalizzata
 - **Double**
 - Numeri reali in virgola mobile (doppia dimensione rispetto a float)
 - Chiamata: %f o %lf (long float)
 - **Char**
 - Caratteri
 - Chiamata: %c
 - **Array**
 - Insieme di dati dello stesso tipo, organizzati in un vettore
 - int a[i][j]....
dove i e j sono numeri o costanti indicate precedentemente
 - **Stringhe**
 - figura mista tra array e char
 - char frase[] = "..."
 - chiamata: %c
 - **typedef**
 - definizione di un nuovo tipo
 - typedef tipo*nome
nome tipo valori
- es.
- ```
typedef char*stringa
stringa esempio
```

- **struct**
  - struttura una serie di dati di tipo diverso, simile all'array
 

```
struct nome {
 tipo dato
 tipo dato
 tipo dato
}
```
  - chiamata: nome.dato
- **puntatori**
  - stampa l'indirizzo del puntatore
  - `Printf("%p", (void*) nome)`

```
MakeLists.txt x main.c x
#include <stdio.h>

int main() {
 int n = 5;
 int* p = &n;
 printf("Il valore di n e' %d \n", n); // %d stampa un intero, n
 printf("Il valore di n e' %d \n\n", *p); // *p è il puntatore a n

 printf("L'indirizzo di p e' %p \n", &p); // &p è il reference di p: l'indirizzo a cui è salvato
 printf("L'indirizzo di n e' %p \n\n", &n); // &n è il reference di n: l'indirizzo a cui è salvato

 printf("n e' salvato alla cella numero %p\n", p); // p è il valore del puntatore, l'indirizzo di n
 return 0;
}
```

## 13.3 OPERATORI

### 13.3.1 Printf

Mostra i dati inseriti a schermo

```
printf("... %chiamata", variabile)
```

- `\n` stampa una “newline”
- `\t` stampa un carattere di tabulazione
- `%d` stampa un intero int
- `%f` stampa un float
  - `%x.yf` dove x è il numero delle cifre intere e y il numero delle cifre frazionarie
  - `%g` per la rappresentazione normalizzata
- `%s` stampa una stringa char

### 13.3.2 scanf

registra i dati inseriti da un input

```
scanf("%chiamata", variabile)
```

- `%d` legge un intero int
- `%f` legge un float
  - `%x.yf` dove x è il numero delle cifre intere e y il numero delle cifre frazionarie
  - `%g` per la rappresentazione normalizzata
- `%s` legge una stringa char

### 13.3.3 if - else

- condizione
- if(condizione)
 

```
{conseguenze;}
```
- per segnare cosa fare nel caso non si avveri if

```
if...
else...
```

- ogni else è collegato direttamente all'if che lo precede
- per proseguire una catena di if

```
if...
else if
```

### 13.3.4 switch-case

- a seconda del valore di una costante cambia lo switch
- dichiarazione variabile scelta e poi i vari casi

```
switch (a) {
 case 1: ...
 break
 case 2: ...
 break
 case 3: ...
 break
}
```

### 13.3.5 for

- ciclo condizionato

```
for(1; 2; 3){
 istruzioni
}
```

dove 1 è la partenza, 2 è la condizione e 3 è l'istruzione a fine ciclo  
possono essere omesse scrivendo solo {;;}

### 13.3.6 while

- condizione che se risulta vera attiva il ciclo, altrimenti passa oltre
- while(condizione)
 

```
{espressione;}
```
- può sostituire il for così:

```
1
while (2) {... 3; }
```

### 13.3.7 do-while

- quando l'istruzione nel ciclo deve essere eseguita almeno una volta si utilizza questo costrutto

```
do {
 espressione
} while (condizione);
```

### 13.3.8 break

- interrompe l'esecuzione del case, provocando un salto alla prima istruzione successiva
- può essere usato per forzare l'uscita da for, do-while, while

### 13.3.9 continue

- continua il ciclo con una nuova iterazione a partire dall'inizio, saltando le istruzioni successive

### 13.3.10 exit

- termina immediatamente il programma
- exit (0) lo termina con un messaggio di errore
- exit (1) lo termina senza ulteriori messaggi

### 13.3.11 goto

- porta l'esecuzione ad un punto indicato tramite un tag saltando i passaggi centrali
- if (...) goto tag
- ...
- ...
- tag: istruzioni successive

### 13.3.12 system ("PAUSE")

- interrompe l'esecuzione del programma fino a quando l'utente non la fa ripartire

## 13.4 OPERATORI LOGICI E MATEMATICI

- **==**: uguaglianza
- **!=**: diversità
- **<, <=, >, >=**: minore, minore o uguale, maggiore o uguale, maggiore
- **/, %**: div (risultato divisione), mod (resto divisione)
- **&&**: e
- **||**: o
- **n++, n--**: n+1, n-1, entrambi eseguiti dopo la lettura di n
- **++n, --n**: n+1, n-1, entrambi eseguiti prima della lettura di n
- **pow(a, b)**: potenza con base a ed esponente b
- **sqrt(a)**: radice quadrata di a

## 13.5 PUNTATORE

Per passare un dato “per valore” (sostituendo quindi la cella indicata) e non “per indirizzo” (indicando la cella) si usano i puntatori

Usare **&variabile** indica alla variabile puntatore dove inviare il risultato. L'effetto complessivo è semplicemente che le variabili sono in “collegamento” tra di loro, ogni modifica fatta alla variabile puntatore influenza anche la variabile puntata

Il complementare è \* che dice alla variabile quale funzione leggere

### 13.6 FUNZIONI

Vengono programmate dopo il main, sono funzione ausiliarie che possono essere chiamate se precedentemente dichiarate all'interno del main

```
Int main() {
 Dichiarazione prototipo della funzione con il suo nome
 ...
 Chiamata funzione (variabili da leggere)
}
```

```
Tipo_di_ritorno nome_funzione (Tipo_variabile nome_locale, ...){
 ...
 return dato da mandare alla funzione
}
```

## 14 LEZIONE RIASSUNTO

---

Oggetti autovalutanti: numeri, stringhe, lambda, T, Nil, keywords.

Passi dell'interprete:

- Definizione
- Applicazione
- Valutazione

Espressione simbolica (symbolic expression): ad esempio defun è un simbolo che denota una funzione.

Una symbolic expression è una lista, infatti in lisp programmi e liste coincidono. Dopo la prima parentesi devo specificare un simbolo che denota una funzione ad esempio. La valutazione di LISP controlla in sequenza i valori della lista/programma.

La funzione eval va a scandire tutti gli elementi della lista e ad associare i valori agli elementi della lista, la funzione apply va ad applicare la funzione corpo ai valori degli elementi.

Atomi: valori o simboli.

La differenza tra valori e simboli: ai simboli viene associato un oggetto (tipo simboli di funzione o variabile).

Un simbolo, quando viene valutato, produce il valore associato al simbolo. Il valore quando viene valutato produce sé stesso.

I linguaggi funzionali puri hanno la proprietà di trasparenza referenziale: ci sono dei simboli e io so in ogni momento quale valore sia associato a questi. In common lisp definire delle variabili fa venire meno il concetto di trasparenza referenziale in quanto il valore di una funzione può non essere sempre lo stesso.

Ricordarsi che, per esempio, (defparameter a 'funzione...) associa ad a non la funzione ma solo il suo nome.

Funzioni di ordine superiore: funzione che può avere come parametro formale altre funzioni o che può produrre altre funzioni come risultato.

Esempio: compose

```
(defun compose (f g) (funcall f (funcall g)))
(funcall (compose 'succ 'succ) 1) → 3
```

Si usa il quote perché compose si aspetta che vengano passati i nomi delle funzioni.

Se nel testo c'è "applica il nome di una funzione ad un argomento" bisogna usare funcall.

Se nel testo c'è "dati i nomi di due funzioni costruire una funzione che li componga" bisogna usare funcall.

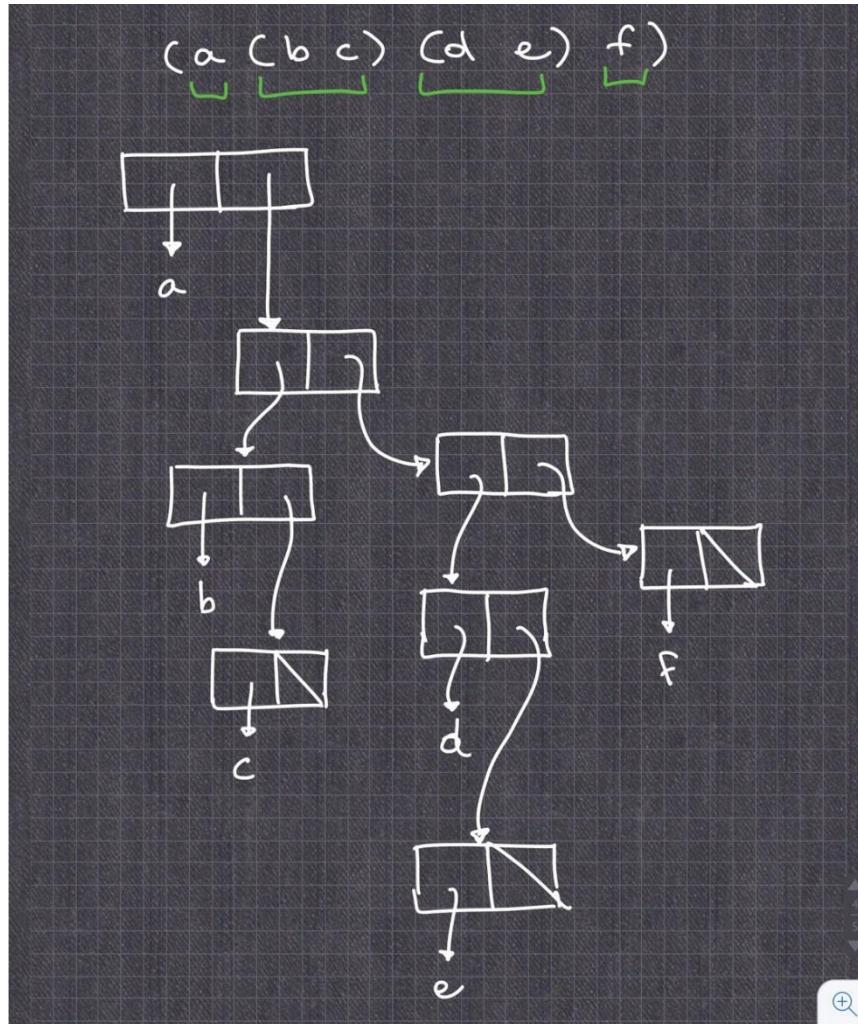
Domande papabili:

- Spiegare il concetto di trasparenza referenziale
- Activation frame
- Programma da sviluppare in LISP
- Domanda semplice su C o definizione di una semplice variabile
- Cons-cell
- Equivalenza lambda-let

Cons cell: è una coppia di puntatori.

```
CL-USER 6 > (cons 'a (cons 'b (cons (cons 'c (cons 42 'f))
 (cons (cons 'd (cons (cons (cons 'e nil) nil) nil) nil)
 nil))))
```

(A B (C 42 . F) (D ((E))))



#### Equivalenza lambda-let

|                                                                                                                                                    |                                                                                                                                    |
|----------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------|
| <pre>(defun f (x y)   (let ((a (+ 1 (* x y)))         (b (- 1 y))         )       (+ (* x (quadrato a))          (* y b)          (* a b))))</pre> | <pre>(defun f (x y)   ((lambda (a b)     (+ (* x (quadrato a))        (* y b)        (* a b)))    (+ 1 (* x y))    (- 1 y)))</pre> |
|----------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------|

Se l'interprete trova una lambda expression, allora definirà una closure/chiusura, ovvero riserverà uno spazio di memoria al contenimento dell'associazione dei parametri formali.

Apply è l'ultimo passo della valutazione perché prima devono essere valutati tutti i valori.

**Apply:** dato il nome di una funzione dei valori applica la funzione ai valori. I valori devono essere autovalutanti, non possono essere richiami ad altre funzioni. Ad esempio (apply ‘somma (succ 1) (succ 1)) è sbagliato. In più apply richiede una lista di argomenti e non argomenti sciolti.

La differenza con funcall è proprio che questa permette di valutare il contenuto delle parentesi.

```
CL-USER 16 : 5 > (apply '+ '(1 2))
3

CL-USER 17 : 5 > (funcall '+ 1 (+ 1 1))
3

CL-USER 18 : 5 > (apply '+ '(1 (+ 1 1)))
Error: In + of (1 (+ 1 1)) arguments should be of type NUMBER.
```

#### Ricorsioni:

- **Semplice:** ricorsione solo sul cdr, agisce solo sui dati al primo livello
- **Doppia:** ricorsione car/cdr, agisce a tutti i livelli
- **In coda:** la ricorsione deve essere l’ultima chiamata in assoluto della funzione. Se la ricorsione viene usata da un’altra funzione (ad esempio (append (funz-ricorsiva ...))) non è in coda in quanto l’ultima istruzione eseguita sarà append.

Il problema della ricorsione doppia è che bisogna pensare bene a come combinare le chiamate. Bisogna distinguere quindi tra i vari casi:

- Numberp: controlla se l’atomo è un numero
- Oddp: controlla se l’atomo è un numero dispari
- Evenp: controlla se l’atomo è un numero pari
- Listp: controlla se l’atomo è una lista
- Stringp: controlla se l’atomo è una stringa

## APPENDICE A: EMACS E PROLOG

### Appunti di laboratorio di Linguaggi di Programmazione Prolog

- Per avviare emacs su SWI-Prolog scrivere "emacs." + invio
- Dalla finestra emacs: new file > nome.pl (**NON DIMENTICARE L'ESTENSIONE DEL FILE**)

NB: le variabili si scrivono con l'iniziale maiuscola

commenti in prolog con %

nel caso ci siano più risultati possibili scrivere ; dopo ogni risultato per scorrere l'elenco.

|                                                    | Emacs                                                                                     | Prolog                                                                                  |  |
|----------------------------------------------------|-------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------|--|
| database person a e azienda-> scrivo una relazione | <pre>lavora_per(bill, google). lavora_per(jane, google). lavora_per(mario, oracle).</pre> | <pre>?- lavora_per(bill, X). X = google.</pre>                                          |  |
|                                                    | <pre>collega(X, Y) :- lavora_per(X, Z), lavora_per(Y, Z), X \= Y.</pre>                   | <pre>?- collega(bill, jane). true.</pre>                                                |  |
| Numeri naturali: 0 naturale, s(0) numerale ecc.    | <pre>naturale(0). naturale(s(N)) :- naturale(N).</pre>                                    | <pre>?- naturale(0). true.  ?- naturale(s(0)). true.  ?- naturale(s(s(0))). true.</pre> |  |
| somma                                              | <pre>succ(N, s(N)).  sum(N, 0, N). sum(N, s(M), s(T)) :- sum(N, M, T).</pre>              | <pre>?- sum(s(0), s(s(0)), s(s(s( true</pre>                                            |  |

|                                              |                                                                                                                                                                                                  |                                                      |                                                                                                       |
|----------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------|-------------------------------------------------------------------------------------------------------|
|                                              |                                                                                                                                                                                                  |                                                      |                                                                                                       |
| fattoriale                                   | <pre> <b>fact(0, 1).</b> <b>fact(N, M) :- N&gt;0,</b> <b>          N1 is N-1,</b> <b>          fact(N1, M1),</b> <b>          M is N*M1.</b> </pre>                                              | <pre>?- fact(10, X). X = 3628800 .</pre>             |                                                                                                       |
| Liste n-esimo elemento (la lista parte da 0) | <pre> <b>in_testa(X, [X _]).</b> <b>nth(0, [X _], X).</b> <b>nth(N, [_ T], X) :- N1 is N-1,</b> <b>                     nth(N1, T, X).</b> </pre>                                                | <pre>?- nth(2, [5, 8, 3, 9], 3). <b>true</b></pre>   | <p>X è la testa (primo elemento della lista)<br/>   — significa che non ci interessa quella parte</p> |
|                                              | <pre> <b>contains(X, [X _]).</b> <b>contains(X, [_ T]) :- contains(X, T).</b> </pre>                                                                                                             | <pre>?- contains(9, [2,5,7,8,2,9], <b>true</b></pre> |                                                                                                       |
|                                              | <pre> <b>append([], L, L).</b> <b>append([H T], L, [H X]) :- append(T, L, X).</b> </pre>                                                                                                         | <pre>?- append([1,2,3], [4,6], [1 <b>true</b>.</pre> |                                                                                                       |
|                                              | <pre> <b>sorted([]).</b> <b>sorted([_]).</b> <b>sorted([X,Y T]) :- X &lt;= Y,</b> <b>                      sorted([Y T]).</b> </pre>                                                             | <pre>?- sorted([2,3,5,7]). <b>true</b> ;</pre>       |                                                                                                       |
|                                              | <pre> <b>last([X], X).</b> <b>last([_ T], X) :- last(T, X).</b> </pre>                                                                                                                           | <pre>?- last([1,4,7,2,0,5], 5). <b>true</b></pre>    |                                                                                                       |
|                                              | <pre> <b>remove_all([], _, []).</b> <b>remove_all([X T], X, L) :- remove_all(T, L).</b> <b>remove_all([H T], X, [H L]) :- X \= H,</b> <b>                           remove_all(T, L).</b> </pre> |                                                      |                                                                                                       |
|                                              | <pre> <b>somma_lista([], 0).</b> <b>somma_lista([H T], N) :- somma lis</b> <b>                    N is H+N1.</b> </pre>                                                                          |                                                      |                                                                                                       |

|  |                                                                            |                                              |  |
|--|----------------------------------------------------------------------------|----------------------------------------------|--|
|  | <b>duplica([],[]).</b><br><b>duplica([Y T], [Y,Y T1]):-</b> <u>duplica</u> | ?- duplica([1,2],[1,1,2,2]).<br><b>true.</b> |  |
|--|----------------------------------------------------------------------------|----------------------------------------------|--|

Finire un predicato min che, data una lista, restituisce il minimo

Es.

```
?- min([1, 2, 3, 4], X).
```

```
X = 1
```

```
% lista vuota, non serve precisarlo perché se non viene trovato un fatto
% che rispecchi la ricerca, questa restituirà false per default
% min([], false)

%lista con un solo elemento
min([X], X).

%caso ricorsivo 1: minimo in testa alla lista
min([H|T], H) :-
 min(T, X),
 X >= H.

% quindi il minimo è nella testa H, mentre nella coda ci sono solo
% numeri maggiori o uguali

%caso ricorsivo 2: minimo nella coda della lista
min([H|T], X) :- min(T, X),
 X < H.

% in questo caso la testa della coda non era il minimo, quindi continua
% a scorrere la coda T
```

Scrivere nel caso ricorsivo [T] è sbagliato perché la coda di una lista è una lista, quindi scrivendo [T] sarebbe come descrivere una lista con un solo elemento che consiste in una lista.

```
selection_sort([], []).
selection_sort(X, [H|T]) :-
 min(X, H), %trova il minimo della lista iniziale
 remove_one(X, H, Y), %rimuove dalla lista iniziale la testa e assegna a Y la nuova lista
 selection_sort(Y, T). %continua il selection sort sulla nuova lista Y

%merge(prima lista, seconda lista, lista unita).

%caso con le liste vuote
merge([], X, X).
merge(X, [], X).

%casi di unione di liste precedentemente ordinate
merge([H1|T1], [H2|T2], [H|T]) :-
 H1 <= H2,
 H is H1,
 merge(T1, [H2|T2], T).

merge([H1|T1], [H2|T2], [H|T]) :-
 H1 > H2,
 H is H2,
 merge([H1|T1], T2, T).

% prima controlla quale delle due teste sia minore, la aggiunge alla
% lista finale e poi continua con la coda della lista da cui abbiamo
% attinto il valore, l'altra lista non toccata, la coda della lista
% finale
```

Scrivere una regola che elimina un valore dalla lista ma solo nella prima posizione nella quale compare.

```
% remove_one([1, 2, 2, 2], 2, X).
% X = [1, 2, 2].
%
% struttura --> remove_one(lista_iniziale, elem. da rimuovere,
% lista_risultante).
%
% caso della lista vuota: qualsiasi sia l'elemento da rimuovere, la
% lista rimane vuota
remove_one([], _, []).
%
%caso con singolo elemento da eliminare
remove_one([X], X, []).
%
%caso con singolo elemento non da eliminare
remove_one([X], Y, [X]) :-
 X \= Y.
%
%caso dove X è la testa della lista
remove_one([H|T], X, T) :-
 H = X.
%
%caso dove X è nella coda
remove_one([H|T], X, [H|Z]) :-
 H \= X,
 remove_one(T, X, Z).
%
% nel caso il valore da rimuovere non sia la testa, la query verrà
% continuata con solo la coda fino a quando non troverà (eventualmente)
% la variabile. a quel punto restituirà solo la coda della sottolista
% senza la testa e questa verrà attaccata alla lista precedente
```

Selection\_sort(lista da ordinare, lista ordinata)

```
%split_in_two(lista da dividere, prima metà, seconda metà).

%casi base
split_in_two([], [], []).
split_in_two([X], [X], []).

%caso ricorsivo
split_in_two([H1, H2|T], [H1|T1], [H2|T2]) :-
 split_in_two(T, T1, T2).
```

```

%mergesort(lista non ordinata, lista ordinata)
% 1)divide in 2 la lista
% 2)chiama mergesort sulle due liste risultanti
% 3)fa il merge delle due liste

%casi base
mergesort([], []).
mergesort([X], [X]).

mergesort(X, L):-

 split_in_two(X, L1, L2),

 mergesort(L1, L1x),

 mergesort(L2, L2x),

 merge(L1x, L2x, L).

% controllo se la lista è una lista in base alla sua definizione, ovvero
% se è divisibile in testa e coda

listp([]).
listp([_|_]).

%attacca la seconda lista in coda alla prima
append([], X, X).
append([H|T], X, [H|Y]):-
 append(T, X, Y).

% "appiattisce le liste", ovvero le porta tutte allo stesso livello e le
% unisce in un'unica lista

%flatten(lista innestata, lista restituita)
%caso base
flatten([], []).

flatten([H|T], L):-

 listp(H), %controlla che la testa sia una lista
 flatten(H, X), %appiattisce la testa
 flatten(T, Y), %appiattisce la coda
 append(X, Y, Z). %concatena le due liste create dall'appiattimento

% se la testa non è una lista fallisce il controllo precedente, quindi
% passa a questa clausola unificatrice
flatten([H | T], [H | X]):-
 flatten(T, X). %tiene la testa uguale e prosegue sul resto della lista

```

## B CODICI LISP

---

```
(defun fact (n)
 (if (= n 0)
 1
 (* n (fact (- n 1)))))

(defun fact-iter (n acc)
 (if (= n 0)
 acc
 (fact-iter (- n 1)
 (* n acc)))))

(defun my-list-length (lst)
 (if (null lst)
 0
 (+ 1
 (my-list-length
 (cdr lst)))))

(defun my-list-length-iter (lst acc)
 (if (null lst)
 acc
 (my-list-length-iter (cdr lst) (+1 acc)))))

(defun double-list (lst)
 (if (null lst)
 nil
 (cons (* 2 (car lst))
 (double-list (cdr lst))))))

(defun double-list-iter (lst)
 (let ((nuova-testa (* 2 (car lst)))
 (coda (unless (null lst) (cdr lst))))
 (if (null lst)
 nil
 (cons nuova-testa (double-list coda)))))

(defun prendi-positivi (lst)
 (cond ((>= (car lst) 0) (cons (car lst)
 (prendi-positivi (cdr lst))))
 ((< (car lst) 0) (prendi-positivi (cdr lst)))))

(defun filtra-lista (lst func)
 (cond ((null lst) nil)
 ((funcall func (car lst))
 (cons (car lst)
 (filtra-lista (cdr lst) func)))
 (t (filtra-lista (cdr lst) func))))

(defun applica-parzialmente (func arg)
 (lambda (x) (funcall func arg x)))
```

## C CODICI C/C++

```
#include <stdio.h>
#include <stdlib.h>
/*Scrivere un programma in C che legge due numeri interi a, b e stampa il più
piccolo */
int main () {

 int a, b;

 printf("Digita due numeri interi separati da uno spazio\n");
 scanf ("%d %d", &a, &b);

 if (a<b)
 printf("Il minore dei due numeri inseriti è %d\n", a);
 else
 printf ("Il minore dei due numeri inseriti è %d\n", b);
 system ("PAUSE");
}

#include<stdio.h>
#include<stdlib.h>
#include<math.h>
/* creare un programma che stampa la media di
n numeri inseriti dall'utente */
int main() {
 int x,somma = 0, n = 0;
 double media = 0;

 do{
 scanf("%d", &x);
 if (x != 0)
 {somma=somma+x;
 n=n+1;}
 }while (x != 0);

 /*legge fino allo 0 escluso*/
 if (n!=0) {
 media = (somma*1.0)/n;
 printf ("%lf\n", media);
 }
 system ("PAUSE");
 return 0;
}

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
/* Scrivere un programma in C che legge 3 numeri interi a, b, c e stampa le
soluz
di secondo grado $ax^2+bx+c=0$, se non esistono scrive "Non ha soluzioni nel
campo */
int main () {
 int a,b,c,delta;
 double x1,x2;

 scanf ("%d%d%d", &a, &b, &c);
 delta=(b*b)-(4*a*c);

 if(delta<0)
 printf("l'equazione non ha soluzione\n");
 if(delta==0)
 if (a != 0) {
```

```

 x1= -b/(2*a*1.0);
 printf("%lf\n",x1);
 }
 if (delta > 0)
 if (a != 0) {
 x1=(-b+sqrt(delta))/(2*a);
 x2=(-b-sqrt(delta))/(2*a);
 printf("%lf %lf\n",x1,x2);
 }
 if (a==0)
 if (b !=0)
 printf ("%lf\n", -c/(b*1.0));
 system("PAUSE");
 return 0;
}

#include <stdio.h>
#include <stdlib.h>
/*stampa la diagonale della tabellina in un array*/
int main (){

 int a[10][10], i, j;

 for (i=0; i<=9; i++)
 for (j=0; j<=9; j++)
 a[i][j]=(i+1)*(j+1);

 for (i=0; i<=9; i++){
 for (j=0; j<=9; j++)
 if (i==j)
 printf("%3d ",a[i][j]);
 else printf(" ");
 printf("\n");
 }
 system ("PAUSE");
 return 0;
}

#include <stdio.h>
#include <stdlib.h>
/* ordina un array con algoritmo bubblesort */
int main () {
 int a[10], i, j, temp;

 for (i=0; i<=10; i++)
 a[i]=rand();

 for (i=0; i<=9; i++)
 printf("%d ", a[i]);

 printf("\n\n");
 system("PAUSE");
 printf("\n\n");

 for (i=0; i<=9; i++)
 for (j=9; j>i; j--)
 if (a[j] < a[j-1]){
 temp =a[j-1];
 a[j-1]=a[j];
 a[j]=temp;
 }
 for (i=0; i<=9; i++)
 printf("%d ", a[i]);
}

```

```

 printf ("\n\n");
 system ("PAUSE");
 return 0;
 }

#include<stdio.h>
#include<stdlib.h>
#include<math.h>
/* Cercare il minimo tramite l'uso di funzioni ausiliarie */

// la funzione controlla i due valori e restituisce solo quello minore
int minimo (int a, int b){
 int r;
 if (a < b)
 r=a;
 else
 r=b;
 return r;
}

// la funzione controlla i due valori e assegna al puntatore c il minimo
// non ha un tipo di ritorno poichè sfrutta il puntatore per modificare
// il minimo all'interno del main
void minimol (int a, int b, int *c){
 if (a < b)
 *c=a; // assegna alla cella puntata il valore a
 else
 *c=b; // assegna alla cella puntata il valore b
}
int main(){
 int x,y,z, t1,t2;

 printf("Inserire tre numeri:\n");
 scanf("%d%d%d", &x, &y, &z);

 minimol(x,y, &t1); // utilizzando dei puntatori abbiamo una
 minimol(t1,z, &t2); // modifica nel main da parte del metodo ausiliario
 printf ("il minimo fra %d %d e %d e' %d\n",x,y,z,t2);

 t1 = minimo(x,y); // assegna direttamente a t1 il minimo
 t2 = minimo(t1,z); // assegna direttamente a t2 il minimo
 printf ("il minimo fra %d %d e %d e' %d\n",x,y,z,t2);

 printf ("il minimo fra %d %d e %d e' %d\n",x,y,z,minimo(minimo(x,y),z));

 system ("PAUSE");
 return 0;
}

```