

Maurizio Gabbrielli  
Simone Martini

# **Linguaggi di programmazione Principi e paradigmi**

**Seconda edizione**

**McGraw-Hill**

---

Milano • New York • San Francisco • Washington D.C. • Auckland  
Bogotá • Lisboa • London • Madrid • Mexico City • Montreal  
New Delhi • San Juan • Singapore • Sydney • Tokyo • Toronto

Copyright © 2011, 2006 The McGraw-Hill Companies, S.r.l.  
Publishing Group Italia  
Via Ripamonti, 89 – 20139 Milano

**McGraw-Hill**



A Division of the McGraw-Hill Companies

I diritti di traduzione, di riproduzione, di memorizzazione elettronica e di adattamento totale e parziale con qualsiasi mezzo (compresi i microfilm e le copie fotostatiche) sono riservati per tutti i Paesi.

Le fotocopie *per uso personale* del lettore possono essere effettuate nei limiti del 15% di ciascun volume/fascicolo di periodico dietro pagamento alla SIAE del compenso previsto dall'art. 68, commi 4 e 5, della legge 22 aprile 1941 n. 633.

Le riproduzioni effettuate per finalità di carattere professionale, economico o commerciale o comunque *per uso diverso da quello personale* possono essere effettuate a seguito di specifica autorizzazione rilasciata da AIDRO, Corso di Porta Romana n. 108, 20122 Milano, e-mail [segreteria@aidro.org](mailto:segreteria@aidro.org) e sito web [www.aidro.org](http://www.aidro.org)

Nomi e marchi citati nel testo sono generalmente depositati o registrati dalle rispettive case produttrici.

Publisher: Paolo Roncoroni  
Development Editor: Filippo Aroffo  
Produzione: Donatella Giuliani  
Impaginazione: a cura degli Autori  
Grafica di copertina: Editta Gelsomini  
Immagine di copertina: © Julien Eichinger  
Stampa: Rotolito Lombarda, Pioltello (MI)

ISBN: 978-88386-6573-8  
Printed in Italy  
123456789IMPLGV54321

*A Francesca e Antonella,  
che non vorranno mai leggere questo libro,  
ma che hanno contribuito a farcelo scrivere.*

*A Costanza, Maria e Teresa,  
che forse lo leggeranno,  
ma che hanno fatto di tutto per non farcelo scrivere.*

---

<b>Prefazione</b>	<b>xv</b>
<b>Introduzione</b>	<b>xvii</b>
<b>1 Macchine astratte</b>	<b>1</b>
1.1 La nozione di macchina astratta e l'interprete . . . . .	1
1.1.1 Interprete . . . . .	3
1.1.2 Un esempio di macchina astratta: la macchina hardware . . . . .	6
1.2 Implementazione di un linguaggio . . . . .	9
1.2.1 Realizzazione di una macchina astratta . . . . .	10
1.2.2 Implementazione: il caso ideale . . . . .	13
1.2.3 Implementazione: il caso reale e la macchina intermedia . . . . .	18
1.3 Gerarchie di macchine astratte . . . . .	21
1.4 Sommario del capitolo . . . . .	24
1.5 Nota bibliografica . . . . .	25
1.6 Esercizi . . . . .	25
<b>2 Descrivere un linguaggio di programmazione</b>	<b>27</b>
2.1 Livelli di descrizione . . . . .	27
2.2 Grammatica e sintassi . . . . .	28
2.2.1 Grammatiche libere . . . . .	30
2.3 Vincoli sintattici contestuali . . . . .	40
2.4 Compilatori . . . . .	42
2.5 Semantica . . . . .	47
2.6 Pragmatica . . . . .	49
2.7 Implementazione . . . . .	49
2.8 Sommario del capitolo . . . . .	49
2.9 Nota bibliografica . . . . .	50
2.10 Esercizi . . . . .	50
<b>3 Analisi lessicale: linguaggi regolari</b>	<b>51</b>
3.1 Analisi lessicale . . . . .	51
3.1.1 Token . . . . .	51
3.2 Espressioni regolari . . . . .	52
3.2.1 Linguaggi formali e operazioni . . . . .	53
3.2.2 Espressioni regolari . . . . .	53

3.3	Automi finiti . . . . .	56
3.3.1	Automi finiti non deterministici . . . . .	58
3.3.2	Automi finiti deterministici . . . . .	59
3.3.3	DFA e NFA sono equivalenti . . . . .	60
3.4	Da espressioni regolari ad automi finiti . . . . .	63
3.5	Automi finiti e grammatiche . . . . .	65
3.6	Minimizzare un DFA . . . . .	67
3.7	Generatori di analizzatori lessicali . . . . .	71
3.7.1	Lex e i generatori di analizzatori sintattici . . . . .	75
3.8	Dimostrare che un linguaggio non è regolare . . . . .	77
3.9	Sommario del capitolo . . . . .	80
3.10	Nota bibliografica . . . . .	81
3.11	Esercizi . . . . .	81
<b>4</b>	<b>Analisi sintattica: linguaggi liberi</b> . . . . .	<b>83</b>
4.1	Linguaggi, derivazioni e alberi . . . . .	83
4.2	Automi a pila . . . . .	84
4.3	Dimostrare che un linguaggio non è libero . . . . .	88
4.4	Oltre i linguaggi liberi . . . . .	89
4.5	Analizzatori sintattici . . . . .	90
4.6	Manipolazioni delle grammatiche . . . . .	93
4.6.1	Eliminare le $\epsilon$ -produzioni . . . . .	93
4.6.2	Eliminare i simboli inutili . . . . .	94
4.6.3	Eliminare le produzioni unitarie . . . . .	96
4.6.4	Mettere insieme le cose . . . . .	97
4.6.5	Eliminare la ricorsione sinistra . . . . .	98
4.6.6	Fattorizzazione sinistra . . . . .	99
4.7	Parser top-down . . . . .	100
4.7.1	Parser a discesa ricorsiva . . . . .	101
4.7.2	First e Follow . . . . .	102
4.7.3	Grammatiche LL(1) . . . . .	104
4.7.4	Parser LL(1) non ricorsivi . . . . .	106
4.7.5	Grammatiche LL(k) . . . . .	107
4.8	Parser bottom-up . . . . .	108
4.8.1	Prefissi e maniglie . . . . .	111
4.8.2	Item e automa LR(0) . . . . .	112
4.8.3	Tabella e parser LR . . . . .	115
4.8.4	Parser SLR(1) . . . . .	117
4.8.5	Parser LR(1) . . . . .	119
4.8.6	Parser LALR(1) . . . . .	121
4.8.7	Parsing con grammatiche ambigue . . . . .	125
4.9	Linguaggi e grammatiche deterministici . . . . .	128
4.10	Generatori di analizzatori sintattici . . . . .	129
4.10.1	Yacc e l'ambiguità . . . . .	132
4.11	Sommario del capitolo . . . . .	132
4.12	Nota bibliografica . . . . .	133

4.13	Esercizi . . . . .	133
<b>5</b>	<b>Fondamenti</b> . . . . .	<b>135</b>
5.1	Il problema della fermata . . . . .	135
5.2	Espressività dei linguaggi di programmazione . . . . .	137
5.2.1	Formalismi per la calcolabilità . . . . .	138
5.3	Esistono più funzioni che algoritmi . . . . .	140
5.4	Sommario del capitolo . . . . .	142
5.5	Nota bibliografica . . . . .	142
5.6	Esercizi . . . . .	142
<b>6</b>	<b>I nomi e l'ambiente</b> . . . . .	<b>143</b>
6.1	Nomi e oggetti denotabili . . . . .	143
6.1.1	Oggetti denotabili . . . . .	145
6.2	Ambiente e blocchi . . . . .	146
6.2.1	I blocchi . . . . .	148
6.2.2	Tipi di ambiente . . . . .	148
6.2.3	Operazioni sull'ambiente . . . . .	151
6.3	Regole di scope . . . . .	154
6.3.1	Scope statico . . . . .	155
6.3.2	Scope dinamico . . . . .	157
6.4	Sommario del capitolo . . . . .	160
6.5	Nota bibliografica . . . . .	161
6.6	Esercizi . . . . .	161
<b>7</b>	<b>La gestione della memoria</b> . . . . .	<b>165</b>
7.1	Tecniche di gestione della memoria . . . . .	165
7.2	Gestione statica della memoria . . . . .	167
7.3	Gestione dinamica mediante pila . . . . .	168
7.3.1	Record di attivazione per i blocchi in-line . . . . .	170
7.3.2	Record di attivazione per le procedure . . . . .	172
7.3.3	Gestione della pila . . . . .	174
7.4	Gestione dinamica mediante heap . . . . .	176
7.4.1	Blocchi di dimensione fissa . . . . .	177
7.4.2	Blocchi di dimensione variabile . . . . .	178
7.5	Implementazione delle regole di scope . . . . .	181
7.5.1	Scope statico: la catena statica . . . . .	181
7.5.2	Scope statico: il display . . . . .	186
7.5.3	Scope dinamico: lista di associazioni e CRT . . . . .	188
7.6	Sommario del capitolo . . . . .	193
7.7	Nota bibliografica . . . . .	194
7.8	Esercizi . . . . .	194

<b>8 Strutturare il controllo</b>	<b>197</b>
8.1 Le espressioni . . . . .	197
8.1.1 Sintassi delle espressioni . . . . .	198
8.1.2 Semantica delle espressioni . . . . .	201
8.1.3 Valutazione delle espressioni . . . . .	203
8.2 La nozione di comando . . . . .	208
8.2.1 La variabile . . . . .	209
8.2.2 L'assegnamento . . . . .	210
8.3 Comandi per il controllo di sequenza . . . . .	213
8.3.1 Comandi per il controllo di sequenza esplicito . . . . .	215
8.3.2 Comandi condizionali . . . . .	218
8.3.3 Comandi iterativi . . . . .	222
8.4 Programmazione strutturata . . . . .	229
8.5 La ricorsione . . . . .	231
8.5.1 La ricorsione in coda . . . . .	233
8.5.2 Ricorsione o iterazione? . . . . .	239
8.6 Sommario del capitolo . . . . .	239
8.7 Nota bibliografica . . . . .	240
8.8 Esercizi . . . . .	240
<b>9 Astrarre sul controllo</b>	<b>243</b>
9.1 Sottoprogrammi . . . . .	244
9.1.1 Astrazione funzionale . . . . .	246
9.1.2 Passaggio dei parametri . . . . .	247
9.2 Funzioni di ordine superiore . . . . .	258
9.2.1 Funzioni come parametro . . . . .	258
9.2.2 Funzioni come risultato . . . . .	264
9.3 Eccezioni . . . . .	266
9.3.1 Implementare le eccezioni . . . . .	271
9.4 Sommario del capitolo . . . . .	274
9.5 Nota bibliografica . . . . .	275
9.6 Esercizi . . . . .	275
<b>10 Strutturare i dati</b>	<b>279</b>
10.1 Tipi di dato . . . . .	279
10.1.1 Tipi come supporto all'organizzazione concettuale . . . . .	280
10.1.2 Tipi per la correttezza . . . . .	281
10.1.3 Tipi e implementazione . . . . .	282
10.2 Sistemi di tipi . . . . .	283
10.2.1 Controlli statici e dinamici . . . . .	284
10.3 Tipi scalari . . . . .	286
10.3.1 Booleani . . . . .	287
10.3.2 Caratteri . . . . .	287
10.3.3 Interi . . . . .	287
10.3.4 Reali . . . . .	288
10.3.5 Virgola fissa . . . . .	288

10.3.6 Complessi . . . . .	288
10.3.7 Void . . . . .	289
10.3.8 Enumerazioni . . . . .	289
10.3.9 Intervalli . . . . .	290
10.3.10 Tipi ordinali . . . . .	291
10.4 Tipi composti . . . . .	291
10.4.1 Record . . . . .	291
10.4.2 Record varianti e unioni . . . . .	293
10.4.3 Array . . . . .	298
10.4.4 Insiemi . . . . .	304
10.4.5 Puntatori . . . . .	304
10.4.6 Tipi ricorsivi . . . . .	310
10.4.7 Funzioni . . . . .	312
10.5 Equivalenza . . . . .	313
10.5.1 Equivalenza per nome . . . . .	313
10.5.2 Equivalenza strutturale . . . . .	314
10.6 Compatibilità e conversione . . . . .	316
10.7 Polimorfismo . . . . .	320
10.7.1 Overloading . . . . .	321
10.7.2 Polimorfismo universale parametrico . . . . .	321
10.7.3 Polimorfismo universale di sottotipo . . . . .	324
10.7.4 Cenni sull'implementazione . . . . .	326
10.8 Controllo e inferenza di tipo . . . . .	327
10.9 Sicurezza: un bilancio . . . . .	328
10.10 Evitare i dangling reference . . . . .	330
10.10.1 Tombstone . . . . .	330
10.10.2 Lucchetti e chiavi . . . . .	331
10.11 Garbage collection . . . . .	332
10.11.1 Contatori dei riferimenti . . . . .	334
10.11.2 Mark and sweep . . . . .	336
10.11.3 Intermezzo: rovesciare i puntatori . . . . .	337
10.11.4 Mark and compact . . . . .	338
10.11.5 Copia . . . . .	339
10.12 Sommario del capitolo . . . . .	342
10.13 Nota bibliografica . . . . .	342
10.14 Esercizi . . . . .	343
<b>11 Astrarre sui dati</b>	<b>347</b>
11.1 Tipi di dato astratti . . . . .	348
11.2 Nascondere l'informazione . . . . .	351
11.2.1 Indipendenza dalla rappresentazione . . . . .	354
11.3 Moduli . . . . .	354
11.4 Sommario del capitolo . . . . .	356
11.5 Nota bibliografica . . . . .	358
11.6 Esercizi . . . . .	358

<b>12 Il paradigma orientato agli oggetti</b>	<b>359</b>
12.1 Limiti dei tipi di dato astratti . . . . .	359
12.1.1 Un primo bilancio . . . . .	363
12.2 Concetti fondamentali . . . . .	363
12.2.1 Oggetti . . . . .	364
12.2.2 Classi . . . . .	366
12.2.3 Incapsulamento . . . . .	368
12.2.4 Sottotipi . . . . .	370
12.2.5 Ereditarietà . . . . .	375
12.2.6 Selezione dinamica dei metodi . . . . .	380
12.3 Aspetti implementativi . . . . .	384
12.3.1 Ereditarietà singola . . . . .	386
12.3.2 Il problema della classe base fragile . . . . .	388
12.3.3 Selezione dinamica dei metodi nella JVM . . . . .	389
12.3.4 Ereditarietà multipla . . . . .	392
12.4 Polimorfismo e generici . . . . .	398
12.4.1 Polimorfismo di sottotipo . . . . .	398
12.4.2 Generici in Java . . . . .	401
12.4.3 Implementazione dei generici in Java . . . . .	405
12.4.4 Generici, array e gerarchia di sottotipo . . . . .	406
12.4.5 Overriding covarianti e controvarianti . . . . .	408
12.5 Sommario del capitolo . . . . .	411
12.6 Nota bibliografica . . . . .	412
12.7 Esercizi . . . . .	412
<b>13 Il paradigma funzionale</b>	<b>415</b>
13.1 Computazioni senza stato . . . . .	415
13.1.1 Espressioni e funzioni . . . . .	417
13.1.2 Computazione come riduzione . . . . .	419
13.1.3 Gli ingredienti fondamentali . . . . .	419
13.2 Valutazione . . . . .	421
13.2.1 Valori . . . . .	421
13.2.2 Sostituzione senza cattura . . . . .	422
13.2.3 Strategie di valutazione . . . . .	423
13.2.4 Confronto tra strategie . . . . .	425
13.3 Programmare in un linguaggio funzionale . . . . .	427
13.3.1 Ambiente locale . . . . .	427
13.3.2 Interattività . . . . .	428
13.3.3 Tipi . . . . .	428
13.3.4 Pattern matching . . . . .	429
13.3.5 Oggetti infiniti . . . . .	431
13.3.6 Aspetti imperativi . . . . .	432
13.4 Implementazione: la macchina SECD . . . . .	434
13.5 Il paradigma funzionale a confronto . . . . .	437
13.6 Fondamenti: $\lambda$ -calcolo . . . . .	440
13.7 Sommario del capitolo . . . . .	446

13.8 Nota bibliografica . . . . .	447
13.9 Esercizi . . . . .	447
<b>14 Il paradigma logico</b>	<b>449</b>
14.1 Deduzione come computazione . . . . .	449
14.1.1 Un esempio . . . . .	450
14.2 Sintassi . . . . .	454
14.2.1 Il linguaggio della logica del prim'ordine . . . . .	454
14.2.2 I programmi logici . . . . .	456
14.3 Teoria dell'unificazione . . . . .	457
14.3.1 La variabile logica . . . . .	457
14.3.2 Sostituzione . . . . .	459
14.3.3 L'unificatore più generale . . . . .	462
14.3.4 Un algoritmo di unificazione . . . . .	464
14.4 Il modello computazionale . . . . .	467
14.4.1 L'universo di Herbrand . . . . .	467
14.4.2 Interpretazione dichiarativa e procedurale . . . . .	468
14.4.3 La chiamata di procedura . . . . .	469
14.4.4 Controllo: il non determinismo . . . . .	472
14.4.5 Alcuni esempi . . . . .	475
14.5 Estensioni . . . . .	478
14.5.1 PROLOG . . . . .	478
14.5.2 Programmazione logica e basi di dati . . . . .	483
14.5.3 Programmazione logica con vincoli . . . . .	484
14.6 Pregi e difetti del paradigma logico . . . . .	486
14.7 Sommario del capitolo . . . . .	488
14.8 Nota bibliografica . . . . .	489
14.9 Esercizi . . . . .	490
<b>15 La programmazione concorrente</b>	<b>493</b>
15.1 Thread e processi . . . . .	493
15.2 Una breve panoramica storica . . . . .	494
15.3 Tipi di programmazione concorrente . . . . .	496
15.3.1 Meccanismi di comunicazione . . . . .	499
15.3.2 Meccanismi di sincronizzazione . . . . .	501
15.4 Memoria condivisa . . . . .	503
15.4.1 Attesa attiva . . . . .	503
15.4.2 Sincronizzazione basata sullo scheduler . . . . .	506
15.5 Scambio di messaggi . . . . .	513
15.5.1 Meccanismi di naming . . . . .	513
15.5.2 Comunicazione asincrona . . . . .	516
15.5.3 Comunicazione sincrona . . . . .	518
15.5.4 Chiamata di procedura remota e rendez-vous . . . . .	519
15.6 Non determinismo e composizione parallela . . . . .	522
15.6.1 La composizione parallela . . . . .	525
15.7 La concorrenza in Java . . . . .	525

15.7.1 Creazione dei thread . . . . .	526
15.7.2 Scheduling e terminazione dei thread . . . . .	527
15.7.3 Sincronizzazione e comunicazione fra thread . . . . .	529
15.8 Sommario del capitolo . . . . .	532
15.9 Nota bibliografica . . . . .	533
15.10 Esercizi . . . . .	533
<b>16 Una breve panoramica storica</b>	<b>535</b>
16.1 Gli inizi . . . . .	535
16.2 Fattori evolutivi dei linguaggi . . . . .	538
16.3 Anni '50 e '60 . . . . .	539
16.4 Anni '70 . . . . .	544
16.5 Anni '80 . . . . .	549
16.6 Anni '90 . . . . .	553
16.7 Sommario del capitolo . . . . .	556
16.8 Nota bibliografica . . . . .	557
<b>Bibliografia</b>	<b>559</b>
<b>Indice analitico</b>	<b>565</b>

## Prefazione

Ho accettato con grande piacere l'invito rivoltomi a scrivere queste poche righe di prefazione per almeno due ragioni. La prima è che la richiesta mi viene da due colleghi che ho sempre tenuto in grandissima considerazione, fin dal tempo in cui li ho potuti conoscere ed apprezzare come studenti prima e giovani ricercatori poi.

La seconda ragione è che il testo è molto vicino al libro che io avrei sempre voluto scrivere e, per ragioni varie, non ho mai scritto. In particolare, l'approccio seguito dal libro è quello che io stesso ho seguito nell'organizzazione di vari corsi di linguaggi di programmazione che ho insegnato, in diverse fasi e sotto diverse etichette, per quasi trent'anni.

L'approccio, schematizzato in due parole, è quello di introdurre i concetti generali (sia dei meccanismi linguistici sia delle corrispondenti strutture di implementazione) in modo indipendente da linguaggi specifici e solo in un secondo tempo collocare nello schema i "linguaggi veri". Questo è l'unico approccio che permette di mettere in evidenza le similitudini tra linguaggi (ed anche tra paradigmi) apparentemente molto diversi. Allo stesso tempo si rende così più facile il compito di imparare linguaggi diversi. Nella mia esperienza di docente, gli ex-studenti si ricordano i principi appresi nel corso anche dopo molti anni e continuano ad apprezzarne l'approccio, che ha loro permesso di adattarsi alle evoluzioni delle tecnologie senza grandi difficoltà.

Il testo di Gabbielli e Martini ha come riferimento prevalente un corso della laurea (triennale) in Informatica. Per questa ragione non ha prerequisiti complessi e affronta l'argomento trovando un perfetto equilibrio tra rigore e semplicità. Particolarmente apprezzabile e riuscito è lo sforzo di mettere in luce il collegamento con alcuni "pezzi di teoria" importanti (come i linguaggi formali, la calcolabilità, la semantica), che il libro giustamente richiama, anche perché in molti corsi di laurea questi argomenti non vengono più trattati.

Giorgio Levi

## Introduzione

*Facilius per partes in cognitionem totius adducimur.*  
(Seneca, Epist., 89, 1)

L'apprendimento di un linguaggio di programmazione costituisce per molti studenti il rito d'iniziazione all'informatica. Si tratta di un passaggio importante, ma che presto si rivela insufficiente: tra gli attrezzi del mestiere ci sono molti linguaggi ed una competenza importante del bravo informatico è quella di saper passare da un linguaggio all'altro (e di apprenderne di nuovi) con naturalezza e velocità.

Questa competenza non la si acquisisce soltanto imparando *ex novo* molti linguaggi diversi. Come le lingue naturali, anche i linguaggi di programmazione hanno tra loro somiglianze, analogie, fenomeni di importazione dall'uno all'altro, genealogie che ne influenzano le caratteristiche. Se è impossibile imparare bene decine di linguaggi di programmazione, è possibile invece conoscere a fondo i meccanismi che ispirano e guidano il progetto e l'implementazione di centinaia di linguaggi diversi. Questa conoscenza delle "parti" facilita la comprensione del "tutto" costituito da un nuovo linguaggio, fornendo una competenza metodologica fondamentale nella vita professionale dell'informatico, in quanto consente di anticipare l'innovazione e di sopravvivere all'obsolescenza delle tecnologie.

È per questi motivi che un corso sugli aspetti generali dei linguaggi di programmazione è, in tutto il mondo, un passaggio chiave della formazione avanzata (universitaria o professionale) di un informatico. Relativamente ai linguaggi di programmazione, le competenze fondamentali che un informatico deve possedere sono almeno di quattro tipi:

- gli aspetti propriamente linguistici;
- come i costrutti linguistici possono essere implementati ed il relativo costo;
- gli aspetti architetturali che influenzano l'implementazione;
- le tecniche di traduzione (compilazione).

È raro che un singolo corso possa affrontare tutti e quattro questi aspetti. In particolare, la descrizione degli aspetti architetturali costituisce un argomento sufficientemente complesso ed elaborato da meritare una trattazione separata. Gli altri aspetti sono il contenuto primario di un corso generale sui linguaggi di programmazione e costituiscono il contenuto principale di questo testo.

La letteratura anglosassone, a differenza di quella in lingua italiana, è ricca di testi che affrontano questi argomenti, alcuni carichi di storia e sui quali si sono formate generazioni di studenti. Tutti questi testi, tuttavia, hanno in mente un

lettore avanzato che conosca già diversi linguaggi di programmazione, che abbia una competenza non superficiale dei meccanismi di base fondamentali, che non si spaventi davanti a frammenti di codice espressi in linguaggi a lui ignoti (perché riesce a comprenderli per analogia o per differenza da quelli che già conosce). Si tratta di testi, dunque, che potremmo chiamare di “linguaggi comparati”: estesi, approfonditi, stimolanti. Ma *tropo* estesi e approfonditi (in una parola: difficili) per lo studente che inizia il suo cammino con un solo (o al massimo due) linguaggi di programmazione e che deve ancora approfondire i concetti di base.

Questo testo intende colmare questa lacuna. Gli esperti vedranno che l’indice degli argomenti ripercorre in larga misura i temi classici. Ma questi stessi temi sono trattati in modo elementare, cercando di assumere come prerequisiti solo il minimo indispensabile e sforzandosi di non fare un catalogo delle opzioni possibili nei diversi linguaggi di programmazione esistenti. Il lettore di riferimento è quello che conosce (bene) un linguaggio (per esempio Pascal, C, C++, o Java); meglio se ha avuto anche una qualche esposizione ad un altro linguaggio o ad un altro paradigma. Si sono evitati riferimenti consistenti a linguaggi ormai desueti e gli esempi di codice sono solo raramente espressi in uno specifico linguaggio di programmazione: il testo usa con libertà una sorta di pseudolinguaggio (ispirato nella sua sintassi concreta a C e Java), cercando di descrivere così gli aspetti più rilevanti che uniscono i diversi linguaggi.

Di tanto in tanto un “riquadro” a capo pagina presenta un approfondimento, o il richiamo di una nozione di base, o qualche dettaglio specifico dei linguaggi più comuni (C, C++, Java 5.0; ML e LISP per i linguaggi funzionali; PROLOG per i linguaggi logici). Il materiale dei riquadri può quasi sempre essere tranquillamente omesso in una prima lettura.

Tutti i capitoli presentano una breve serie di esercizi, intesi come banco di prova per la comprensione del materiale. Non vi sono esercizi davvero difficili o che richiedano più di una decina di minuti per esser risolti.

Il Capitolo 5 (*Fondamenti*) affronta temi che solitamente non sono trattati in un testo di linguaggi di programmazione. È tuttavia naturale, discutendo di semantica statica e confrontando tra loro linguaggi, chiedersi quali siano i limiti dell’analisi statica dei programmi e se quello che si può fare in un linguaggio si possa fare anche in un altro. Invece che rimandare ad altri testi, vista la rilevanza sia culturale sia pragmatica di queste questioni, abbiamo deciso di rispondere direttamente a queste domande. In modo informale, ma rigoroso, nel contesto di poche pagine viene presentata l’individuabilità del problema della fermata e l’equivalenza dei linguaggi di programmazione quanto alle funzioni calcolabili. Questo permette di esporre lo studente, che non sempre ha nel suo curriculum un corso completo sui “fondamenti”, ai risultati principali relativi alle limitazioni dei procedimenti di calcolo, che riteniamo fondamentali per la sua formazione.

Insieme ai principi, il testo introduce anche ai principali *paradigmi di programmazione*: quello orientato agli oggetti, quello funzionale, quello logico, quello concorrente. La necessità di realizzare un testo introduttivo e, insieme, di mantenerne l’estensione in un numero di pagine ragionevole, spiega l’esclusione di temi importanti, quali per esempio i linguaggi di scripting.

**Uso del testo** Il testo è in primo luogo un manuale universitario, ma è anche adatto allo studio personale del professionista che voglia approfondire la propria conoscenza dei meccanismi che stanno dietro ai linguaggi che utilizza. La scelta dei temi e lo stile di presentazione sono stati ampiamente influenzati dall’esperienza di insegnamento di questi contenuti presso il corso di laurea in Informatica della Facoltà di Scienze Matematiche, Fisiche e Naturali dell’Alma Mater Studiorum – Università di Bologna.

Nella nostra esperienza, il testo può coprire due moduli di sei crediti, posti al secondo o terzo anno della laurea triennale. Un modulo può essere dedicato agli aspetti relativi alle macchine astratte e alle tecniche di compilazione (Capitoli da 1 a 5); un altro può coprire gli aspetti fondamentali (diciamo i 4/5) dei Capitoli da 6 a 12 e accennare ad uno dei restanti paradigmi. Al crescere della maturità degli studenti, aumenta ovviamente la quantità di materiale che può essere presentato.

**La seconda edizione** La seconda edizione, oltre ad una revisione generale del materiale e alla correzione dei refusi, contiene tre capitoli del tutto nuovi. In primo luogo la semplice trattazione della sintassi del Capitolo 2 è stata estesa con la presentazione, sintetica ma non superficiale, dei linguaggi regolari e degli analizzatori lessicali (Capitolo 3) e, quindi, dei linguaggi liberi e degli analizzatori sintattici (Capitolo 4). Entrambi i capitoli presentano i principali risultati teorici e le loro applicazioni alla tecnologia dei compilatori. In questo modo, il testo può essere usato come unico manuale in un corso che voglia trattare in modo non superficiale anche dei traduttori. I due capitoli hanno un carattere un po’ più matematico degli altri, ma non richiedono nessun prerequisito che vada oltre il principio di induzione.

Il terzo capitolo aggiunto a questa seconda edizione presenta la programmazione concorrente. Anche se una trattazione esauriente di questa tematica richiederebbe (almeno) un volume a sé, l’assenza di qualsiasi riferimento alla concorrenza ci è sembrata una lacuna troppo vistosa in un testo sui linguaggi di programmazione, visto che una parte significativa del software oggi usa programmi concorrenti, dal livello del sistema operativo sino a quello dei servizi sul web. Il capitolo illustra le principali problematiche che si presentano passando dai programmi sequenziali a quelli concorrenti, insieme alle relative soluzioni. Abbiamo cercato di offrire un panorama adeguato delle principali tecniche e dei costrutti linguistici per realizzare meccanismi di interazione, sincronizzazione e comunicazione fra programmi o processi concorrenti. Seguendo il principio informatore di tutto il testo non abbiamo fatto riferimento ad un linguaggio specifico, anche se abbiamo cercato di concretizzare alcune nozioni esaminando il caso di Java.

Per rendere il testo più agile e adattabile alle diverse esigenze, alcuni approfondimenti non sono inclusi nel testo, ma sono disponibili sul sito

[www.ateneonline.it/gabbrielli](http://www.ateneonline.it/gabbrielli).

La loro presenza è indicata con l’icona



**Ringraziamenti** La nostra gratitudine a Giorgio Levi va ben al di là del fatto che ha avuto la bontà di scrivere la Prefazione. Entrambi dobbiamo a lui le nostre prime conoscenze dei meccanismi che sottostanno ai linguaggi di programmazione: il suo insegnamento ritorna in questo libro in modo tutt'altro che marginale.

Ugo Dal Lago ha composto parte delle figure usando METAPOST. Andrea Aquino, Sara Bergonzoni, Matteo Brucato, Ferdinanda Camporesi, Cinzia Di Giusto, Jacopo Mauro, Wilmer Ricciotti, Francesco Spegni e Paolo Tacchella hanno letto e commentato le bozze di alcuni capitoli. Un ringraziamento sincero ai tanti lettori che hanno segnalato refusi ed errori nella prima edizione.

## Macchine astratte

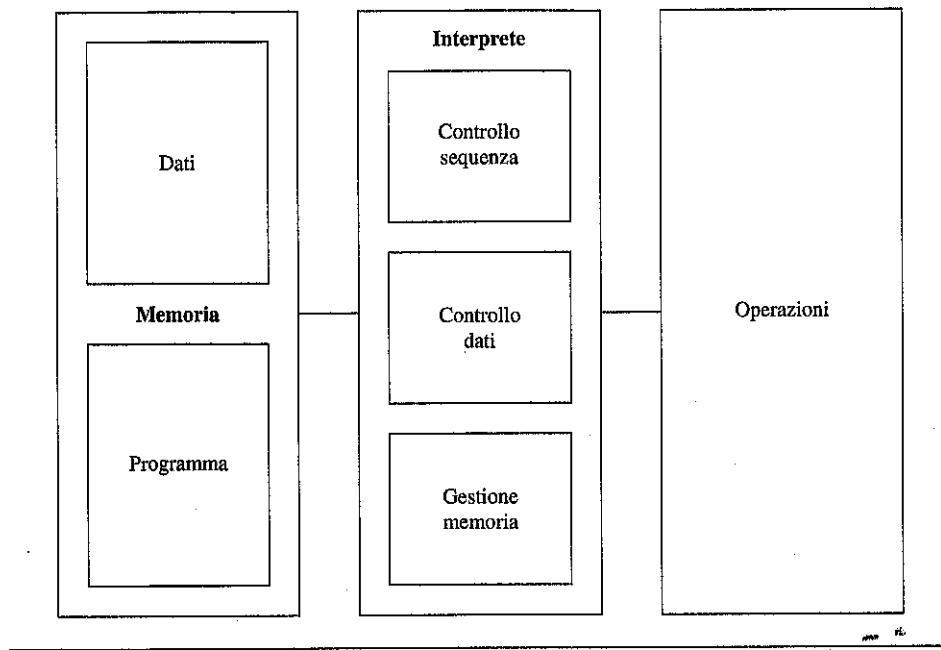
I meccanismi d'astrazione giocano un ruolo cruciale nell'informatica in quanto, isolando gli aspetti rilevanti in un particolare contesto, permettono di dominare la complessità insita nella maggior parte dei sistemi di calcolo. Nell'ambito dei linguaggi di programmazione tali meccanismi sono fondamentali sia dal punto di vista teorico, dato che numerosi concetti importanti possono essere formalizzati opportunamente in termini d'astrazione, sia in senso pratico, visto che i linguaggi di programmazione odierni usano diffusamente costrutti per l'astrazione.

Una delle nozioni più generali che coinvolgono l'astrazione è quella di *macchina astratta*. In questo capitolo vedremo come tale nozione sia intimamente collegata a quella di linguaggio di programmazione e come essa permetta di descrivere che cosa sia l'implementazione di un linguaggio, senza per questo doverci addentrare nei dettagli specifici di una particolare implementazione. Per far questo descriveremo in termini generali l'*interprete* ed il *compilatore* di un linguaggio. Vedremo infine come le macchine astratte possono essere strutturate in gerarchie per descrivere e realizzare sistemi software complessi.

### 1.1 La nozione di macchina astratta e l'interprete

Il termine "macchina" in questo contesto si riferisce evidentemente alla macchina calcolatrice. Come sappiamo, un calcolatore (elettronico, digitale) è una macchina fisica che permette di eseguire degli algoritmi, opportunamente formalizzati per poter essere "comprendibili" all'esecutore. Intuitivamente una macchina astratta non è altro che un'astrazione del concetto di calcolatore fisico.

Per poter essere effettivamente eseguiti gli algoritmi devono essere opportunamente formalizzati in termini dei costrutti di un linguaggio di programmazione. In altri termini, gli algoritmi che vogliamo eseguire devono essere rappresentati mediante le istruzioni di un opportuno linguaggio di programmazione  $\mathcal{L}$ , linguaggio che sarà definito formalmente da una specifica sintassi e da una precisa semantica (si veda il Capitolo 2). Per il momento non ci serve specificare ulteriormente la natura di  $\mathcal{L}$ : ci basta sapere che la sintassi di  $\mathcal{L}$  permette di utilizzare un certo insieme finito di costrutti, detti istruzioni, che permettono di comporre i programmi. Un *programma* di  $\mathcal{L}$  (o programma scritto in  $\mathcal{L}$ ) dunque non è altro che un



**Figura 1.1** La struttura di una macchina astratta.

insieme finito di istruzioni di  $\mathcal{L}$ . Con queste premesse, vediamo una definizione centrale in questo capitolo.

**Definizione 1.1 (Macchina Astratta)** *Supponiamo che sia dato un linguaggio di programmazione  $\mathcal{L}$ . Definiamo una macchina astratta per  $\mathcal{L}$ , e la indichiamo con  $\mathcal{M}_{\mathcal{L}}$ , un qualsiasi insieme di strutture dati e di algoritmi che permettano di memorizzare ed eseguire programmi scritti in  $\mathcal{L}$ .*

Quando non c'interessa specificare il linguaggio  $\mathcal{L}$ , parleremo semplicemente di macchina astratta  $\mathcal{M}$ , omettendo l'indice. Vedremo fra poco alcuni esempi di macchina astratta e come questa si possa effettivamente realizzare; per il momento soffermiamoci sulla sua struttura. Come illustrato in Figura 1.1, una generica macchina astratta  $\mathcal{M}_{\mathcal{L}}$  è composta da una *memoria* e da un *interprete*. La memoria serve per immagazzinare dati e programmi mentre l'interprete è il componente che esegue le istruzioni contenute nei programmi, come vediamo meglio nel prossimo paragrafo.

### 1.1.1 Interpretazione

Evidentemente l'interprete dovrà compiere delle operazioni specifiche che dipendono dal particolare linguaggio  $\mathcal{L}$  che deve essere interpretato. Tuttavia, pur nella diversità dei vari linguaggi, si possono riconoscere delle tipologie di operazioni ed anche una “modalità di esecuzione” comuni a tutti gli interpreti. Per quanto riguarda il tipo di operazioni che l'interprete esegue e le relative strutture dati individuiamo le seguenti categorie:

1. operazioni per l'elaborazione dei dati primitivi;
2. operazioni e strutture dati per il controllo della sequenza di esecuzione delle operazioni;
3. operazioni e strutture dati per il controllo del trasferimento dei dati;
4. operazioni e strutture dati per la gestione della memoria.

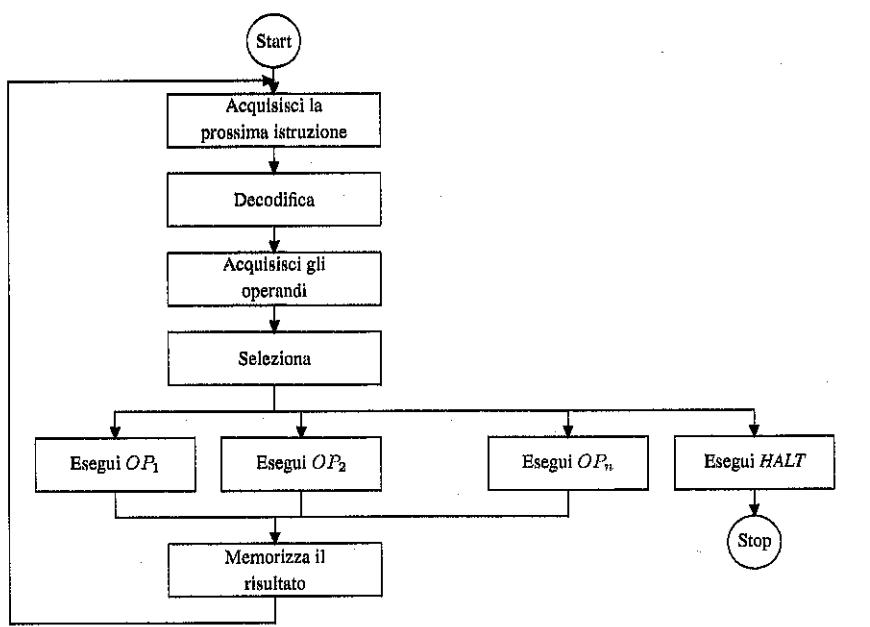
Vediamo nel dettaglio questi quattro punti.

1. La necessità di operazioni quali quelle di cui al punto 1 è evidente: trattandosi di una macchina, anche se astratta, volta all'esecuzione di algoritmi, dovremo avere delle operazioni che permettano di manipolare dati primitivi, dati cioè che siano rappresentabili in modo diretto nella macchina. Ad esempio, facendo riferimento alla macchina astratta fisica, ma anche alle macchine astratte per molti linguaggi di programmazione, i numeri (interi o reali) sono quasi sempre dati primitivi e per essi sono presenti direttamente nella struttura della macchina varie operazioni aritmetiche (somma, moltiplicazione ecc.). Tali operazioni aritmetiche sono dunque operazioni primitive della macchina astratta<sup>1</sup>.

2. Le operazioni e le strutture dati per il cosiddetto “controllo di sequenza” servono per gestire il flusso di esecuzione delle istruzioni presenti in un programma. La normale esecuzione sequenziale di un programma può dover esser modificata in corrispondenza del verificarsi di determinate condizioni. L'interprete dispone quindi di strutture dati opportune (ad esempio per memorizzare l'indirizzo della prossima istruzione da eseguire) che sono manipolate con operazioni particolari, diverse dalle operazioni che manipolano i dati (ad esempio, operazioni per aggiornare l'indirizzo della prossima istruzione da eseguire).

3. Le operazioni per il controllo del trasferimento dati servono per controllare come gli operandi e, in generale, i dati, devono essere trasferiti dalla memoria all'interprete e viceversa. Tali operazioni riguardano le varie modalità di indirizzamento della memoria e l'ordine con cui gli operandi devono essere recuperati dalla memoria. In alcuni casi, per gestire opportunamente il trasferimento dati, possono essere necessarie delle strutture dati ausiliarie: questo è il caso ad esempio della pila in alcuni tipi di macchine (sia hardware sia software).

<sup>1</sup>Si noti comunque che esistono linguaggi di programmazione per i quali i valori numerici e le relative operazioni non sono primitivi. Questo è il caso ad esempio di alcuni linguaggi dichiarativi.



**Figura 1.2** Il ciclo di esecuzione di un generico interprete.

4. La gestione della memoria, infine, riguarda tutte le operazioni relative all’allocazione di memoria per i dati e per i programmi. Nel caso di macchine astratte vicine alla macchina hardware la gestione della memoria è abbastanza semplice. Nel caso limite di una macchina fisica a registri senza multiprogrammazione, un programma ed i relativi dati potrebbero essere allocati in una zona di memoria all’inizio dell’esecuzione e rimanervi fino alla fine, senza praticamente alcuna necessità di gestione della memoria. Le macchine astratte per i comuni linguaggi di programmazione invece, come vedremo, prevedono una gestione della memoria più complicata. Infatti alcuni costrutti di questi linguaggi causano, direttamente o indirettamente, operazioni di allocazione e deallocazione di memoria che, per poter essere gestite correttamente, richiedono opportune strutture dati (ad esempio pile) e operazioni dinamiche (cioè eseguite a tempo di esecuzione).

Il ciclo di esecuzione dell’interprete, sostanzialmente comune a tutti gli interpreti, è descritto in Figura 1.2 ed è costituito dalle seguenti fasi: inizialmente avviene l’acquisizione, dalla memoria, della prossima istruzione da eseguire. L’istruzione è quindi decodificata per individuare qual’è l’operazione richiesta dall’istruzione e quali sono gli operandi. Gli operandi, nel numero richiesto e secondo le modalità individuate nella fase precedente, sono prelevati dalla memoria dopo di che viene eseguita l’operazione richiesta (che dovrà essere una delle operazioni primitive della macchina in questione). Una volta completata l’esecuzione dell’operazione l’eventuale risultato viene memorizzato; quindi, a meno

### Linguaggi “di basso livello” e “di alto livello”

È utile una nota terminologica che sarà ripresa in prospettiva storica nel Capitolo 16. Nell’ambito dei linguaggi di programmazione si usano spesso i termini “basso livello” e “alto livello” per riferirsi, rispettivamente, alla vicinanza alla macchina hardware e alla vicinanza all’utente umano.

Chiameremo dunque linguaggi di programmazione *di basso livello* i linguaggi di macchine astratte molto vicine alla macchina hardware o che coincidono con questa. Questi linguaggi sono stati i primi ad essere impiegati, già alla fine degli anni ’40, per programmare i calcolatori, ma sono risultati estremamente scomodi da usarsi. Difatti le istruzioni di questi linguaggi, dovendo tener conto di caratteristiche fisiche della macchina, fanno sì che nella scrittura di un programma, ovvero nella codifica di un algoritmo, si debbano considerare dei dettagli assolutamente irrilevanti dal punto di vista algoritmico. Va ricordato che spesso quando si parla genericamente di “linguaggio macchina” s’intende il linguaggio (di basso livello) di una macchina hardware. Un particolare linguaggio di basso livello è il linguaggio *assembly*, che è una versione simbolica del linguaggio di una macchina hardware (cioè che usa simboli quali ADD, MUL ecc. invece dei relativi codici binari assoluti). I programmi in linguaggio assembly sono tradotti in codice macchina da un opportuno programma detto *assemblatore*.

I linguaggi di programmazione *di alto livello* sono invece quelli che, mediante opportuni meccanismi di astrazione, permettono di usare costrutti che prescindono dalle caratteristiche fisiche del calcolatore. I linguaggi di alto livello sono dunque adatti ad esprimere algoritmi in modo relativamente facile per l’utente umano. Ovviamente, anche i costrutti di un linguaggio di alto livello dovranno corrispondere a istruzioni della macchina fisica, perché sia possibile eseguire i programmi.

che l’istruzione appena eseguita non sia un’istruzione di arresto, si passa all’esecuzione dell’istruzione successiva, ripetendo dall’inizio il ciclo. Visto l’interprete, possiamo definire come segue il linguaggio interpretato.

**Definizione 1.2 (Linguaggio macchina)** *Data una macchina astratta  $\mathcal{M}_L$ , il linguaggio  $\mathcal{L}$  “compreso” dall’interprete di  $\mathcal{M}_L$  è detto linguaggio macchina di  $\mathcal{M}_L$ .*

I programmi scritti nel linguaggio macchina di  $\mathcal{M}_L$  saranno memorizzati nelle strutture di memoria della macchina in modo tale da essere distinti dagli altri dati primitivi sui quali opera l’interprete (si noti che dal punto di vista dell’interprete anche i programmi costituiscono una particolare categoria di dati). Dato che, normalmente, la rappresentazione interna dei programmi usata dalla macchina  $\mathcal{M}_L$ , è diversa dalla loro rappresentazione esterna, a rigore dovremmo parlare di due linguaggi diversi. Tuttavia, per non complicare la notazione, per il momento non consideriamo tale differenza e quindi parliamo di un unico linguaggio macchina  $\mathcal{L}$  per  $\mathcal{M}_L$ .

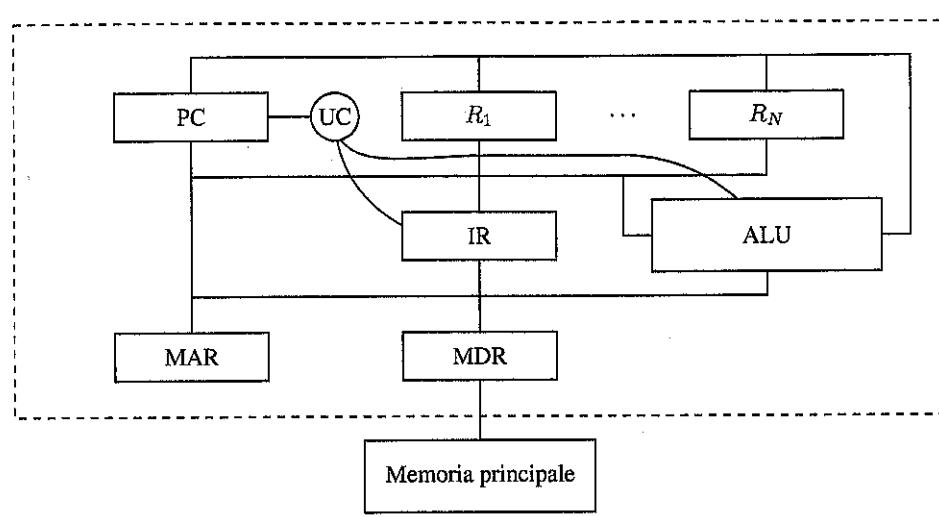


Figura 1.3 La struttura di un calcolatore convenzionale.

### 1.1.2 Un esempio di macchina astratta: la macchina hardware

Il concetto di macchina astratta, come appare evidente da quanto detto sin qui, si può usare per descrivere una varietà di sistemi diversi, che vanno dalla macchina hardware al World Wide Web.

Come primo esempio di macchina astratta vediamo il caso, concreto e tangibile, di una macchina hardware convenzionale quale quella in Figura 1.3 realizzata fisicamente da circuiti logici e dispositivi elettronici. Chiameremo  $MH_{\mathcal{L}\mathcal{H}}$  tale macchina e  $\mathcal{L}\mathcal{H}$  il suo linguaggio macchina.

Analizzando quelli che abbiamo detto essere i componenti di una qualsiasi macchina astratta, in questo specifico caso possiamo identificare le seguenti parti.

**Memoria** La componente di memorizzazione della macchina fisica è costituita dai vari livelli di memoria di un calcolatore fisico: memoria secondaria realizzata mediante dispositivi magnetici od ottici; memoria principale organizzata in una sequenza lineare di celle, o parole, di lunghezza fissa (di solito multipla di 8 bit, ad esempio 32 oppure 64 bit); cache; *registri* all'interno della Central Processing Unit (CPU).

La memoria fisica, sia essa principale, cache o banco dei registri, permette di memorizzare dati e programmi usando, come noto, l'alfabeto binario.

I dati sono divisi in pochi "tipi" primitivi: di solito abbiamo i numeri interi, i numeri cosiddetti "reali" (in realtà un sottoinsieme dei razionali), i caratteri, e le sequenze di bit di lunghezza fissa. A seconda del tipo di dati si hanno rappresentazioni fisiche diverse, che usano una o più parole di memoria per ogni elemento di un certo tipo. Ad esempio, gli interi potranno essere rappresentati in comple-

mento a 1 o a 2 usando una parola, mentre i reali potrebbero essere rappresentati in virgola mobile con una o due parole a seconda che siano in precisione semplice oppure doppia. I caratteri alfanumerici sono anch'essi rappresentati in termini di sequenze di cifre binarie usando un opportuno codice di rappresentazione (ad esempio, il codice ASCII oppure UNICODE).

Non entriamo qui nel dettaglio di questi meccanismi di rappresentazione che saranno analizzati con più precisione nel Capitolo 10. Sottolineiamo che, pur essendo tutti i dati rappresentati fisicamente mediante sequenze di bit, già a livello di macchina hardware possiamo distinguere varie categorie, o meglio, *tipi*, di dati primitivi che possono essere manipolati direttamente dalle operazioni che la macchina hardware può eseguire e che, per tale motivo, sono detti tipi predefiniti.

**Il linguaggio della macchina hardware** Il linguaggio  $\mathcal{L}\mathcal{H}$  che la macchina hardware è in grado di eseguire è costituito da istruzioni relativamente semplici: una tipica istruzione a due operandi di  $\mathcal{L}\mathcal{H}$ , ad esempio, occupa una parola di memoria ed ha il formato

CodiceOperativo Operando1 Operando2

dove CodiceOperativo è un particolare codice che identifica una delle operazioni primitive messe a disposizione dalla specifica macchina hardware, mentre Operando1 e Operando2 sono due valori che permettono di reperire i due operandi, facendo riferimento alle strutture di memorizzazione della macchina e le sue modalità di indirizzamento. Ad esempio,

ADD R5, R0

potrebbe indicare la somma del contenuto dei registri R0 e R5 e la memorizzazione del risultato in R5; mentre

ADD (R5), (R0)

potrebbe indicare la somma del contenuto delle celle di memoria i cui indirizzi sono contenuti in R0 e R5, con la memorizzazione del risultato nella cella il cui indirizzo è in R5. Si noti come in questi esempi si stiano usando codici simbolici quali ADD, R0, (R0) per motivi di chiarezza: nel linguaggio che stiamo considerando si tratta, invece, di valori numerici binari (e gli indirizzi sono espressi in modo "assoluto"). Dal punto di vista della rappresentazione interna le istruzioni non sono altro che dati che hanno un formato particolare. L'insieme delle istruzioni possibili (con i relativi operandi e modi di indirizzamento), così come le istruzioni e le strutture dati specifiche per gestire l'esecuzione dei programmi, dipendono dalla specifica macchina fisica, anche se si possono riconoscere delle classi di macchine con caratteristiche analoghe. Ad esempio, possiamo distinguere le macchine hardware convenzionali CISC (Complex Instruction Set Computers) che hanno numerose operazioni macchina, anche relativamente complesse, dalle architetture RISC (Reduced Instruction Set Computers) nelle quali si tende ad usare meno istruzioni e, soprattutto, più semplici in modo tale che possano essere eseguite in pochi (possibilmente uno) cicli di clock ed in pipeline.

**Interprete** Sempre facendo riferimento alla struttura generale di una macchina astratta, nel caso dell'interprete della macchina hardware possiamo riconoscere i seguenti componenti.

1. Le operazioni per l'elaborazione dei dati primitivi sono le usuali operazioni aritmetico-logiche realizzate dalla ALU (Arithmetic Logic Unit): operazioni aritmetiche su interi e numeri in virgola mobile, operazioni booleane, shift, confronto ecc.
  2. Riguardo al controllo della sequenza di esecuzione delle operazioni, la struttura dati principale è costituita dal registro Contatore Programma (o PC, Program Counter), che contiene l'indirizzo della prossima istruzione da eseguire. Le operazioni per il controllo di sequenza sono le operazioni specifiche per operare su tale registro, quali operazioni di incremento (per la gestione del normale flusso sequenziale di esecuzione) e operazioni per modificare il suo valore (per la gestione dei salti).
  3. Per controllare il trasferimento dei dati, le strutture canoniche sono costituite dai registri della CPU che interfacciano la memoria principale: il registro indirizzo dei dati (MAR, *Memory Address Register*) ed il registro dei dati (MDR, *Memory Data Register*). Vi sono inoltre opportune operazioni per modificare i contenuti di questi registri e per poter realizzare le varie modalità di indirizzamento (diretto, indiretto ecc.). Infine, troviamo le operazioni per accedere e modificare i registri interni della CPU.
  4. La gestione della memoria dipende in modo essenziale dall'architettura considerata. Nel caso più semplice di una macchina a registri senza multiprogrammazione, la gestione della memoria è rudimentale: il programma è caricato in memoria all'inizio dell'esecuzione e vi rimane fino alla sua terminazione. Per aumentare la velocità di calcolo, tuttavia, tutte le architetture moderne usano tecniche di gestione della memoria più sofisticate. In primo luogo troviamo livelli intermedi di memoria tra i registri e la memoria (cache), che richiedono opportune strutture dati e algoritmi per essere gestiti. In secondo luogo, è quasi sempre realizzata qualche forma di multiprogrammazione (l'esecuzione di un programma può venire sospesa per concedere la CPU ad altri programmi, per ottimizzare la gestione delle risorse quali l'accesso alla memoria secondaria). Queste tecniche, gestite dal sistema operativo, richiedono in genere il supporto di hardware specializzato per gestire la presenza contemporanea di più programmi in memoria al momento dell'esecuzione (ad esempio, per la rilocazione dinamica degli indirizzi).
- Tutte le tecniche sin qui descritte richiedono nella macchina hardware specifiche strutture dati ed operazioni per la gestione delle memorie. Vi sono inoltre altri casi che corrispondono ad architetture meno convenzionali: nel caso di una macchina (hardware) a pila invece che a registri, troveremo la struttura dati pila, con le relative operazioni di push e pop.

L'interprete della macchina hardware è realizzato da un insieme di dispositivi fisici che sostanzialmente costituiscono l'Unità di Controllo (UC) e che, usando anche le operazioni per il controllo di sequenza, permettono di eseguire il cosid-

detto ciclo *fetch-decode-execute*. Tale ciclo è analogo a quello di un generico interprete descritto in Figura 1.2 e consiste delle seguenti fasi.

Nella fase di *fetch*<sup>2</sup> viene recuperata dalla memoria la prossima istruzione da eseguire, ossia l'istruzione il cui indirizzo è contenuto nel registro PC (registro che viene quindi incrementato). L'istruzione, che ricordiamo è costituita da un codice operativo e da eventuali operandi, viene quindi memorizzata in un opportuno registro, detto registro istruzione.

Nella fase di *decode*, l'istruzione memorizzata nel registro istruzione viene decodificata usando opportuni circuiti logici: questi permettono di interpretare correttamente sia il codice operativo dell'istruzione sia le modalità di indirizzamento degli operandi, individuando quindi qual è l'operazione primitiva che deve essere eseguita e quali sono gli operandi. Gli operandi vengono quindi recuperati, secondo le modalità specificate nell'istruzione stessa, usando le operazioni di trasferimento dati.

Infine, nella fase di *execute*, viene effettivamente eseguita l'operazione primitiva della macchina, ad esempio usando i circuiti della ALU nel caso che si tratti di un'operazione aritmetico-logica. L'eventuale risultato è memorizzato, sulla base di quanto specificato dalle modalità di indirizzamento e dal codice operativo (sempre memorizzati nel registro istruzione), usando di nuovo le funzionalità di trasferimento dati. A questo punto, completata l'esecuzione dell'istruzione, si passa alla fase di fetch per l'istruzione successiva e il ciclo continua (a meno che l'istruzione eseguita non fosse un'istruzione di stop).

Si noti che, anche se concettualmente la macchina hardware distingue dati da istruzioni, a livello fisico non vi è alcuna distinzione fra di essi visto che entrambi sono internamente rappresentati come sequenze di bit. Tale distinzione avviene sostanzialmente in base allo stato della CPU: nello stato di fetch ogni parola prelevata dalla memoria è considerata come un'istruzione, mentre nella fase di execute ogni parola prelevata dalla memoria è considerata come un dato. Osserviamo, infine, come una descrizione accurata del funzionamento della macchina fisica richiederebbe l'introduzione di altri stati, oltre ai tre di fetch, decode ed execute. La discussione che abbiamo fatto ha il solo scopo di mostrare come la nozione generale di interprete si istanzi nel caso della macchina hardware.

## 1.2 Implementazione di un linguaggio

Abbiamo visto che una macchina astratta  $\mathcal{M}_{\mathcal{L}}$  è per definizione un dispositivo che permette di eseguire programmi scritti in  $\mathcal{L}$ . Una macchina astratta corrisponde dunque univocamente ad un linguaggio, il suo linguaggio macchina. Inversamente, dato un linguaggio di programmazione  $\mathcal{L}$ , vi sono molte (infinte) macchine astratte che hanno  $\mathcal{L}$  come proprio linguaggio macchina. Tali macchine differiscono tra loro nel modo in cui l'interprete è realizzato e nelle strutture dati che utilizzano, mentre tutte coincidono nel linguaggio interpretato, appunto  $\mathcal{L}$ .

<sup>2</sup>Dall'inglese *to fetch*, cercare, andare a prendere.

## La microprogrammazione

Le tecniche di microprogrammazione sono state introdotte negli anni '60 con lo scopo di permettere a tutta una gamma di calcolatori diversi, da quelli più lenti ed economici a quelli più veloci e costosi, di condividere lo stesso insieme di istruzioni e di avere quindi lo stesso linguaggio assembly (i calcolatori più noti per i quali la microprogrammazione fu usata sono quelli della serie IBM 360). Il linguaggio della macchina hardware microprogrammata è di livello estremamente basso e consiste di *microistruzioni* che specificano semplici operazioni di trasferimento dati fra registri, da e per la memoria principale ed eventualmente attraverso i circuiti logici che realizzano le operazioni aritmetiche. Ogni istruzione del linguaggio che si intende realizzare (cioè del linguaggio macchina che vede l'utente della macchina) è simulata mediante uno specifico insieme di microistruzioni. Queste microistruzioni che codificano le operazioni, unitamente ad un particolare insieme di microistruzioni che realizza il ciclo di interpretazione, costituiscono un *micropogramma* che risiede su speciali memorie di sola lettura (o comunque scrivibili solo con opportuni dispositivi) e realizza l'interprete del linguaggio (assembly) comune per i vari calcolatori con hardware diverso. Le macchine fisiche più evolute (e costose), avendo a disposizione hardware più potente, per realizzare una stessa istruzione usano meno simulazione rispetto ai modelli meno costosi, e quindi esibiscono maggiore velocità.

Dal punto di vista terminologico, nel caso di simulazione mediante micropogrammi si parla di *emulazione* e il livello della microprogrammazione è detto *firmware*.

Osserviamo, infine, come una macchina microprogrammata costituisca un primo, semplice esempio di *gerarchia* di due macchine astratte. Al livello più alto abbiamo la macchina assembly, costruita sopra quella che abbiamo chiamato la macchina microprogrammata. L'interprete della macchina assembly è realizzato (implementato) nel linguaggio di più basso livello (le microistruzioni), il quale è a sua volta interpretato direttamente dalle strutture fisiche della macchina microprogrammata. Discuteremo in modo approfondito questa situazione nel Paragrafo 1.3.

*Implementare* un linguaggio di programmazione  $\mathcal{L}$  significa realizzare una macchina astratta che abbia  $\mathcal{L}$  come linguaggio macchina. Prima di vedere quali sono le tecniche di implementazione effettivamente utilizzate per i linguaggi di programmazione attuali, vediamo innanzitutto quali sono le varie possibilità teoriche di realizzazione di una macchina astratta.

### 1.2.1 Realizzazione di una macchina astratta

Una qualsiasi macchina astratta  $\mathcal{M}_{\mathcal{L}}$  per essere effettivamente realizzata dovrà prima o poi utilizzare qualche dispositivo fisico (meccanico, elettronico, biologico ecc.) per eseguire le istruzioni di  $\mathcal{L}$ . L'uso di tale dispositivo, tuttavia, può essere esplicito o隐含的. Infatti, oltre alla realizzazione "fisica" (in hardware) dei

costrutti di  $\mathcal{M}_{\mathcal{L}}$ , possiamo anche pensare a realizzazioni che usino invece livelli (software o firmware) intermedi fra  $\mathcal{M}_{\mathcal{L}}$  e il dispositivo fisico di base. Possiamo dunque ricondurre le varie possibilità di realizzazione di una macchina astratta ai seguenti tre casi e alle loro combinazioni:

- realizzazione in *hardware*;
- simulazione mediante *software*;
- simulazione (emulazione) mediante *firmware*.

**Realizzazione in hardware** La realizzazione di  $\mathcal{M}_{\mathcal{L}}$  direttamente in hardware è, in linea di principio, sempre possibile ed è concettualmente abbastanza semplice: si tratta infatti di realizzare, mediante dispositivi fisici quali memorie, reti aritmetico-logiche, bus ecc., una macchina fisica tale che il suo linguaggio macchina coincida con  $\mathcal{L}$ . Per fare questo basta realizzare in hardware quelle strutture dati e algoritmi che costituiscono la macchina astratta<sup>3</sup>.

La realizzazione di una macchina  $\mathcal{M}_{\mathcal{L}}$  in hardware offre il vantaggio di permettere l'esecuzione dei programmi di  $\mathcal{L}$  in modo molto veloce, dato che questi sono eseguiti in modo diretto da dispositivi fisici. Questo vantaggio tuttavia è compensato da vari svantaggi che, nel caso in cui  $\mathcal{L}$  sia un generico linguaggio di alto livello, sono predominanti. Infatti i costrutti di un linguaggio di alto livello  $\mathcal{L}$  sono abbastanza complicati e molto lontani dalle funzionalità elementari disponibili a livello di circuiti fisici. Per poter realizzare  $\mathcal{M}_{\mathcal{L}}$  serve quindi una fase di progettazione fisica molto complicata, con la realizzazione di opportuni dispositivi fisici specifici per la macchina che vogliamo realizzare. Inoltre tale macchina, una volta realizzata, risulterebbe in pratica impossibile da modificare, per cui eventuali successive modifiche a  $\mathcal{L}$  sarebbero precluse, se non a fronte di costi proibitivi. Per questi motivi, nella pratica la realizzazione di  $\mathcal{M}_{\mathcal{L}}$  in hardware si usa solo per linguaggi di basso livello, i cui costrutti sono molto vicini alle operazioni che si possono definire in modo naturale usando i dispositivi fisici stessi. Si possono implementare direttamente in hardware anche alcuni linguaggi "dedicati", sviluppati per applicazioni specifiche nelle quali vi sia la necessità di ottenere la massima velocità di esecuzione. Questo è il caso, ad esempio, di alcuni linguaggi specifici per sistemi che operano in tempo reale.

Ciò non toglie che vi siano molti casi in cui la struttura della macchina astratta di un linguaggio di alto livello ha influenzato la realizzazione di un'architettura hardware, non nel senso di una diretta realizzazione in hardware della macchina astratta, ma nella scelta di operazioni primitive e strutture dati che permettessero una più semplice e efficiente realizzazione dell'interprete del linguaggio di alto livello. Questo è il caso, ad esempio, dell'architettura del B5500, un computer degli anni '60, influenzata dalla struttura del linguaggio ALGOL.

<sup>3</sup>Il Capitolo 5 affronterà la questione del perché questo sia sempre possibile nel caso di un linguaggio di programmazione.

**Simulazione mediante software** La seconda possibilità di realizzazione di una macchina astratta consiste nella realizzazione delle strutture dati e degli algoritmi di  $\mathcal{M}_{\mathcal{L}}$  mediante programmi scritti in un altro linguaggio  $\mathcal{L}'$ , che possiamo supporre già implementato. Avendo a disposizione una macchina  $\mathcal{M}'_{\mathcal{L}'}$  per il linguaggio  $\mathcal{L}'$ , possiamo infatti realizzare la macchina  $\mathcal{M}_{\mathcal{L}}$  mediante opportuni programmi scritti in  $\mathcal{L}'$  che interpreteranno i costrutti di  $\mathcal{L}$  simulando le funzionalità di  $\mathcal{M}_{\mathcal{L}}$ .

In questo caso avremo la massima flessibilità, potendo cambiare con facilità i programmi che realizzano i costrutti di  $\mathcal{M}_{\mathcal{L}}$ . Avremo tuttavia delle prestazioni inferiori al caso precedente, in quanto la realizzazione di  $\mathcal{M}_{\mathcal{L}}$  passa attraverso quella di un'altra macchina astratta  $\mathcal{M}'_{\mathcal{L}'}$ , che a sua volta dovrà essere realizzata in hardware, software o firmware, aggiungendo un ulteriore livello di interpretazione.

**Emulazione mediante firmware** La terza possibilità, infine, è intermedia fra la realizzazione in hardware e quella software e consiste nella simulazione (in questo caso detta anche emulazione) delle strutture dati e degli algoritmi di  $\mathcal{M}_{\mathcal{L}}$  mediante microprogrammi, che abbiamo sommariamente introdotto nel riquadro di pag. 10.

Concettualmente questa soluzione è simile alla simulazione mediante software: in entrambi i casi  $\mathcal{M}_{\mathcal{L}}$  è simulata mediante opportuni programmi che sono poi eseguiti da una macchina fisica. Tuttavia, nel caso della emulazione firmware tali programmi sono microprogrammi invece che programmi di un linguaggio di alto livello.

Come abbiamo visto nel riquadro, i microprogrammi usano uno speciale linguaggio di livello molto basso (con operazioni primitive estremamente semplici) e invece che nella memoria principale risiedono in un'opportuna memoria di sola lettura per poter essere eseguiti dalla macchina fisica ad alta velocità. Per questo motivo questa realizzazione di una macchina astratta permette di ottenere una velocità di esecuzione superiore a quella ottenibile con la simulazione software, anche se inferiore rispetto alla soluzione in hardware. D'altra parte, la flessibilità di questa soluzione è inferiore a quella della simulazione software, visto che, mentre è semplice modificare un programma scritto in un linguaggio di alto livello, modificare un microprogramma è relativamente complicato e richiede opportuni dispositivi per riscrivere le memorie sulle quali i microprogrammi sono memorizzati. La situazione è comunque migliore rispetto al caso della realizzazione in hardware, dato che i microprogrammi si possono comunque modificare.

Ovviamente, perché questa soluzione sia possibile la macchina fisica che si ha a disposizione deve essere micropogrammabile.

Sintetizzando, la realizzazione di  $\mathcal{M}_{\mathcal{L}}$  in hardware offre massima velocità, ma flessibilità nulla. La realizzazione mediante software offre massima flessibilità e minima velocità, quella mediante firmware è intermedia fra le due.

## Funzioni parziali

Una funzione  $f : A \rightarrow B$  è una corrispondenza tra elementi di  $A$  ed elementi di  $B$  tale che, per ogni elemento  $a$  di  $A$  esiste uno ed un solo elemento di  $B$  che gli corrisponde e che chiamiamo  $f(a)$ .

Una funzione parziale  $f : A \rightarrow B$  è ancora una corrispondenza tra i due insiemi  $A$  e  $B$ , ma essa può essere indefinita in corrispondenza di qualche elemento del dominio  $A$ . Più formalmente: è una relazione tra  $A$  e  $B$  tale che, per ogni  $a \in A$ , se esiste un corrispondente elemento in  $b$ , questo è unico e lo denotiamo con  $f(a)$ . La nozione di funzione parziale è per noi importante perché i programmi definiscono in modo naturale funzioni parziali. Ad esempio, il programma seguente (scritto in un linguaggio di ovvia sintassi e semantica

```
read(x);
if (x==1) then print(x);
else while (true) do skip;
```

calcola la funzione parziale

$$f(n) = \begin{cases} 1 & \text{se } x = 1 \\ \text{indefinita} & \text{altrimenti.} \end{cases}$$

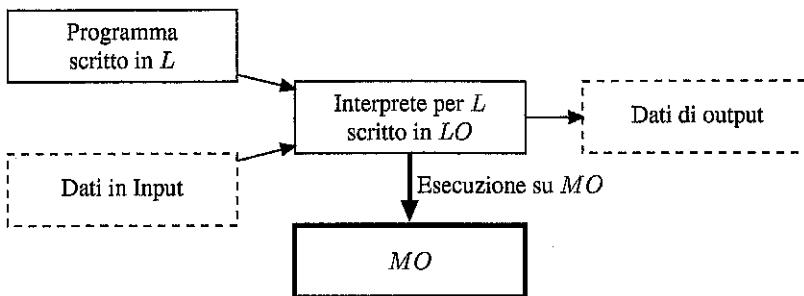
### 1.2.2 Implementazione: il caso ideale

Consideriamo un generico linguaggio  $\mathcal{L}$  che si vuole implementare, ovvero di cui si vuole realizzare una macchina astratta  $\mathcal{M}_{\mathcal{L}}$ . Assumendo di escludere, per i motivi visti, una realizzazione diretta di  $\mathcal{M}_{\mathcal{L}}$  in hardware, possiamo supporre di avere a disposizione, per la realizzazione di  $\mathcal{M}_{\mathcal{L}}$ , una macchina astratta  $\mathcal{M}_{\mathcal{L}_0}$ , che chiameremo *macchina ospite*, che è già stata realizzata (non ci interessa sapere come) e che quindi ci permette di usare direttamente i costrutti del suo linguaggio macchina  $\mathcal{L}_0$ .

Intuitivamente, l'implementazione di  $\mathcal{L}$  sulla macchina ospite  $\mathcal{M}_{\mathcal{L}_0}$  avviene mediante una qualche "traduzione" di  $\mathcal{L}$  in  $\mathcal{L}_0$ . Tuttavia, possiamo distinguere due modalità di implementazione, concettualmente molto diverse, a seconda del fatto che si abbia una traduzione "implicita" realizzata dalla simulazione dei costrutti  $\mathcal{M}_{\mathcal{L}}$  mediante programmi scritti in  $\mathcal{L}_0$ , oppure una traduzione esplicita dei programmi in  $\mathcal{L}$  in corrispondenti programmi in  $\mathcal{L}_0$ . Vediamo innanzitutto queste due modalità nelle loro versioni ideali che chiamiamo:

1. *implementazione interpretativa pura*;
2. *implementazione compilativa pura*.

**Notazione** Nel seguito, come già fatto in precedenza, usiamo il pedice  $\mathcal{L}$  per indicare che un particolare costrutto (macchina, interprete, programma ecc.) si riferisce al linguaggio  $\mathcal{L}$ , mentre usiamo l'apice  $\mathcal{L}$  per indicare che un programma



**Figura 1.4** Implementazione interpretativa pura.

è scritto nel linguaggio  $\mathcal{L}$ . Indichiamo poi con  $\text{Prog}^{\mathcal{L}}$  l'insieme di tutti i possibili programmi che si possono scrivere nel linguaggio  $\mathcal{L}$ , mentre  $\mathcal{D}$  denota l'insieme dei dati di input e output (che per semplicità di trattazione non distinguiamo tra loro).

Un programma scritto in  $\mathcal{L}$  può essere visto come una funzione parziale (si veda il riquadro)

$$\mathcal{P}^{\mathcal{L}} : \mathcal{D} \rightarrow \mathcal{D}$$

tale che

$$\mathcal{P}^{\mathcal{L}}(\text{Input}) = \text{Output}$$

se l'esecuzione di  $\mathcal{P}^{\mathcal{L}}$  sul dato di ingresso  $\text{Input}$  termina e produce come risultato  $\text{Output}$ , mentre la funzione non è definita se l'esecuzione di  $\mathcal{P}^{\mathcal{L}}$  sul dato di ingresso  $\text{Input}$  non termina<sup>4</sup>.

**Implementazione interpretativa pura** Nell'*implementazione interpretativa pura*, schematizzata in Figura 1.4, si realizza l'interprete di  $\mathcal{M}_{\mathcal{L}}$  mediante un insieme di istruzioni in  $\mathcal{L}_0$ . Si realizza cioè un programma, che è un interprete e chiameremo  $\mathcal{I}_{\mathcal{L}}^{\mathcal{L}_0}$ , scritto in  $\mathcal{L}_0$  che interpreta tutte le possibili istruzioni di  $\mathcal{L}$ .

Una volta che un tale interprete sia realizzato, per eseguire un programma  $\mathcal{P}^{\mathcal{L}}$  (scritto nel linguaggio  $\mathcal{L}$ ) con un certo dato di input  $D \in \mathcal{D}$ , dovremo semplicemente eseguire, sulla macchina  $\mathcal{M}_{\mathcal{L}_0}$ , il programma  $\mathcal{I}_{\mathcal{L}}^{\mathcal{L}_0}$  con  $\mathcal{P}^{\mathcal{L}}$  e  $D$  come dati di input. Con più precisione possiamo dare la seguente definizione.

**Definizione 1.3 (Interprete)** Un interprete per il linguaggio  $\mathcal{L}$ , scritto nel linguaggio  $\mathcal{L}_0$ , è un programma che realizza una funzione parziale

$$\mathcal{I}_{\mathcal{L}}^{\mathcal{L}_0} : (\text{Prog}^{\mathcal{L}} \times \mathcal{D}) \rightarrow \mathcal{D} \quad \text{tale che} \quad \mathcal{I}_{\mathcal{L}}^{\mathcal{L}_0}(\mathcal{P}^{\mathcal{L}}, \text{Input}) = \mathcal{P}^{\mathcal{L}}(\text{Input}). \quad (1.1)$$

<sup>4</sup>Si noti che non c'è perdita di generalità a considerare un solo dato di input, dato che questo può codificare un qualsiasi insieme di dati.

Il fatto che un programma possa essere considerato come un dato di input per un altro programma non deve meravigliare, visto che, come già accennato, un programma non è altro che un insieme di istruzioni che in ultima analisi sono rappresentate da un certo insieme di simboli (e quindi da sequenze di bit).

Nell'implementazione interpretativa pura di  $\mathcal{L}$ , dunque, non vi è una traduzione esplicita dei programmi scritti in  $\mathcal{L}$ . Vi è solo un procedimento di "decodifica": l'interprete  $\mathcal{I}_{\mathcal{L}}^{\mathcal{L}_0}$ , per eseguire un'istruzione del linguaggio  $\mathcal{L}$ , le fa corrispondere un certo insieme di istruzioni di  $\mathcal{L}_0$ . Tale decodifica non è una traduzione esplicita perché il codice corrispondente ad un'istruzione di  $\mathcal{L}$  viene eseguito, non prodotto in uscita dall'interprete.

Si noti che volutamente non abbiamo specificato la natura della macchina  $\mathcal{M}_{\mathcal{L}_0}$ : il linguaggio  $\mathcal{L}_0$  può dunque essere un linguaggio di alto livello, un linguaggio di basso livello o anche un linguaggio di firmware.

**Implementazione compilativa pura** Nell'*implementazione compilativa pura*, esemplificata in Figura 1.5, l'implementazione di  $\mathcal{L}$  avviene traducendo esplicitamente i programmi scritti in  $\mathcal{L}$  in programmi scritti in  $\mathcal{L}_0$ . La traduzione è eseguita da un opportuno programma, detto *compilatore* e che noi indicheremo con  $\mathcal{C}_{\mathcal{L}, \mathcal{L}_0}$ . In questo contesto il linguaggio  $\mathcal{L}$  è detto comunemente *linguaggio sorgente*, mentre il linguaggio  $\mathcal{L}_0$  è il *linguaggio oggetto*. Per eseguire un programma  $\mathcal{P}^{\mathcal{L}}$  (scritto nel linguaggio  $\mathcal{L}$ ) con un dato di input  $D$ , dovremo innanzitutto eseguire  $\mathcal{C}_{\mathcal{L}, \mathcal{L}_0}$  con  $\mathcal{P}^{\mathcal{L}}$  come input. Questo produrrà come risultato un programma compilato  $\mathcal{P}^{\mathcal{L}_0}$  scritto in  $\mathcal{L}_0$ . A questo punto potremo eseguire  $\mathcal{P}^{\mathcal{L}_0}$ , sulla macchina  $\mathcal{M}_{\mathcal{L}_0}$ , con il dato di input  $D$ , per ottenere il risultato desiderato.

**Definizione 1.4 (Compilatore)** Un compilatore da  $\mathcal{L}$  a  $\mathcal{L}_0$  è un programma che realizza una funzione

$$\mathcal{C}_{\mathcal{L}, \mathcal{L}_0} : \text{Prog}^{\mathcal{L}} \rightarrow \text{Prog}^{\mathcal{L}_0}$$

tale che, dato un programma  $\mathcal{P}^{\mathcal{L}}$ , se

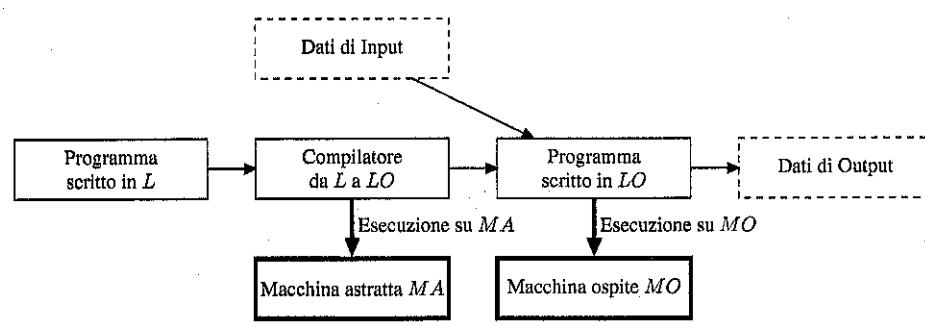
$$\mathcal{C}_{\mathcal{L}, \mathcal{L}_0}(\mathcal{P}^{\mathcal{L}}) = \mathcal{P}^{\mathcal{L}_0} \quad (1.2)$$

allora, per ogni  $\text{Input} \in \mathcal{D}^5$ ,

$$\mathcal{P}^{\mathcal{L}}(\text{Input}) = \mathcal{P}^{\mathcal{L}_0}(\text{Input}). \quad (1.3)$$

Si noti che, a differenza di quanto avviene con la implementazione interpretativa pura, qui la fase di traduzione, descritta dalla (1.2) e detta *compilazione*, è separata dalla fase di esecuzione, indicata invece dalla (1.3). La compilazione infatti produce come risultato un programma che potremo eseguire successivamente, quando vorremo. È da osservare che se  $\mathcal{M}_{\mathcal{L}_0}$  è l'unica macchina che

<sup>5</sup>Si noti che, per semplicità, supponiamo che i dati su cui i programmi operano siano gli stessi per il linguaggio sorgente e per quello oggetto. Se così non fosse anche i dati dovrebbero essere opportunamente tradotti.



**Figura 1.5** Implementazione compilativa pura.

abbiamo a disposizione, e dunque  $\mathcal{L}_0$  è l'unico linguaggio che possiamo usare, il compilatore sarà anch'esso un programma scritto in  $\mathcal{L}_0$ . Tuttavia questo non è essenziale: il compilatore potrebbe infatti essere eseguito anche su un'altra macchina astratta che usa un linguaggio diverso, pur producendo codice eseguibile su  $\mathcal{M}_{\mathcal{L}_0}$ .

**Un confronto fra le due tecniche** Avendo presentato sia l'implementazione puramente interpretativa che quella puramente compilativa discutiamo adesso i vantaggi e gli svantaggi dei due approcci.

Per quanto riguarda l'implementazione interpretativa pura lo svantaggio principale risiede nella sua *scarsa efficienza*. Difatti, dato che non vi è una fase preliminare di traduzione, per eseguire il programma  $\mathcal{P}^{\mathcal{L}}$  l'interprete  $\mathcal{I}_{\mathcal{L}}^{\mathcal{L}_0}$  deve effettuare al momento dell'esecuzione una decodifica dei costrutti del linguaggio  $\mathcal{L}$ . Ai tempi di esecuzione intrinsecamente richiesti dalle operazioni di  $\mathcal{P}^{\mathcal{L}}$  si devono sommare quindi anche i tempi necessari alla decodifica. Ad esempio, se il linguaggio  $\mathcal{L}$  contenesse il comando iterativo `for` e tale comando non fosse presente nel linguaggio  $\mathcal{L}_0$ , per poter eseguire un comando quale

P1: `for (I = 1, I<=n, I=I+1) C;`

l'interprete  $\mathcal{I}_{\mathcal{L}}^{\mathcal{L}_0}$  dovrebbe decodificare tale comando al momento dell'esecuzione, per eseguire al suo posto una serie di operazioni che realizzano il ciclo e che potrebbero essere esemplificate dal seguente frammento di codice:

```

P2:
  R1 = 1
  R2 = n
L1: if R1 > R2 then goto L2
     traduzione di C
...
  R1 = R1 + 1
  goto L1
L2: ...
  
```

È importante ripetere che, come evidente dall'equazione (1.1), l'interprete non genera codice: il codice appena discusso non è prodotto esplicitamente dall'interprete, ma descrive soltanto le operazioni che questo deve effettuare, al momento dell'esecuzione, dopo aver decodificato il comando `for`.

Va anche osservato che per ogni nuova occorrenza di uno stesso comando nel programma scritto in  $\mathcal{L}$  l'interprete deve effettuare una nuova decodifica (sempre durante l'esecuzione), il che non migliora certo l'efficienza. Nel nostro esempio, la decodifica del comando `C` all'interno del ciclo deve essere eseguita  $n$  volte con evidente conseguente inefficienza.

Come spesso accade, gli svantaggi in termini di efficienza sono compensati da vantaggi in termini di *flessibilità*. Infatti, interpretare al momento dell'esecuzione i costrutti del programma che vogliamo eseguire permette di interagire in modo diretto con l'esecuzione di un programma. Questo è particolarmente importante, ad esempio, per poter definire in modo relativamente semplice strumenti di debugging dei programmi. In generale, inoltre, lo sviluppo di un interprete è più semplice dello sviluppo di un compilatore, per cui si preferiscono soluzioni interpretative nel caso in cui si voglia implementare in tempi (relativamente) brevi un nuovo linguaggio. Va ricordato infine che, anche se questa preoccupazione oggi non è più particolarmente sentita, l'implementazione interpretativa permette di usare una quantità di memoria molto ridotta, dato che il programma è memorizzato solo nella sua versione sorgente (cioè nel linguaggio  $\mathcal{L}$ ) e non viene prodotto nuovo codice.

I vantaggi e gli svantaggi dell'approccio compilativo all'implementazione dei linguaggi sono duali rispetto a quelli visti per l'implementazione interpretativa.

La traduzione del programma sorgente  $\mathcal{P}^{\mathcal{L}}$  in un programma oggetto  $\mathcal{P}^{\mathcal{L}_0}$  avviene separatamente dall'esecuzione di quest'ultimo. Se trascuriamo il tempo necessario alla compilazione, dunque, l'esecuzione di  $\mathcal{P}^{\mathcal{L}_0}$  risulterà più efficiente di quello che si otterebbe con un'implementazione interpretativa, in quanto non appesantisca dalla fase di decodifica delle istruzioni. Nel nostro esempio di prima, il frammento di programma P1 sarà tradotto in P2 dal compilatore. Successivamente, quando necessario, P2 sarà eseguito senza dover a questo punto ripetere la decodifica dell'istruzione `for`. Inoltre, a differenza di quanto avviene con l'interprete, la decodifica di una istruzione del linguaggio  $\mathcal{L}$  viene fatta dal compilatore una sola volta, indipendentemente dal numero di occorrenze di tale istruzione al momento dell'esecuzione. Nel nostro esempio, il comando `C` viene decodificato e tradotto una sola volta al momento della compilazione, e il codice prodotto da tale traduzione viene eseguito  $n$  volte al momento dell'esecuzione. Discuteremo nel Paragrafo 2.4 della struttura di un compilatore e delle ottimizzazioni che esso può apportare al codice prodotto.

Uno degli svantaggi maggiori dell'approccio compilativo risiede nella perdita di informazioni riguardo alla struttura del programma sorgente, perdita che rende più difficile l'interazione con il programma a tempo di esecuzione. Ad esempio, se a run-time si verificasse un errore potrebbe essere difficile determinare qual è il comando del programma sorgente che lo ha determinato, visto che tale comando è stato compilato in una sequenza di istruzioni del linguaggio oggetto.

### L'interprete e il compilatore si possono realizzare sempre?

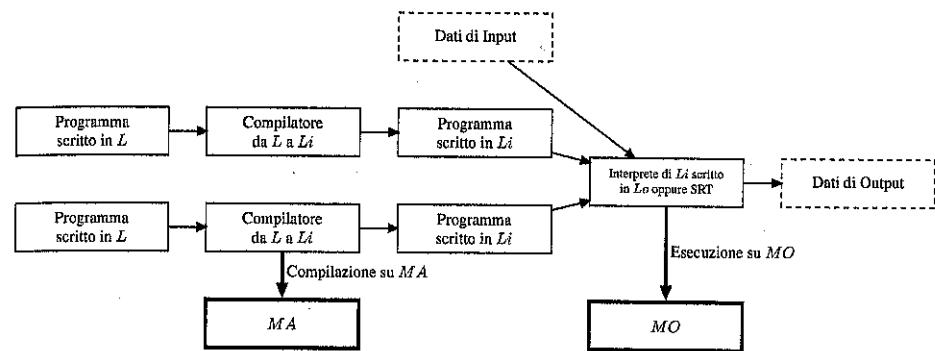
A questo punto il lettore potrebbe domandarsi se la realizzazione di un interprete o di un compilatore sia sempre possibile. Ossia, dato il linguaggio  $\mathcal{L}$  che si vuole implementare, come si può essere sicuri che sia possibile realizzare nel linguaggio  $\mathcal{L}_o$  un particolare programma  $\mathcal{I}_{\mathcal{L}}^{\mathcal{L}_o}$  che riesce ad interpretare tutti i costrutti di  $\mathcal{L}$ ? E come si può essere sicuri del fatto che sia possibile tradurre i programmi di  $\mathcal{L}$  nei programmi di  $\mathcal{L}_o$  usando un opportuno programma  $C_{\mathcal{L}, \mathcal{L}_o}$ ?

La risposta precisa a queste domande richiede delle nozioni di teoria della calcolabilità che saranno introdotte nel Capitolo 5. Per adesso possiamo solo rispondere che l'esistenza dell'interprete e del compilatore è garantita a patto che il linguaggio  $\mathcal{L}_o$  che usiamo per l'implementazione sia sufficientemente espressivo rispetto al linguaggio  $\mathcal{L}$  che vogliamo implementare. Come vedremo, tutti i linguaggi di uso comune, e quindi anche il nostro  $\mathcal{L}_o$ , hanno lo stesso (massimo) potere espressivo, che coincide con quello di un certo modello astratto di calcolatore che chiameremo Macchina di Turing. Questo significa che ogni possibile algoritmo che siamo in grado di formulare può essere realizzato da un programma scritto in  $\mathcal{L}_o$ . Dato che l'interprete per  $\mathcal{L}$  altro non è, come abbiamo detto, che un particolare algoritmo in grado di eseguire le istruzioni di  $\mathcal{L}$ , evidentemente non ci sono difficoltà di ordine teorico nella realizzazione dell'interprete  $\mathcal{I}_{\mathcal{L}}^{\mathcal{L}_o}$ . Per il compilatore, assumendo che esso sia scritto in  $\mathcal{L}_o$ , il discorso è analogo: visto che  $\mathcal{L}$  non è più espressivo di  $\mathcal{L}_o$ , deve essere possibile tradurre i programmi di  $\mathcal{L}$  in quelli di  $\mathcal{L}_o$  in modo tale da mantenerne il significato. Inoltre, visto che  $\mathcal{L}_o$  permette di realizzare un qualsiasi algoritmo per ipotesi, esso permetterà anche di realizzare il particolare programma compilatore  $C_{\mathcal{L}, \mathcal{L}_o}$  che realizza la traduzione.

In questo caso può essere più difficile dunque realizzare strumenti di debugging; più in generale, si ha una flessibilità minore rispetto a quella dell'approccio interpretativo.

### 1.2.3 Implementazione: il caso reale e la macchina intermedia

Le implementazioni compilative e interpretative pure possono essere considerate i due estremi di quello che avviene nella pratica quando si implementa un linguaggio. Difatti, nelle implementazioni dei linguaggi reali sono quasi sempre presenti entrambe le componenti. Per quanto riguarda l'implementazione interpretativa, osserviamo subito che ogni interprete "reale" lavora su una rappresentazione interna dei programmi che non coincide quasi mai con la rappresentazione esterna dei programmi stessi. Il passaggio dalla notazione esterna di  $\mathcal{L}$  alla rappresentazione interna si configura come una vera e propria traduzione (compilazione, nella nostra terminologia) da  $\mathcal{L}$  ad un linguaggio intermedio, che è quello interpretato. Analogamente, in ogni implementazione compilativa alcuni costrutti particolarmente complicati sono simulati. Ad esempio, alcune istruzioni di ingresso/uscita per essere tradotte nel linguaggio della macchina hardware richiedrebbero varie centinaia di istruzioni, per cui si preferisce tradurle in una chiamata



**Figura 1.6** Implementazione: il caso reale con la macchina intermedia.

ad un opportuno programma (o direttamente al sistema operativo) che al momento dell'esecuzione simula (e dunque interpreta) tali istruzioni.

La situazione reale dell'implementazione di un linguaggio di alto livello è dunque quella descritta in Figura 1.6. Supponiamo, come in precedenza, di avere un linguaggio  $\mathcal{L}$  che deve essere implementato e di disporre di una macchina ospite  $\mathcal{M}_{\mathcal{L}_o}$  già realizzata. Fra la macchina  $\mathcal{M}_{\mathcal{L}}$  che vogliamo realizzare e la macchina ospite esiste un ulteriore livello caratterizzato da un proprio linguaggio  $\mathcal{L}_i$  e dalla relativa macchina astratta  $\mathcal{M}_{\mathcal{L}_i}$ , che chiameremo rispettivamente linguaggio intermedio e macchina intermedia.

Come illustrato dalla Figura 1.6, abbiamo sia un compilatore  $C_{\mathcal{L}, \mathcal{L}_i}$  che traduce da  $\mathcal{L}$  a  $\mathcal{L}_i$ , sia un interprete  $\mathcal{I}_{\mathcal{L}_i}^{\mathcal{L}_o}$  che "gira" sulla macchina  $\mathcal{M}_{\mathcal{L}_o}$  e che simula la macchina  $\mathcal{M}_{\mathcal{L}_i}$ . Un generico programma  $\mathcal{P}^{\mathcal{L}}$  per essere eseguito è prima tradotto dal compilatore in un programma  $\mathcal{P}^{\mathcal{L}_i}$  del linguaggio intermedio. Quindi questo programma è eseguito dall'interprete  $\mathcal{I}_{\mathcal{L}_i}^{\mathcal{L}_o}$ . Si noti che nella figura abbiamo indicato "interprete o supporto a run time" (SRT) perché non sempre è necessario realizzare interamente l'interprete  $\mathcal{I}_{\mathcal{L}_i}^{\mathcal{L}_o}$ : nel caso in cui il linguaggio intermedio e il linguaggio della macchina ospite non siano molto distanti, per simulare la macchina intermedia può bastare l'interprete della macchina ospite, esteso da opportuni programmi detti, appunto, supporto a run-time.

A seconda di quanto il livello intermedio sia spostato verso il livello sorgente o il livello ospite, avremo vari tipi di implementazione. Schematizzando, possiamo identificare i seguenti casi.

1.  $\mathcal{M}_{\mathcal{L}} = \mathcal{M}_{\mathcal{L}_i}$ : implementazione puramente interpretativa.
2.  $\mathcal{M}_{\mathcal{L}} \neq \mathcal{M}_{\mathcal{L}_i} \neq \mathcal{M}_{\mathcal{L}_o}$ :
  - (a) Se l'interprete della macchina intermedia è sostanzialmente diverso dall'interprete di  $\mathcal{M}_{\mathcal{L}_o}$ , diremo che siamo in presenza di un'implementazione di tipo interpretativo.

(b) Se l'interprete della macchina intermedia è sostanzialmente uguale all'interprete di  $Mo_{\mathcal{L}_0}$  (di cui estende alcune funzionalità), diremo che siamo in presenza di un'implementazione di *tipo compilativo*.

3.  $Mi_{\mathcal{L}_i} = Mo_{\mathcal{L}_0}$ : implementazione puramente compilativa.

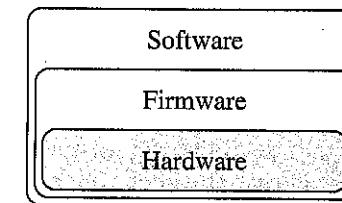
Il primo e l'ultimo caso, nei quali la macchina intermedia coincide rispettivamente con la macchina del linguaggio da implementare e con la macchina ospite, corrispondono ai casi limite già visti nel paragrafo precedente.

Nel caso invece in cui la macchina intermedia sia effettivamente presente, si parla di implementazione di tipo *interpretativo* quando l'interprete della macchina intermedia è sostanzialmente diverso dall'interprete di  $Mo_{\mathcal{L}_0}$ . In questo caso dunque si deve realizzare, usando il linguaggio  $\mathcal{L}_0$ , l'interprete  $I_{\mathcal{L}_i}^{\mathcal{L}_0}$ . La differenza di questa soluzione da quella puramente interpretativa risiede nel fatto che non tutti i costrutti di  $\mathcal{L}$  devono essere simulati: per alcuni di essi, una volta tradotti da  $\mathcal{L}$  nel linguaggio intermedio  $\mathcal{L}_i$ , esiste un corrispettivo diretto nel linguaggio della macchina ospite, per cui per questi non serve alcuna simulazione. Tuttavia la distanza fra  $Mi_{\mathcal{L}_i}$  e  $Mo_{\mathcal{L}_0}$  è tale che i costrutti per i quali questo avviene sono pochi e dunque l'interprete della macchina intermedia deve essere simulato in molte delle sue componenti.

Nell'implementazione di tipo *compilativo*, invece, la macchina intermedia è più vicina alla macchina ospite e sostanzialmente ne condivide l'interprete. In questo caso quindi, la macchina intermedia  $Mi_{\mathcal{L}_i}$  sarà realizzata usando funzionalità di  $Mo_{\mathcal{L}_0}$ , opportunamente estese per quei costrutti del linguaggio sorgente  $\mathcal{L}$  che, anche quando tradotti nel linguaggio intermedio  $\mathcal{L}_i$ , non trovano un corrispondente immediato nella macchina ospite. Questo è il caso, ad esempio, di alcune operazioni di ingresso/uscita che anche nelle implementazioni più compilative di solito vengono simulate da opportuni programmi scritti in  $\mathcal{L}_0$ . L'insieme di questi programmi che estendono le funzionalità della macchina ospite e che a tempo di esecuzione simulano alcune funzionalità del linguaggio  $\mathcal{L}_i$ , e quindi del linguaggio  $\mathcal{L}$ , costituiscono il cosiddetto *supporto a run time* di  $\mathcal{L}$ .

Come si può comprendere dalla discussione appena fatta, la distinzione tra i casi intermedi non è netta, ma esiste un intero spettro di tipi di implementazione che vanno dal caso in cui tutto è simulato al caso in cui invece tutto è tradotto nel linguaggio della macchina ospite. Cosa simulare e cosa tradurre dipende molto dal linguaggio in questione e dalla macchina ospite disponibile. Evidentemente, almeno in linea di principio, si tenderà ad interpretare i costrutti del linguaggio da implementare che sono molto lontani dal linguaggio della macchina ospite e si tenderà a compilare il resto. Inoltre, al solito, si privilegeranno soluzioni di tipo compilativo nei casi in cui si voglia aumentare l'efficienza di esecuzione dei programmi, mentre ci si orienterà all'approccio interpretativo nei casi in cui si preferisca una maggiore flessibilità.

Si noti anche che la macchina intermedia, anche se sempre presente in linea di principio, spesso non è resa esplicita. Fanno eccezione alcuni casi di linguaggi per i quali sono state definite formalmente delle macchine intermedie, con i relativi linguaggi, principalmente per motivi di portabilità. L'implementazione compilativa di un linguaggio su una nuova piattaforma hardware, infatti, è un compito impe-



**Figura 1.7** I tre livelli di un calcolatore microprogrammato.

gnativo che richiede uno sforzo progettuale non indifferente. L'implementazione interpretativa è meno pesante, ma richiede anch'essa un certo impegno e inoltre spesso pone problemi di efficienza. Spesso si vuole implementare un linguaggio su molte piattaforme diverse, ad esempio quando occorre inviare programmi sulla rete per eseguirli su calcolatori remoti (come accade con i cosiddetti *applet*). In tal caso è molto conveniente compilare prima i programmi in un linguaggio intermedio e quindi implementare (in modo interpretativo) tale linguaggio intermedio sulle varie piattaforme. Evidentemente l'implementazione del codice intermedio risulta molto più semplice dell'implementazione del codice sorgente, vista la fase di compilazione effettuata. Questa soluzione alla portabilità delle implementazioni è stata adottata per la prima volta su larga scala con il linguaggio Pascal, che venne definito insieme ad una macchina intermedia (con il suo linguaggio *P-code*) progettata proprio per questo scopo. Una soluzione analoga è usata dal linguaggio Java, la cui macchina intermedia, detta JVM (Java Virtual Machine), ha come linguaggio macchina il cosiddetto Java Byte-Code ed è oramai implementata su ogni calcolatore.

Come ultima nota sottolineiamo il fatto, evidente da quanto abbiamo sin qui detto, che non si dovrebbe parlare di "linguaggio interpretato" o "linguaggio compilato": ogni linguaggio può essere implementato con entrambe le tecniche. Si dovrebbe invece parlare di implementazione interpretativa o di implementazione compilativa di un linguaggio.

### 1.3 Gerarchie di macchine astratte

In base a quanto abbiamo visto, un calcolatore microprogrammato sul quale sia implementato un linguaggio di programmazione di alto livello può essere rappresentato da uno schema quale quello di Figura 1.7: ogni livello realizza una macchina astratta con un proprio linguaggio ed un proprio insieme di funzionalità.

Questo schema può essere esteso ad un numero arbitrario di livelli ed una gerarchia siffatta, anche se non sempre esplicitata, è largamente usata nei progetti software. Si usano spesso, in altre parole, gerarchie di macchine astratte in cui ogni macchina usa le funzionalità del livello sottostante e offre nuove funzionalità al livello soprastante. Gli esempi di gerarchie di questo tipo sono molteplici; si

pensi ad esempio alla semplice attività di programmazione: quando scriviamo un programma  $\mathcal{P}$  in un linguaggio  $\mathcal{L}$  in effetti non facciamo altro che definire un nuovo linguaggio  $\mathcal{L}_p$  (e quindi una nuova macchina astratta) costituito dalle (nuove) funzionalità che  $\mathcal{P}$  mette a disposizione dell'utente attraverso la sua interfaccia. Un tale programma può quindi essere usato da un altro programma, che definirà nuove funzionalità, e quindi un nuovo linguaggio e così via. Si noti che in senso lato possiamo parlare di macchine astratte anche quando si abbia a che fare con un insieme di comandi che, a rigore, non costituirebbero un vero e proprio linguaggio di programmazione. Questo è il caso di un programma, delle funzionalità di un sistema operativo, o anche delle funzionalità di un livello middleware in una rete di calcolatori.

Nel caso generale, dunque, dobbiamo ipotizzare una gerarchia di macchine  $M_{\mathcal{L}_0}, M_{\mathcal{L}_1}, \dots, M_{\mathcal{L}_n}$ . La generica macchina  $M_{\mathcal{L}_i}$  è implementata sfruttando le funzionalità (cioè il linguaggio) della macchina sottostante  $M_{\mathcal{L}_{i-1}}$ ; contemporaneamente  $M_{\mathcal{L}_i}$  fornisce il proprio linguaggio  $\mathcal{L}_i$  alla macchina sovrastante  $M_{\mathcal{L}_{i+1}}$ , che sfruttando tale linguaggio utilizza le nuove funzionalità che  $M_{\mathcal{L}_i}$  offre rispetto ai livelli inferiori. Spesso tale gerarchia ha anche lo scopo di mascherare i livelli inferiori:  $M_{\mathcal{L}_i}$  non può accedere direttamente alle risorse delle macchine sottostanti, ma solo sfruttare quello che fornisce il linguaggio  $\mathcal{L}_{i-1}$ .

Una tale strutturazione di un sistema software in livelli di macchina astratte è utile per dominare la complessità del sistema e, soprattutto, permette una certa indipendenza fra i vari livelli, nel senso che una qualsiasi modifica all'interno delle funzionalità di un livello non ha (o non dovrebbe avere) alcuna influenza sugli altri livelli. Ad esempio, se usiamo un linguaggio di alto livello  $\mathcal{L}$  che usa i meccanismi di gestione dei file di un sistema operativo, un'eventuale modifica di tali meccanismi (purché l'interfaccia rimanga la stessa) non influenza un eventuale programma scritto in  $\mathcal{L}$ .

Un esempio canonico di gerarchie di questo tipo in un contesto apparentemente lontano dai linguaggi di programmazione è costituito dalla gerarchia<sup>6</sup> dei protocolli di comunicazione in una rete di calcolatori, quali ad esempio la gerarchia dello standard ISO/OSI.

In un contesto più vicino all'argomento di questo libro, vediamo l'esempio della Figura 1.8.

Al livello più basso abbiamo un calcolatore hardware, realizzato mediante dispositivi fisici elettronici (almeno per il momento, in futuro la possibilità di dispositivi biologici sarà da considerarsi concretamente). Sopra questo livello potremmo avere il livello della macchina astratta microprogrammata. immediatamente sopra (o direttamente sopra il livello hardware se il livello firmware non è presente) troviamo la macchina astratta fornita dal sistema operativo, realizzata mediante programmi in linguaggio macchina. Una tale macchina può essere a sua volta vista come un gerarchia di vari livelli (nucleo, gestore della memoria, gestore delle periferiche, file system, interprete dei comandi) che implementano

<sup>6</sup>Nella letteratura sulle reti si parla spesso di pila (*stack*) invece che, più correttamente, di gerarchia.

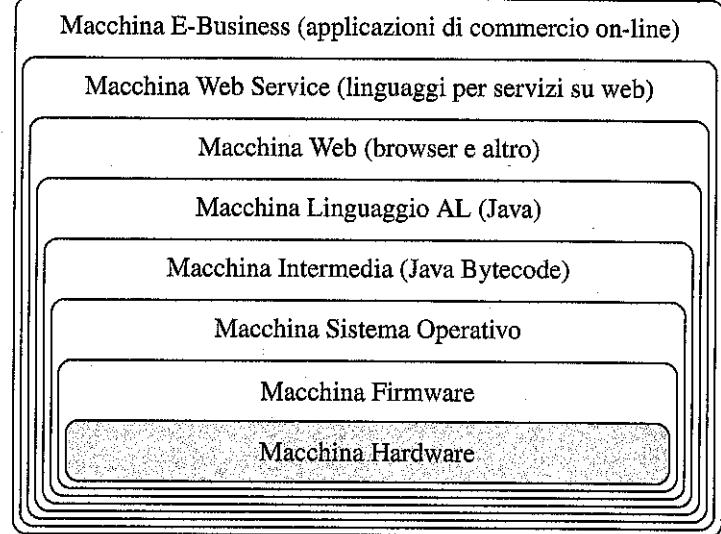


Figura 1.8 Una gerarchia di macchine astratte.

funzionalità via via più lontane dalla macchina fisica: dal nucleo, che interagisce con l'hardware e gestisce lo stato d'avanzamento dei processi, fino all'interprete dei comandi (o shell) che permette all'utente l'interazione con il sistema operativo. Nel suo complesso dunque il sistema operativo da un lato estende le funzionalità della macchina fisica offrendo ai livelli superiori alcune funzionalità non presenti nella macchina fisica (ad esempio, primitive per operare sui file), dall'altro maschera alcune primitive hardware (ad esempio, primitive per gestire l'ingresso/uscita) che i livelli superiori della gerarchia non hanno alcun interesse a vedere direttamente. La macchina astratta fornita dal sistema operativo costituisce la macchina ospite sulla quale è implementato un linguaggio di alto livello, secondo una delle modalità che abbiamo visto nelle sezioni precedenti, e quindi normalmente con la presenza di una macchina intermedia che nello schema di Figura 1.8 è la Java Virtual Machine, col proprio linguaggio Bytecode. Il livello fornito dalla macchina astratta del linguaggio di alto livello che abbiamo implementato, Java nel nostro caso, normalmente non è l'ultimo livello della gerarchia. A questo punto infatti possiamo avere una o più applicazioni che complessivamente forniscono nuovi servizi. Ad esempio, possiamo avere un ulteriore livello "macchina web" nel quale vengono implementate le funzionalità necessarie a gestire la comunicazione sul Web (protocolli di comunicazione, visualizzazione di codice HTML, esecuzione di applets ecc.). Ancora sopra, possiamo trovare il livello "Web Service" nel quale vengono fornite le funzionalità necessarie a far interagire servizi web, sia in termini di protocolli di interazione che in termini di comportamento dei processi coinvolti. A questo livello possono essere definiti dei

### Trasformazioni di programmi e valutazione parziale

Oltre alle “traduzioni” di programmi da un linguaggio ad un altro, quali quelle effettuate dai compilatori, vi sono numerose tecniche di trasformazione di programmi all’interno di uno stesso linguaggio, principalmente definite con lo scopo di migliorare l’efficienza. La valutazione parziale è una di queste tecniche e consiste nel valutare un programma del quale sia nota una parte dell’input, in modo tale da ottenere un programma specializzato rispetto a tale input, che sia più efficiente del programma originale. Ad esempio, se abbiamo un programma  $P(X, Y)$  che, dopo aver elaborato i dati forniti in  $X$ , a seconda del risultato di questa prima fase effettua delle operazioni sui dati in  $Y$ , evidentemente, se i dati  $X$  sono sempre gli stessi, possiamo trasformare tale programma in  $P'(Y)$ , dove le computazioni relative a  $X$  sono già state svolte (prima dell’esecuzione) e ottenere così un programma più veloce.

Più formalmente, un valutatore parziale per il linguaggio  $\mathcal{L}$  è un programma che realizza una funzione

$$\mathcal{P}eval_{\mathcal{L}} : (\text{Prog}^{\mathcal{L}} \times \mathcal{D}) \rightarrow \text{Prog}^{\mathcal{L}}$$

che ha le seguenti caratteristiche: dato un generico programma  $P$  scritto in  $\mathcal{L}$  e che accetta due argomenti il risultato della valutazione parziale di  $P$  rispetto ad un suo *primo input*  $D_1$

$$\mathcal{P}eval_{\mathcal{L}}(P, D_1) = P' \quad (1.4)$$

è il programma  $P'$  che accetta *un solo argomento* e tale che, per un qualsiasi dato di input  $Y$ , si ha

$$\mathcal{I}_{\mathcal{L}}(P, (D_1, Y)) = \mathcal{I}_{\mathcal{L}}(P', Y), \quad (1.5)$$

dove  $\mathcal{I}_{\mathcal{L}}$  è l’interprete del linguaggio.

veri e propri nuovi linguaggi specifici per definire i comportamenti dei cosiddetti “business processes” basati su servizi Web (un esempio è il Business Process Execution Language). Infine, all’ultimo livello troviamo una specifica applicazione, nel nostro caso di commercio elettronico, che pur fornendo funzionalità molto specifiche e ristrette può anch’essa essere vista in termini di una ulteriore macchina astratta.

### 1.4 Sommario del capitolo

Il capitolo ha introdotto la nozione di macchina astratta e le principali modalità di implementazione di un linguaggio di programmazione. In particolare abbiamo visto:

- *La macchina astratta*: una formalizzazione astratta di un generico esecutore di algoritmi, formalizzati in termini di uno specifico linguaggio di programmazione.
- *L’interprete*: un componente essenziale della macchina astratta che ne caratterizza il comportamento, mettendo in relazione “operazionale” il linguaggio della macchina astratta col mondo fisico circostante.
- *Il linguaggio macchina*: il linguaggio di una generica macchina astratta.
- *Varie tipologie di linguaggi*: caratterizzate dalla distanza dalla macchina fisica.
- *L’implementazione di un linguaggio*: nelle sue varie modalità, da quella puramente interpretativa a quella puramente compilativa; la nozione di *compilatore* è qui particolarmente importante.
- *La nozione di macchina intermedia*: essenziale nell’implementazione reale di un qualsiasi linguaggio, nota soprattutto in alcuni casi “famosi” (macchina P-code per Pascal e Java Virtual Machine).
- *Gerarchie di macchine astratte*: Le macchine astratte si possono comporre gerarchicamente e molti sistemi software possono essere visti in questi termini.

### 1.5 Nota bibliografica

La nozione di macchina astratta è presente in molti contesti diversi, dai linguaggi di programmazione ai sistemi operativi, anche se a volte è usata in modo più informale di quanto fatto in questo capitolo. In alcuni casi è anche detta macchina virtuale, come ad esempio in [81] che comunque presenta un approccio simile a quello qui seguito.

La descrizione della macchina hardware che abbiamo usato si può trovare in un qualsiasi testo di architettura dei calcolatori, ad esempio [97].

La macchina intermedia è stata introdotta nelle prime implementazioni di Pascal, si veda [77]. Per usi più recenti della macchina intermedia nel contesto delle implementazioni di Java si possono consultare i molti testi sulla JVM, ad esempio [56].

Per quanto riguarda la compilazione infine un testo classico è [4], mentre [8] è un libro più recente con una trattazione più aggiornata.

### 1.6 Esercizi

1. Si forniscano tre esempi, in contesti diversi, di macchine astratte.
2. Si descriva il funzionamento dell’interprete di una generica macchina astratta.
3. Si descrivano le differenze fra implementazione interpretativa ed implementazione compilativa di un linguaggio, evidenziandone vantaggi e svantaggi.
4. Supponendo di avere a disposizione una macchina astratta C già realizzata, come potremmo usarla per realizzare una macchina astratta per un altro linguaggio L?
5. Quali sono i vantaggi nell’uso di una macchina intermedia per l’implementazione di un linguaggio ?

6. I primi ambienti Pascal includevano

- (a) un compilatore per Pascal, scritto in Pascal, che produceva P-code (codice della macchina intermedia);
- (b) lo stesso compilatore, tradotto in P-code;
- (c) un interprete per P-code, scritto in Pascal.

Implementare in modo interpretativo il linguaggio Pascal su una macchina ospite significava tradurre (manualmente) l'interprete per P-code nel linguaggio della macchina ospite. Data una tale implementazione interpretativa, come si potrebbe ottenere un'implementazione compilativa per la stessa macchina ospite minimizzando lo sforzo richiesto? (Suggerimento: si parta da una modifica del compilatore per Pascal scritto in Pascal stesso).

7. Si consideri un interprete  $\mathcal{I}_{\mathcal{L}1}^{\mathcal{L}}(X, Y)$ , scritto nel linguaggio  $\mathcal{L}$ , per un diverso linguaggio  $\mathcal{L}1$  dove  $X$  è il programma da interpretare e  $Y$  sono i dati di input. Si consideri un programma  $P$  scritto in  $\mathcal{L}1$ . Che cosa si ottiene valutando

$$\mathcal{P}eval_{\mathcal{L}}(\mathcal{I}_{\mathcal{L}1}^{\mathcal{L}}, P)$$

ossia dalla valutazione parziale di  $\mathcal{I}_{\mathcal{L}1}^{\mathcal{L}}$  rispetto a  $P$ ? (Questa trasformazione è nota come prima proiezione di Futamura).

## Descrivere un linguaggio di programmazione

Un linguaggio di programmazione è un formalismo artificiale nel quale poter esprimere algoritmi. Per quanto artificiale, questo formalismo è tuttavia pur sempre un *linguaggio*: il suo studio può far tesoro di alcuni concetti e strumenti sviluppati nell'ultimo secolo dalla linguistica (che studia lingue sia naturali sia artificiali). Senza addentrarsi in questioni troppo complesse, questo capitolo si pone il problema di cosa voglia dire “dare” (definire, apprendere) un linguaggio di programmazione e di quali strumenti ci si possa avvalere in quest’impresa.

### 2.1 Livelli di descrizione

In un lavoro ormai classico della linguistica, Morris [72] studia i vari livelli ai quali può avvenire la descrizione in un linguaggio, individuando tre grandi ambiti: quello della *grammatica*, quello della *semantica*, quello della *pragmatica*.

La *grammatica* è quella parte della descrizione di un linguaggio che risponde alla domanda “Quali frasi sono corrette?”. Una volta che sia definito l’alfabeto del linguaggio (nel caso del linguaggio naturale, ad esempio: alfabeto latino su 22 o 26 lettere, alfabeto cirillico ecc.) ad un primo stadio, quello *lessicale*, utilizzando questo alfabeto, sono individuate le sequenze corrette di simboli che costituiscono le *parole* (o *token*) del linguaggio. Definiti alfabeto e parole, la *sintassi* passa a descrivere quali sequenze di parole costituiscano frasi legali. La sintassi è dunque una relazione tra segni: tra tutte le possibili sequenze di parole (su un dato alfabeto), seleziona un sottoinsieme di sequenze che costituiscono le frasi del linguaggio stesso<sup>1</sup>.

<sup>1</sup>Nell’ambito della linguistica, ovviamente, le cose sono più complicate. Oltre al livello lessicale e a quello sintattico si ha anche il livello morfologico, separato dai precedenti, nel quale sono definite le diverse forme che le parole (e le frasi) assumono a seconda della funzione grammaticale: ad esempio nel lessico (ossia nel dizionario, nel “thesaurus” della lingua in questione) troviamo la parola “lupo” con il relativo valore lessicale costituito dall’immagine dell’animale noto a tutti. A livello morfologico invece la parola è scomponibile nella radice “lup-” e nel morfema “-o” che segnala il singolare. Nelle lingue naturali sono presenti anche aspetti fonetici che in questa sede evidentemente non interessano.

La *semantica* è quella parte della descrizione di un linguaggio che cerca di dare risposta alla domanda “Cosa significa una frase corretta?” La semantica, dunque, attribuisce un *significato* ad ogni frase corretta. Se nel caso dei linguaggi naturali questo processo di attribuzione di significato può essere molto complesso, nel caso dei linguaggi artificiali la situazione è assai più semplice. Non è difficile supporre, in questo caso, che la semantica sia una relazione tra segni (le frasi corrette) e significati (entità autonome che esistono indipendentemente dai segni che usiamo per descriverle). Ad esempio, il significato di un certo programma potrebbe essere la funzione matematica calcolata da quel programma; la descrizione semantica di quel linguaggio sarà costituita da quelle tecniche che ci permettono di risalire, dato un programma, alla funzione che quel programma calcola.

È al terzo livello che fa la sua comparsa sulla scena l'attore principale, colui che usa un certo linguaggio. La *pragmatica*, infatti, è quella parte della descrizione di un linguaggio che si chiede “Come usare una frase corretta e sensata?” Frasi con lo stesso significato possono essere usate in modo diverso da utenti diversi. Contesti linguistici diversi possono richiedere l'uso di frasi diverse; alcune sono più eleganti, o più auliche, o più dialettali, di altre. Comprendere questi meccanismi di un linguaggio non è meno importante del conoscerne sintassi e semantica.

Nel caso dei linguaggi di programmazione, a questi tre livelli classici ne possiamo aggiungere un quarto, quello dell'*implementazione*. Posto che i linguaggi che ci interessano sono linguaggi procedurali (cioè linguaggi le cui frasi corrette specificano azioni), rimane da descrivere “Come eseguire una frase corretta, in modo da rispettarne la semantica”. Conoscere la semantica è in genere sufficiente per l'utilizzatore del linguaggio, ma il progettista software (e più ancora il progettista del linguaggio) è interessato anche a comprendere mediante quale processo le frasi “operative” del linguaggio realizzano lo stato di cose di cui parlano. E appunto questo è descritto dall'implementazione di un linguaggio.

Possiamo fare un esempio un po' rudimentale, ma che speriamo serva allo scopo. Consideriamo il linguaggio naturale utilizzato per esprimere ricette di cucina. La sintassi determina le frasi corrette mediante le quali una ricetta è espressa. La semantica si incarica di spiegarci “cosa fa” una ricetta, indipendentemente dalla sua (specifica) esecuzione. La pragmatica studia come un cuoco (“quel cuoco”) interpreta le varie frasi della ricetta. Infine, l'implementazione descrive le modalità (dove, come e con quali ingredienti) mediante le quali la ricetta di cucina si trasforma nel piatto che la sua semantica prescrive.

Vedremo analiticamente nelle prossime sezioni quali forme assumano questi quattro livelli nel contesto dei linguaggi di programmazione.

## 2.2 Grammatica e sintassi

Abbiamo già detto che la grammatica di un linguaggio stabilisce prima alfabeto e lessico e quindi definisce, con la sintassi, quali sequenze di simboli corrispondono a frasi ben formate (o “frasi” *tout court*). Evidentemente, almeno dal punto di vista del linguaggio naturale, la definizione dell’alfabeto (finito) è cosa immedia-

ta. Anche il lessico potrebbe essere definito, almeno in prima approssimazione, in modo semplice: accontentandosi, per il momento, di un insieme di vocaboli finito possiamo semplicemente elencare le parole che ci interessano: questo è certamente il caso dei linguaggi naturali, visto che i dizionari sono volumi finiti<sup>2</sup>!

Ma come viene descritta la sintassi? Nel caso di un linguaggio naturale, è lo stesso linguaggio naturale che viene usato, in genere, per descrivere la propria sintassi (esempio classico sono i testi di grammatica delle varie lingue). Anche la sintassi dei linguaggi di programmazione è spesso descritta usando il linguaggio naturale, soprattutto nel caso dei linguaggi più antichi. Il linguaggio naturale, tuttavia, introduce spesso ambiguità nella descrizione della sintassi e, soprattutto, non è di alcun aiuto nel processo di traduzione (compilazione) di un programma in un altro linguaggio (in genere di più basso livello).

La linguistica, d'altra parte, ha sviluppato sin dagli anni '50 (per opera del linguista americano Noam Chomsky) alcune tecniche per descrivere i fenomeni sintattici in modo formale, cioè usando dei formalismi pensati apposta per limitare l'ambiguità che sempre si rivela nel linguaggio naturale. Queste tecniche, note sotto il nome di grammatiche generative, non sono di grande aiuto nella descrizione sintattica dei linguaggi naturali (troppo complessi e raffinati), ma costituiscono invece uno strumento fondamentale per descrivere la sintassi dei linguaggi di programmazione (e, soprattutto, per la loro compilazione, come vedremo brevemente nel Paragrafo 2.4 e poi, più diffusamente, nei Capitoli 3 e 4).

**Esempio 2.1** Vogliamo descrivere un semplice linguaggio: quello delle stringhe *palindrome* costruite a partire dai simboli *a* e *b*<sup>3</sup>. Iniziamo dunque fissando l’alfabeto  $A = \{a, b\}$ . Dobbiamo ora selezionare, tra tutte le stringhe su  $A$  (cioè le successioni finite di elementi di  $A$ ), quelle stringhe che corrispondono a palindrome. Il modo più semplice per farlo è quello di osservare che c’è una semplice definizione ricorsiva di stringa palindroma. Infatti, possiamo dire (e questa è la base dell’induzione) che *a* e *b* sono ciascuna una stringa palindroma. Se, poi, *s* è una generica stringa che sappiamo già essere palindroma, allora anche *asa* e *bsb* sono entrambe stringhe palindrome (e questo è il passo dell’induzione).

Non è difficile convincersi che questa definizione cattura tutte e sole le stringhe palindrome su  $A$  di lunghezza dispari. Rimangono escluse le stringhe di lunghezza pari, come *abba*: per includere anche quest’ultime, aggiungiamo alla base della nostra definizione induttiva che la stringa vuota (cioè la stringa che non contiene alcun simbolo dell’alfabeto) è anch’essa una stringa palindroma. Adesso la nostra definizione caratterizza tutte e sole le stringhe palindrome sull’alfabeto  $A$ : se una stringa è davvero palindroma (per esempio *aabaa*, oppure *abba*) esiste una successione di applicazioni delle regole induttive appena menzionate che

<sup>2</sup>Nell’ambito dei linguaggi di programmazione il lessico può anche essere costituito da un insieme infinito di parole: vedremo più avanti come questo sia possibile.

<sup>3</sup>Una stringa è palindroma se coincide con la sua stringa riflessa, cioè se la stringa non cambia sia quando letta da sinistra a destra sia quando letta da destra a sinistra; un famoso esempio in latino (tralasciando gli spazi) è l’indovinello “in girum imus nocte et consumimur igni”.

la costruisce. Viceversa, se una stringa non è palindroma (per esempio *aabab*) non esiste alcun modo per costruirla induttivamente con le regole introdotte.

Le grammatiche libere, che introdurremo nel prossimo paragrafo, sono una notazione per esprimere in modo conciso e preciso definizioni ricorsive di stringhe come quella appena vista. La definizione induttiva per le stringhe palindrome può essere espressa in forma di grammatica come

$$\begin{aligned} P &\rightarrow \\ P &\rightarrow a \\ P &\rightarrow b \\ P &\rightarrow aPa \\ P &\rightarrow bPb \end{aligned}$$

In queste regole,  $P$  sta per “una generica stringa palindroma”, mentre la freccia “ $\rightarrow$ ” deve essere letta “può essere”. Si riconoscerà immediatamente la base della definizione induttiva nelle prime tre righe, mentre le ultime due costituiscono il passo induttivo.

## 2.2.1 Grammatiche libere

L'esempio appena riportato è un esempio di *grammatica libera da contesto*, una tecnica fondamentale per la descrizione dei linguaggi di programmazione. Iniziamo introducendo un po' di notazione e terminologia. Fissato un insieme finito (o anche numerabile)  $A$ , che chiamiamo *alfabeto*, indichiamo con  $A^*$  l'insieme di tutte le *stringhe finite su A* (cioè le successioni di lunghezza finita di elementi di  $A$ ; l'operatore  $*$  è detto *stella di Kleene*, vedi a pag. 53 per la definizione formale). Osserviamo subito che, in base alla definizione, appartiene ad  $A^*$  anche la successione di lunghezza zero: si tratta della stringa vuota, che denotiamo con  $\epsilon$ .

Un *linguaggio (formale) sull'alfabeto A* non è altro che un sottoinsieme di  $A^*$ . Una grammatica formale serve proprio a definire un certo sottoinsieme di stringhe tra tutte quelle possibili su un dato alfabeto<sup>4</sup>.

**Definizione 2.2 (Grammatica libera)** *Una grammatica libera da contesto (o anche: non contestuale) è una quadrupla ( $NT, T, R, S$ ) dove*

1.  $NT$  è un insieme finito di simboli (i simboli non terminali, o le variabili, o le categorie sintattiche);
2.  $T$  è un insieme finito di simboli (i simboli terminali);
3.  $R$  è un insieme finito di produzioni (o regole), ciascuna delle quali consiste in espressioni della forma

$$V \rightarrow w$$

<sup>4</sup>Nell'ambito dei linguaggi formali, usualmente una sequenza di simboli terminali che appartiene ad un linguaggio è detta “parola” del linguaggio. Non useremo questa terminologia, parlando invece di stringhe, per evitare ambiguità con le parole e le frasi di un linguaggio (naturale o artificiale), nel significato con cui esse vengono intese abitualmente.

dove  $V$  (la testa della produzione) è un singolo simbolo non terminale e  $w$  (il corpo) è una stringa costituita da zero o più simboli terminali o non terminali (cioè  $w$  è una stringa su  $T \cup NT$ ).

4.  $S$  è un elemento di  $NT$  (il simbolo iniziale);

Seguendo la definizione, quindi, la grammatica dell'Esempio 2.1 è costituita dalla quadrupla  $(\{P\}, \{a, b\}, R, P)$ , dove  $R$  è l'insieme di produzioni già riportate nell'esempio stesso.

Osserviamo che, in base alla definizione, una produzione può avere il corpo vuoto, cioè composto da nessun simbolo, o, più propriamente, costituito dalla *stringa vuota*,  $\epsilon$ . Vedremo in un prossimo paragrafo come una certa grammatica definisca un linguaggio sull'alfabeto costituito dai suoi simboli terminali. Iniziamo intanto con un esempio.

**Esempio 2.3** Vediamo una semplice grammatica che descriva le espressioni aritmetiche (sugli operatori  $+$ ,  $*$ ,  $-$  unario e binario). Gli elementi atomici di queste espressioni sono semplici identificatori costituiti da sequenze finite dei simboli  $a$  e  $b$ .

Definiamo la grammatica  $G = (\{E, I\}, \{a, b, +, *, -, (, )\}, R, E)$ , dove  $R$  è il seguente insieme di produzioni

1.  $E \rightarrow I$
2.  $E \rightarrow E + E$
3.  $E \rightarrow E * E$
4.  $E \rightarrow E - E$
5.  $E \rightarrow -E$
6.  $E \rightarrow (E)$
7.  $I \rightarrow a$
8.  $I \rightarrow b$
9.  $I \rightarrow Ia$
10.  $I \rightarrow Ib$

A differenza della grammatica dell'Esempio 2.1, questa ha più simboli non terminali che corrispondono, in modo informale, ad un'espressione ( $E$ ) e ad un identificatore ( $I$ ). Si osservi, ancora una volta, come le produzioni siano un modo sintetico per esprimere definizioni ricorsive: in questo caso si tratta di due definizioni (che sono state anche graficamente separate tra loro), di cui una (quella per  $E$ ) usa nel caso base il simbolo non terminale definito induttivamente con l'altra definizione ricorsiva (quella per  $I$ ).

**BNF** Nel contesto dei linguaggi di programmazione, le grammatiche libere sono state usate per la prima volta per la definizione del linguaggio ALGOL 60. Nel rapporto che introduce ALGOL 60, la grammatica che definisce il linguaggio viene descritta usando una notazione leggermente diversa da quella che abbiamo appena introdotto. Tale notazione, (pensata, tra l'altro, per poter essere usata con

un insieme di caratteri ridotto, che non include la freccia, i corsivi, le maiuscole ecc.) va sotto il nome di *forma normale di Backus e Naur* (BNF), dal nome di due autorevoli membri del comitato ALGOL, John Backus (che aveva precedentemente diretto il progetto FORTRAN e scritto il compilatore per quel linguaggio, il primo ad alto livello) e Peter Naur. Nella BNF:

- la freccia “ $\rightarrow$ ” è sostituita dai caratteri “ $::=$ ”;
- i simboli non terminali sono scritti tra parentesi angolate (per esempio  $\langle Exp \rangle$  e  $\langle Ide \rangle$  potrebbero essere i due simboli non terminali della grammatica dell’Esempio 2.3);
- produzioni con la stessa testa sono raggruppate in un unico blocco, usando la barra verticale “|” per separare le produzioni; nell’Esempio 2.3, le produzioni per  $E$  potrebbero dunque essere espresse nel modo seguente

$$\langle E \rangle ::= \langle I \rangle | \langle E \rangle + \langle E \rangle | \langle E \rangle * \langle E \rangle | \langle E \rangle - \langle E \rangle | -\langle E \rangle | (\langle E \rangle).$$

Talvolta le notazioni si trovano anche mescolate (per esempio, uso della barra verticale, della freccia, e simboli non terminali scritti come maiuscole corsive).

**Derivazioni e linguaggi** Una grammatica definisce induttivamente un insieme di stringhe. Possiamo esplicitare l’aspetto operativo di tale definizione induttiva mediante la nozione di derivazione.

**Esempio 2.4** Possiamo sincerarci che la stringa  $ab * (a + b)$  è una stringa corretta secondo la grammatica dell’Esempio 2.3 leggendo le produzioni come “regole di riscrittura” e applicandole ripetutamente. Usiamo una freccia più grande ( $\Rightarrow$ ) per denotare quest’operazione di riscrittura tra stringhe. Possiamo allora procedere come segue. Partiamo dal simbolo iniziale  $E$  e riscriviamolo usando una produzione (a nostra scelta); usiamo ad esempio la produzione (3), ottenendo la riscrittura  $E \Rightarrow E * E$ . Adesso concentriamoci sul lato destro di questa produzione: abbiamo due  $E$  che possono essere riscritte (espanso) indipendentemente tra loro. Prendiamo quella più a sinistra ed applichiamo la produzione (1), ottenendo  $E * E \Rightarrow I * E$ . Possiamo ora scegliere se espandere  $I$  o  $E$ , e così via. La Figura 2.1 mostra la riscrittura che abbiamo iniziato, sviluppata fino ad ottenere la stringa  $ab * (a + b)$ ; al pedice di ogni  $\Rightarrow$  è stata indicata la produzione utilizzata.

Possiamo capitalizzare l’esempio appena discusso in una definizione.

**Definizione 2.5 (Derivazione)** Fissata una grammatica  $G = (NT, T, R, S)$ , e assegnate due stringhe  $v, w$  su  $NT \cup T$ , diciamo che da  $v$  si deriva immediatamente  $w$  (o anche:  $v$  si riscrive in un passo in  $w$ ) e scriviamo  $v \Rightarrow w$ , se  $w$  si ottiene da  $v$  sostituendo ad un simbolo non terminale  $V$  presente in  $v$  il corpo di una produzione di  $R$  la cui testa sia  $V$ .

Diciamo che da  $v$  si deriva  $w$  (o anche:  $v$  si riscrive in  $w$ ) e scriviamo  $v \Rightarrow^* w$ , se esiste una sequenza finita (eventualmente vuota) di derivazioni immediate  $v \Rightarrow w_0 \Rightarrow w_1 \Rightarrow \dots \Rightarrow w$ .

$E$	$\Rightarrow_3$	$E * E$
	$\Rightarrow_1$	$I * E$
	$\Rightarrow_{10}$	$Ib * E$
	$\Rightarrow_7$	$ab * E$
	$\Rightarrow_6$	$ab * (E)$
	$\Rightarrow_2$	$ab * (E + E)$
	$\Rightarrow_1$	$ab * (I + E)$
	$\Rightarrow_7$	$ab * (a + E)$
	$\Rightarrow_1$	$ab * (a + I)$
	$\Rightarrow_8$	$ab * (a + b)$

Figura 2.1 Una derivazione per la stringa  $ab * (a + b)$ .

Usando la notazione appena introdotta, e relativamente alla grammatica delle espressioni sin qui utilizzata, possiamo scrivere, ad esempio,  $E * E \Rightarrow^* ab * (a + I)$ . Particolarmenete interessanti sono quelle derivazioni dove a sinistra del simbolo  $\Rightarrow^*$  sta il simbolo iniziale della grammatica, e a destra sta invece una stringa composta da soli *terminali*. Si tratta, in un certo senso, di derivazioni massimali, che non possono essere estese (riscrivendo un non terminale) né a sinistra né a destra. Seguendo l’intuizione che ci ha portato ad introdurre le grammatiche, queste derivazioni sono quelle che ci danno le stringhe corrette secondo la grammatica.

**Definizione 2.6 (Linguaggio generato)** Il linguaggio generato da una grammatica  $G = (NT, T, R, S)$  è l’insieme  $\mathcal{L}[G] = \{w \in T^* \mid S \Rightarrow^* w\}$ .

Si osservi che questa definizione, in accordo con quanto detto all’inizio del Paragrafo 2.2.1, definisce proprio un linguaggio su  $T^*$ .

**Alberi di derivazione** La derivazione di una stringa è un processo rigidamente sequenziale. In esso, vi sono dei passi che devono essere effettuati in un certo ordine: per esempio, nella Figura 2.1 è evidente che la  $\Rightarrow_{10}$  deve seguire la prima  $\Rightarrow_3$ , perché la produzione (10) riscrive il simbolo non terminale  $I$  che non esiste nella stringa iniziale (costituita dal solo simbolo iniziale  $E$ ) e che viene introdotto proprio dalla produzione (3). Ma vi sono alcuni passi che potrebbero anche essere scambiati d’ordine. Nella derivazione della Figura 2.1, per esempio, tutte le volte che si deve riscrivere un simbolo non terminale si sceglie sempre quello più a sinistra. Potremmo immaginare, invece, di concentrarci prima sul simbolo non terminale più a destra, ottenendo la derivazione della Figura 2.2.

Le due derivazioni che abbiamo dato per la stessa stringa  $ab * (a + b)$  sono, in modo molto intuitivo, equivalenti: entrambe ricostruiscono nello stesso modo la struttura della stringa (in termini di simboli non terminali e terminali), mentre differiscono soltanto nell’ordine col quale le produzioni sono applicate. Questo fatto è evidenziato nella Figura 2.3, che rappresenta la derivazione della stringa  $ab * (a + b)$  sotto forma di albero.

## Alberi

La nozione di albero è di notevole importanza in informatica ed è molto usata anche nel linguaggio comune (si pensi all'albero genealogico, ad esempio). Un albero è una struttura informativa che può essere definita in vari modi e che esiste in varie "specie". Ai fini di questa trattazione interessano solo gli alberi radicati ordinati, (o alberi *tout court*) che possiamo definire come segue. Un albero (radicato, ordinato) è un insieme finito di elementi, detti *nodi*, tale che se non è vuoto allora un particolare nodo è detto *radice* ed i rimanenti nodi, se esistono, sono partizionati fra gli elementi della n-upla (ordinata)  $\langle S_1, S_2 \dots, S_n \rangle$ ,  $n \geq 0$ , dove ogni  $S_i$ ,  $i \in [1, n]$ , è un albero.

Intuitivamente quindi un albero permette di raggruppare i nodi in livelli dove al livello 0 avremo la radice, al livello 1 le radici degli alberi  $S_1, S_2 \dots, S_n$ , e così via.

Si usa spesso anche un'altra definizione (equivalente) di albero che probabilmente è più significativa per il lettore che abbia familiarità con alberi genealogici, classificazioni botaniche, tassonomie ecc. Secondo questa definizione un albero è un caso particolare di grafo: un albero (radicato, ordinato) è dunque una coppia  $T = (N, A)$ , dove  $N$  è un insieme finito di *nodi* e  $A$  è un insieme di coppie ordinate di nodi, dette *archi*, tale che:

- il numero di archi è eguale al numero di nodi meno uno;
- $T$  è connesso, ossia per ogni coppia di nodi  $n, m \in N$  esiste una sequenza di nodi distinti  $n_0, n_1, \dots, n_k$  tale che  $n_0 = n$ ,  $n_k = m$  e la coppia  $(n_i, n_{i+1})$  è un arco, per  $i = 0, \dots, k - 1$ ;
- un nodo  $r$  è designato *radice* ed i nodi sono ordinati per livelli usando la seguente definizione induktiva: la radice è a livello 0; se un nodo  $n$  è a livello  $i$  ed esiste l'arco  $(n, m) \in A$  allora il nodo  $m$  è a livello  $i + 1$ ;
- dato un nodo  $n$ , i nodi  $m$  tali che esiste un arco  $(n, m) \in A$  sono detti figli di  $n$  (e  $n$  è detto padre di essi); per ogni nodo  $n \in N$  è stabilito un ordinamento totale sull'insieme di tutti i figli di  $n$ .

Con una curiosa mescolanza di termini botanici, matematici e "familiari", nodi con lo stesso padre sono detti fratelli mentre nodi senza figli sono detti *foglie*. La radice è l'unico nodo senza padre.

**Definizione 2.7** Data una grammatica  $G = (NT, T, R, S)$ , un albero di derivazione (o albero di parsing) è un albero ordinato in cui:

1. ogni nodo è etichettato con un simbolo in  $NT \cup T \cup \{\epsilon\}$ ;
2. la radice è etichettata con  $S$ ;
3. ogni nodo interno è etichettato con un simbolo in  $NT$ ;
4. se un certo nodo ha etichetta  $A \in NT$  e i suoi figli sono  $m_1, \dots, m_k$  etichettati rispettivamente con  $X_1, \dots, X_k$ , dove  $X_i \in NT \cup T$  per ogni  $i \in [1, k]$ , allora  $A \rightarrow X_1 \dots X_k$  è una produzione di  $R$ ;

$E$	$\Rightarrow_3$	$E * E$
	$\Rightarrow_6$	$E * (E)$
	$\Rightarrow_2$	$E * (E + E)$
	$\Rightarrow_1$	$E * (E + I)$
	$\Rightarrow_8$	$E * (E + b)$
	$\Rightarrow_1$	$E * (I + b)$
	$\Rightarrow_7$	$E * (a + b)$
	$\Rightarrow_1$	$I * (a + b)$
	$\Rightarrow_{10}$	$Ib * (a + b)$
	$\Rightarrow_7$	$ab * (a + b)$

Figura 2.2 Un'altra derivazione per la stringa  $ab * (a + b)$ .

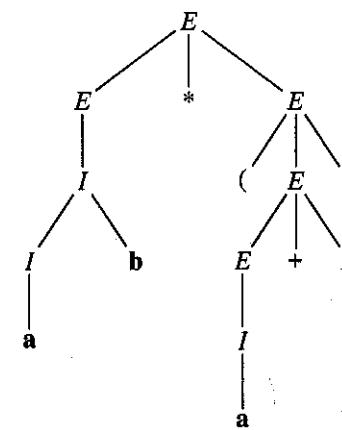
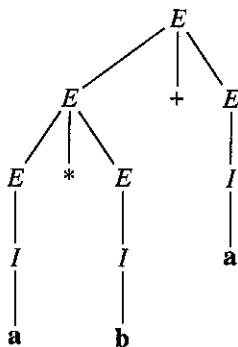


Figura 2.3 Un albero di derivazione per la stringa  $ab * (a + b)$ .

5. se un nodo ha etichetta  $\epsilon$ , allora quel nodo è figlio unico e, detto  $A$  il suo padre,  $A \rightarrow \epsilon$  è una produzione di  $R$ .

È facile rendersi conto che, data una derivazione, possiamo ottenere a partire da essa un albero di derivazione. Basta iniziare dalla radice (etichettata con il simbolo iniziale) ed ogni volta aggiungere un livello di figli in corrispondenza alla produzione utilizzata durante la derivazione. Se applichiamo questo procedimento alla derivazione della Figura 2.1, otteniamo l'albero di derivazione della Figura 2.3. Applichiamo ora questo procedimento di costruzione dell'albero alla derivazione della Figura 2.2: si ottiene ancora una volta l'albero di Figura 2.3. Le due derivazioni danno luogo allo stesso albero, il che corrisponde all'intuizione che le due derivazioni fossero sostanzialmente equivalenti.



**Figura 2.4** Un albero di derivazione per la stringa  $a * b + a$ .

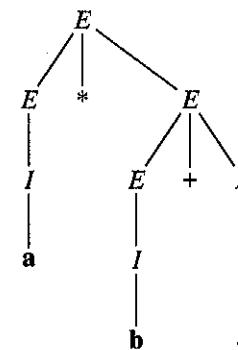
Gli alberi di derivazione sono uno degli strumenti più importanti dell’analisi sintattica di un linguaggio di programmazione. La struttura dell’albero di derivazione infatti esprime, attraverso i sottoalberi, la struttura logica che la grammatica assegna alla stringa. Ad esempio, l’albero della Figura 2.4 corrisponde alla seguente derivazione

$$\begin{aligned}
 E &\Rightarrow_2 E + E \\
 &\Rightarrow_3 E * E + E \\
 &\Rightarrow_1 I * E + E \\
 &\Rightarrow_7 a * E + E \\
 &\Rightarrow_1 a * I + E \\
 &\Rightarrow_8 a * b + E \\
 &\Rightarrow_1 a * b + I \\
 &\Rightarrow_7 a * b + a
 \end{aligned}$$

e nell’albero di Figura 2.4 vediamo che la sottoespressione  $a * b$  appare come figlia sinistra di una somma: questo esprime il fatto che la stringa  $a * b + a$  deve essere interpretata come “moltiplica prima  $a$  per  $b$  e somma poi il risultato ad  $a$ ”, dato che la somma, per poter essere calcolata, ha bisogno dell’operando presente nel sottoalbero di sinistra.

**Ambiguità** Consideriamo la seguente derivazione (sempre nella grammatica dell’Esempio 2.3).

$$\begin{aligned}
 E &\Rightarrow_3 E * E \\
 &\Rightarrow_1 I * E \\
 &\Rightarrow_7 a * E \\
 &\Rightarrow_2 a * E + E \\
 &\Rightarrow_1 a * I + E \\
 &\Rightarrow_8 a * b + E \\
 &\Rightarrow_1 a * b + I \\
 &\Rightarrow_7 a * b + a
 \end{aligned}$$



**Figura 2.5** Un altro albero di derivazione per la stringa  $a * b + a$ .

Se costruiamo l’albero di derivazione corrispondente otteniamo l’albero della Figura 2.5. Confrontando gli alberi delle Figure 2.4 e 2.5, notiamo che abbiamo due alberi diversi che producono la stessa stringa: leggendo da sinistra a destra le foglie dei due alberi otteniamo in entrambi i casi la stringa  $a * b + a$ . Osserviamo però che i due alberi sono radicalmente diversi: il secondo assegna alla stessa stringa una struttura ben diversa (e dunque una diversa precedenza implicita tra gli operatori aritmetici). Se vogliamo usare gli alberi di derivazione per descrivere la struttura logica di una stringa siamo in una pessima situazione: la grammatica dell’Esempio 2.3 non è in grado di assegnare una struttura univoca alla stringa in questione. A seconda di come è costruita la derivazione, la precedenza tra i due operatori aritmetici varia. Vediamo ora una definizione formale di questi concetti.

Innanzitutto chiariamo cosa vuol dire leggere, o visitare, “da sinistra a destra” le foglie di un albero. La visita di un albero consiste nel seguire un percorso fra i nodi dell’albero che permetta di esaminare ogni nodo esattamente una volta. Vi sono vari ordini di visita (anticipato, differito o simmetrico) che se applicati ad uno stesso albero producono sequenze di nodi diverse come risultato della visita. Se si considerano però solo le foglie tutti e tre questi metodi producono lo stesso risultato per cui possiamo usare la seguente definizione.

**Definizione 2.8** Sia  $T = (N, A)$  un albero ordinato non vuoto di radice  $r$ . Il risultato della visita da sinistra a destra di  $T$  è la sequenza di nodi (foglie) ottenuta dalla seguente definizione ricorsiva.

- se  $r$  non ha figli allora il risultato della visita è il nodo  $r$ ;
- se  $r$  ha  $k$  figli  $m_1, m_2, \dots, m_k$  siano  $T_1, T_2, \dots, T_k$  i sottoalberi di  $T$  tali che  $T_i$  ha come radice  $m_i$  ( $T_i$  contiene quindi  $m_i$  e tutta la parte di albero di  $T$  che “sta sotto” tale nodo). Il risultato della visita è la sequenza di foglie ottenuta visitando da sinistra a destra prima  $T_1$  poi  $T_2$  e così via fino a  $T_k$ .

A questo punto possiamo dire cosa significa per una stringa ammettere un albero di derivazione.

$$G' = (\{E, T, A, I\}, \{a, b, +, *, -, (, )\}, R', E)$$

$$\begin{array}{lcl} E & \rightarrow & T \mid T + E \mid T - E \\ T & \rightarrow & A \mid A * T \\ A & \rightarrow & I \mid -A \mid (E) \\ I & \rightarrow & a \mid b \mid Ia \mid Ib \end{array}$$

**Figura 2.6** Una grammatica non ambigua per il linguaggio delle espressioni.

**Definizione 2.9** Diciamo che una stringa di caratteri  $s$  ammette un albero di derivazione  $T$  se  $s$  è il risultato della visita da sinistra a destra di  $T$ .

Finalmente possiamo dare la definizione che ci interessa.

**Definizione 2.10 (Ambiguità)** Una grammatica  $G$  è ambigua se esiste almeno una stringa di  $L[G]$  che ammette più di un albero di derivazione.

Si osservi che l’ambiguità nasce non dall’esistenza di più derivazioni per una stessa stringa (un fatto comune e innocuo), ma dal fatto che (almeno) una stringa ha più alberi di derivazione.

Una grammatica ambigua è inutile come descrizione di un linguaggio di programmazione, perché non può essere usata per tradurre (compilare) in modo univoco un programma. Fortunatamente, data una grammatica ambigua è spesso possibile modificarla, ottenendo un’altra grammatica non ambigua e che genera lo stesso linguaggio<sup>5</sup>. Le tecniche di disambiguazione di una grammatica esulano dagli argomenti di questo testo. A titolo d’esempio, la Figura 2.6 riporta una grammatica non ambigua il cui linguaggio generato coincide con quello della grammatica dell’Esempio 2.3.

Si tratta di una grammatica che interpreta la struttura di un’espressione secondo l’usuale nozione di precedenza tra operatori aritmetici: il – unario ha la precedenza più alta, seguito da \*, seguito a sua volta da + e – binario (tra loro sullo stesso livello quanto a precedenza). La grammatica interpreta poi una sequenza di operatori allo stesso livello di precedenza associando a destra: per esempio, alla stringa  $a + b + a$  viene data la stessa struttura della stringa  $a + (b + a)$ . La non ambiguità viene pagata in termini di complessità: vi sono più non terminali e la comprensione intuitiva della grammatica risulta più difficoltosa.

La necessità di dare grammatiche non ambigue spiega i contorcimenti contenuti nella definizione di alcuni linguaggi di programmazione. Ad esempio, la Figura 2.7 riporta un estratto della grammatica ufficiale di Java relativa ai comandi

<sup>5</sup>Esistono casi patologici di linguaggi che sono generati solo da grammatiche ambigue. Si tratta di linguaggi di nessuna rilevanza nel contesto dei linguaggi di programmazione.

---

```

Statement ::= ... | IfThenStatement | IfThenElseStatement |
StatementWithoutTrailingSubstatement
StatementWithoutTrailingSubstatement ::= ... | Block | EmptyStatement |
ReturnStatement
StatementNoShortIf ::= ... | StatementWithoutTrailingSubstatement |
IfThenElseStatementNoShortIf
IfThenStatement ::= if ( Expression ) Statement
IfThenElseStatement ::= if ( Expression ) StatementNoShortIf else Statement
IfThenElseStatementNoShortIf ::= if ( Expression ) StatementNoShortIf else StatementNoShortIf

```

---

**Figura 2.7** Grammatica di Java per i comandi condizionali.

di condizionali (`if`) (i simboli non terminali sono in corsivo; i simboli terminali di questo estratto sono `if`, `else` e le parentesi).

Questa grammatica è interessante per due motivi. Innanzitutto si noti che quelli che formalmente nelle grammatiche libere sono considerati singoli simboli (terminali o non terminali), qui sono rappresentati da parole costituite da più caratteri. Ad esempio, `if` rappresenta un singolo simbolo terminale e, analogamente, `IfThenElseStatement` rappresenta un simbolo non terminale.

Questo avviene perchè nel nostro linguaggio di programmazione, Java nel caso specifico (ma il discorso sarebbe analogo per un qualsiasi altro linguaggio), si preferisce usare parole significative (`if`, `then`, `else`) che possono, entro certi limiti, suggerire un significato intuitivo (derivante dalla lingua inglese, naturalmente) piuttosto che simboli che, anche se equivalenti per la macchina, risulterebbero di più difficile uso all’utente del linguaggio. In altri termini, come vedremo meglio in un prossimo capitolo, l’uso di nomi simbolici (significativi) facilita sicuramente l’attività del programmatore. Analogamente, per i simboli non terminali, ai fini della comprensione della grammatica è sicuramente meglio usare nomi quali `Statement`<sup>6</sup> che non singoli simboli.

Il secondo aspetto interessante di questa grammatica è legato direttamente alla sua natura involuta, tutt’altro che di immediata comprensione. Questa complicazione è necessaria per risolvere l’ambiguità di stringhe (programmi) come quelle date dal seguente scheletro di programma:

```

if (espressione1) if (espressione2) comando1;
else comando2;

```

Java, come moltissimi altri linguaggi di programmazione, permette comandi condizionali sia con un ramo `else` che senza di esso. Quando un comando `if` senza `else` viene combinato con un `if` con ramo `else` (come nel caso del programma appena esposto), si tratta di determinare a quale dei due `if` si debba accoppiare l’unico `else`. La grammatica della Figura 2.7 è una grammatica

<sup>6</sup>“Comando”.

non ambigua che fa in modo che il ramo `else` “appartenga all’`if` più interno tra tutti quelli ai quali potrebbe appartenere” (dalla definizione di Java [41]). In parole semplici, l’`else` viene accoppiato con il secondo `if` (quello che verifica `espressione2`). Intuitivamente questo avviene perché, una volta che sia stato introdotto un comando `if` con ramo `else`, usando il simbolo non terminale *IfThenElseStatement*, tale comando nel ramo `then` non potrà contenere un comando `if` senza `else`, come risulta dall’ispezione delle regole che definiscono il simbolo non terminale *StatementNoShortIf*.

### 2.3 Vincoli sintattici contestuali

La correttezza sintattica di una frase di un linguaggio di programmazione dipende talvolta dal *contesto* nel quale la frase si trova. Consideriamo, ad esempio, l’assegnamento `I = R+3;`. Supponendo che le produzioni della grammatica in gioco permettano di derivare correttamente questa stringa, essa potrebbe essere tuttavia scorretta nel punto del programma nel quale si trova. Per esempio, se il linguaggio richiede la dichiarazione preventiva delle variabili, occorre che il programma, prima del nostro assegnamento, riporti dichiarazioni per `I` e `R`. Se, poi, il linguaggio è tipizzato, potrebbe non accettare l’assegnamento se il tipo dell’espressione `R+3` non è compatibile<sup>7</sup> con quello della variabile `I`.

Stringhe corrette secondo la grammatica, dunque, possono essere legali solo *in un determinato contesto*. Vi sono molti esempi di questi vincoli sintattici contestuali:

- un identificatore deve essere dichiarato prima dell’uso (Pascal, Java);
- il numero di parametri attuali di una funzione deve essere lo stesso di quello dei parametri formali (C, Pascal, Java ecc.);
- nel caso di un assegnamento, il tipo di un’espressione deve essere compatibile con quello della variabile a cui si assegna (C, Pascal, Java ecc.);
- non sono ammessi assegnamenti alla variabile di controllo di un’iterazione determinata `for` (Pascal);
- prima di usare una variabile, deve esserci stato un assegnamento su di essa (Java);
- un metodo può essere ridefinito (*overridden*) solo da un metodo con la stessa segnatura (Java) o con segnatura compatibile (Java 5);
- ...

Si tratta, con tutta evidenza, di vincoli sintattici; tuttavia la natura contestuale rende impossibile una loro descrizione mediante una grammatica libera da contesto (il nome di questa classe di grammatiche non è stato scelto a caso...). È compito di un testo di linguaggi formali approfondire la questione; ci basti qui

<sup>7</sup>Tutte queste nozioni saranno introdotte a tempo debito: per ora ci serve soltanto un’idea intuitiva dei fenomeni in gioco.

```
int A, B;
read(A);
B = 10/A;
```

Figura 2.8 Un programma che può causare una divisione per zero.

osservare che esistono altri tipi di grammatiche, dette appunto grammatiche contestuali<sup>8</sup>, che permettono di descrivere queste situazioni. Si tratta, tuttavia, di grammatiche pesanti da scrivere, da gestire e, soprattutto, per le quali non esistono tecniche automatiche di generazione di traduttori efficienti quali esistono, invece, per le grammatiche libere. Questa situazione suggerisce, dunque, di limitare l’uso delle grammatiche alla descrizione della sintassi non contestuale, e di esprimere poi gli ulteriori vincoli contestuali mediante il linguaggio naturale, o mediante strumenti formali quali i sistemi di transizione, che vedremo nel Paragrafo 2.5, dedicato alla semantica.

Per quanto i vincoli contestuali facciano certo parte degli aspetti sintattico-grammaticali di un linguaggio, nel gergo dei linguaggi di programmazione ci si riferisce ad essi come a vincoli *di semantica statica*. Nel gergo, “sintassi” significa in genere “descrivibile con una grammatica libera”, “semantica statica” significa “descrivibile con vincoli contestuali verificabili staticamente sul testo del programma”, mentre “semantica dinamica” (o “semantica tout court”) si riferisce a quanto attiene a come e quando il programma sarà eseguito.

La distinzione tra sintassi non contestuale e sintassi contestuale (cioè semantica statica) è stabilita precisamente dalla potenza espressiva delle grammatiche libere. Non sempre è altrettanto netta la distinzione tra semantica statica e semantica dinamica. Alcuni casi sono palesi. Supponiamo, per esempio, di avere un linguaggio la cui definizione stabilisca che nel caso si verifichi una divisione per zero durante l’esecuzione, la macchina astratta del linguaggio *deve* segnalare un esplicito errore. Si tratta evidentemente di un vincolo di semantica dinamica. Ma supponiamo, ora, che la definizione di un altro linguaggio riporti il seguente vincolo:

È sintatticamente scorretto un programma nel quale *si possa* verificare una divisione per zero.

Il programma della Figura 2.8 sarebbe sintatticamente scorretto in tale ipotetico linguaggio, perché esiste una sequenza di esecuzione (quella in cui mediante il comando `read` si assegna il valore 0 ad `A`) che provoca una divisione per zero.

<sup>8</sup>In estrema sintesi, in una grammatica contestuale una produzione può assumere la forma (più generale di quella delle grammatiche libere)  $uAv \rightarrow uwv$ , dove  $u, v$  e  $w$  sono stringhe su  $T \cup NT$ ; in base a tale produzione, il simbolo non terminale  $A$  può essere riscritto in  $w$  solo se appare in un certo contesto, quello dato da  $u$  e  $v$ .

### Sintassi astratta e sintassi concreta

La grammatica di un linguaggio di programmazione definisce un linguaggio come insieme di stringhe. A questo insieme di stringhe corrisponde in modo naturale l'insieme degli alberi di derivazione per esse. Tali alberi sono molto più interessanti delle stringhe di partenza. Innanzitutto essi astraggono dalle specifiche caratteristiche lessicali dei token. Potrà anche accadere che strutture lessicalmente diverse in due linguaggi diversi, possano risultare nello stesso albero: la Figura 2.9 potrebbe essere l'albero corrispondente sia alla stringa Pascal

```
if A=0 then X:=0 else X:=1
```

sia alla stringa Java

```
if (A==0) X=0; else X=1;
```

Come abbiamo già ripetutamente osservato, poi, gli alberi di derivazione sono interessanti in quanto esprimono la struttura canonica assegnata alla stringa.

Non tutti gli alberi di derivazione corrispondono a programmi legali: sappiamo che l'analisi di semantica statica ha il compito di selezionarne alcuni, quelli che soddisfano i vincoli contestuali del linguaggio stesso. L'insieme degli alberi che risultano da questo processo costituisce la *sintassi astratta* di un linguaggio. Si tratta di un modo molto conveniente di pensare ad un linguaggio se siamo interessati alla sua manipolazione (e non meramente a scrivere programmi corretti in esso).

Il vincolo che abbiamo espresso sopra (che certo non è esprimibile mediante una grammatica libera) è un vincolo di semantica statica o dinamica? È chiaro che fa riferimento ad un evento dinamico (la divisione per zero) ma il vincolo, per essere significativo (cioè per escludere davvero alcuni programmi) dovrebbe essere verificato staticamente (come abbiamo fatto noi per il semplice programma mostrato sopra: abbiamo deciso che è sintatticamente scorretto senza eseguirlo, ma solo guardando il codice e ragionando su di esso). Più importante della sua classificazione, è capire se si tratti di un vincolo verificabile o meno. Ovvero: è davvero possibile implementare una macchina astratta per un tale linguaggio, oppure la natura stessa del vincolo implica che non esiste alcuna macchina che sia in grado di verificarlo in modo statico, per un generico programma?

Si tratta di domande di non immediata risposta, ma molto importanti, perché riguardano l'essenza stessa di cosa si possa (o non si possa) fare con una macchina astratta. Dedicheremo a queste domande l'intero prossimo Capitolo 5.

## 2.4 Compilatori

È venuto il momento di vedere a grandi linee come la descrizione sintattica di un linguaggio possa essere utilizzata per tradurre automaticamente un programma. Sappiamo dal Capitolo 1 che un tale traduttore automatico si chiama *compilatore*,

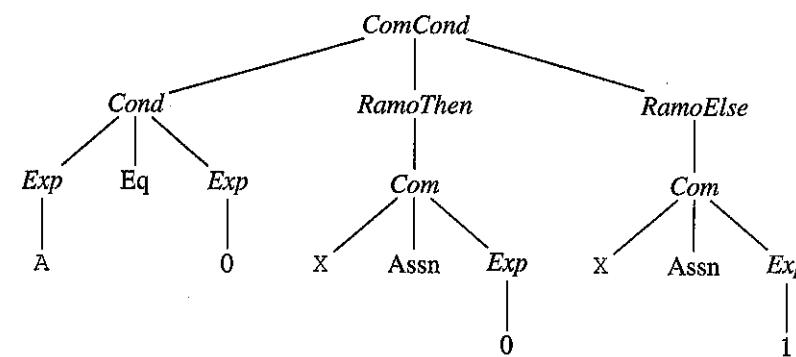


Figura 2.9 Un albero di sintassi astratta.

di cui la Figura 2.10 descrive la struttura logica generale, caratterizzata da una serie di *fasi* in cascata. Le varie fasi, partendo da una stringa che rappresenta il programma nel linguaggio *sorgente*, generano varie rappresentazioni intermedie interne, fino a generare una stringa nel linguaggio *oggetto*. Si osservi che, come visto nel capitolo precedente, in questo contesto "linguaggio oggetto" non significa necessariamente "codice macchina", e neppure "linguaggio di basso livello": si tratta semplicemente del linguaggio verso il quale avviene la traduzione. Nel seguito di questo capitolo descriveremo brevemente le diverse fasi della compilazione per dare un'idea del processo generale. Le prime due fasi (l'analisi lessicale e quella sintattica) saranno oggetto, rispettivamente, dei Capitoli 3 e 4.

**Analisi lessicale** Scopo dell'analisi lessicale è quello di leggere sequenzialmente i simboli (caratteri) di ingresso di cui è composto il programma e di raggruppare tali simboli in unità logicamente significative, che chiamiamo *token* e che sono l'analogo, per il nostro linguaggio di programmazione, delle parole del dizionario di un linguaggio naturale. Ad esempio, l'analizzatore lessicale di C o Java davanti alla stringa `x = 1 + foo++;` restituirà sette token: l'identificatore `x`, l'operatore di assegnamento `=`, il numero `1`, l'operatore di somma `+`, l'identificatore `foo`, l'operatore di autoincremento `++`, l'operatore di fine comando `;`. Token particolarmente significativi sono le parole riservate del linguaggio (come `for`, `if`, `else` ecc.), gli operatori, le parentesi aperte e chiuse (quali `{` e `}` in C o Java, ma anche quali `begin` ed `end` in Pascal). L'analisi lessicale (in inglese chiamata anche *scanning*, *scansione*) è un'operazione molto semplice, che scandisce il testo del programma sorgente da sinistra a destra, in una sola passata, cercando di riconoscere i token. Nessun controllo è ancora fatto sulla sequenza di token, quale, ad esempio, il fatto che le parentesi siano correttamente bilanciate. Come discusso in dettaglio nell'apposito riquadro, lo strumento tecnico che si usa per l'analisi lessicale è riconducibile ad una particolare classe di grammatiche generative (le grammatiche regolari). L'uso di una grammatica per descrivere gli

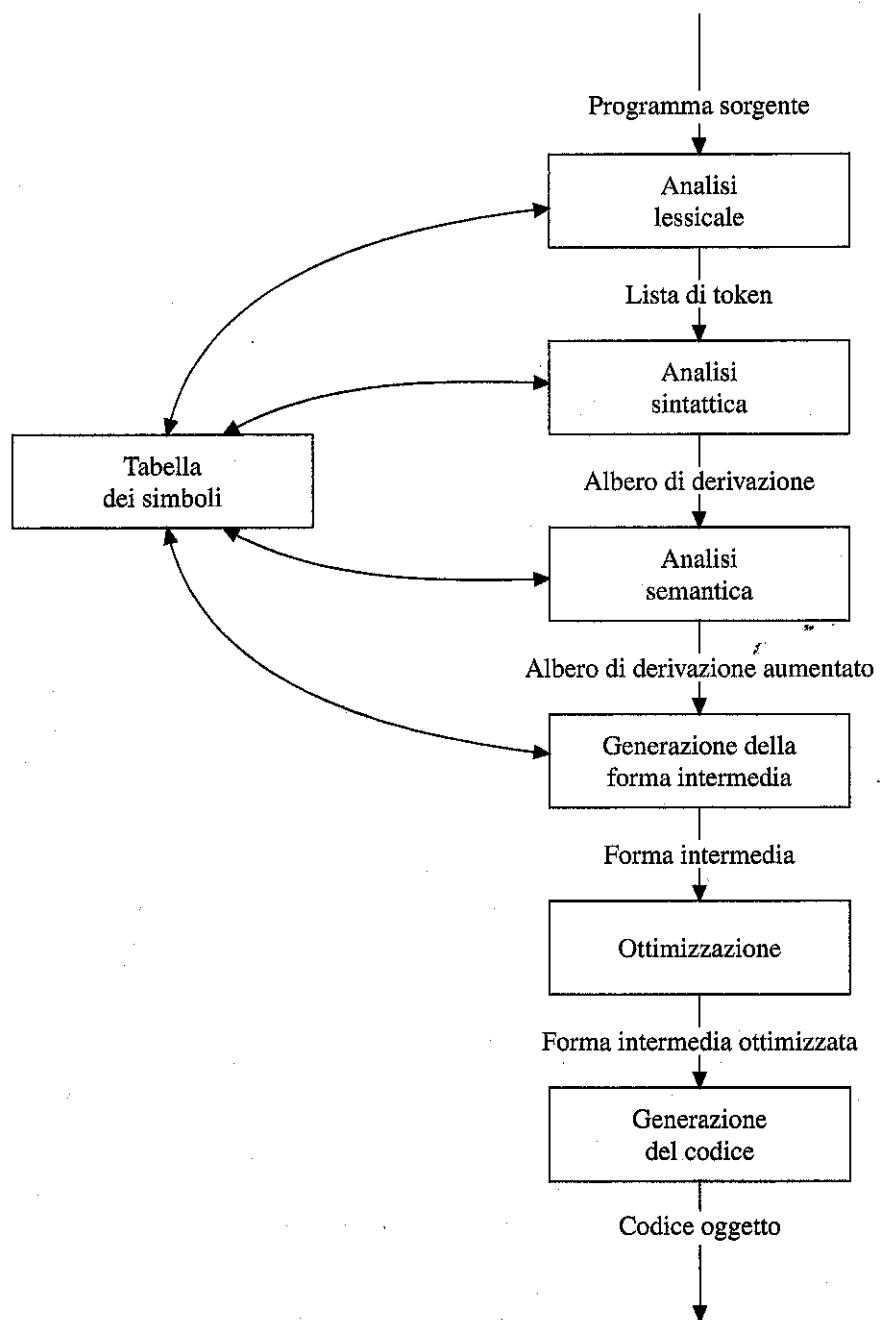


Figura 2.10 Struttura di un compilatore.

### Grammatiche regolari

La differenza tra analisi lessicale e analisi sintattica può essere resa precisa usando una classificazione formale delle grammatiche. Se una grammatica ha tutte le produzioni nella forma  $A \rightarrow bB$  (oppure tutte nella forma  $A \rightarrow Bb$ ), dove  $A$  e  $B$  sono simboli non terminali ( $B$  può anche essere assente, o coincidere con  $A$ ) e  $b$  un singolo terminale, la grammatica si dice *regolare*. Nell'Esempio 2.3, la (sotto-) grammatica relativa al simbolo non terminale  $I$  è una grammatica regolare (mentre non lo è quella per  $E$ ).

Tecnicamente, l'analisi lessicale è quella prima fase della traduzione che decomponе la stringa in ingresso in token, ciascuno dei quali è descritto da una grammatica regolare (nell'Esempio 2.3 l'analisi lessicale avrebbe riconosciuto le sequenze di **a** e **b** quali istanze del simbolo non terminale  $I$ ).

Una volta ottenuta la sequenza di token, l'analisi sintattica avviene su una grammatica propriamente libera da contesto che usa come simboli terminali i token prodotti dalla precedente analisi lessicale. Tratteremo di queste cose, e del potere espressivo delle grammatiche regolari, nel Capitolo 3.

elementi del lessico è dovuto sia alla necessità di riconoscere in modo efficiente tali elementi, sia al fatto che, a differenza di quanto avviene nel caso del lessico delle lingue naturali, per i linguaggi di programmazione il lessico può essere costituito da un insieme infinito di parole e quindi un semplice elenco, quale quello che si ha in un normale dizionario, non è sufficiente (si pensi, ad esempio, alla possibilità di definire gli identificatori come sequenze di caratteri di lunghezza arbitraria che inizino con una lettera).

**Analisi sintattica** Costruita la lista di token, l'analizzatore sintattico (o *parser*) cerca di costruire un albero di derivazione per tale lista. Si tratterà, ovviamente, di un albero di derivazione nella grammatica del linguaggio. Ogni foglia di tale albero deve corrispondere ad un token contenuto nella lista ottenuta dall'analisi lessicale. Inoltre tali foglie, lette da sinistra a destra, devono costituire una frase (ossia una sequenza di simboli terminali) corretta del linguaggio. Sappiamo già che tale albero rappresenta la struttura logica del programma, che sarà sfruttata dalle fasi successive della compilazione.

Potrà accadere che il parser non sia in grado di costruire un albero di derivazione. Questo accade quando la stringa in ingresso non è una stringa corretta secondo la grammatica del linguaggio. In tal caso l'analizzatore sintattico riporta gli errori riscontrati e la compilazione viene abortita.

**Analisi semantica** L'albero di derivazione (che testimonia la correttezza sintattica della stringa di partenza) viene sottoposto ai controlli relativi ai vincoli contestuali del linguaggio. Come abbiamo visto, è a questo stadio che vengono controllate le dichiarazioni, i tipi, il numero dei parametri delle funzioni ecc. Via via che questi controlli vengono effettuati, l'albero di derivazione viene "aumen-

tato” con la relativa informazione, e nuove strutture dati vengono generate. A titolo d'esempio, ad ogni token che corrisponde ad un identificatore di variabile sarà associato il tipo, il luogo di dichiarazione ed altre informazioni utili (per esempio il suo *scope*, vedi Capitolo 6). Per evitare che queste informazioni vengano inutilmente duplicate, esse vengono in genere raccolte in strutture dati esterne all'albero di derivazione. Tra tali strutture, quella che raccoglie le informazioni sugli identificatori è detta tabella dei simboli (cioè degli identificatori) e svolge un ruolo essenziale nelle successive fasi.

A costo di sembrar noiosi, ricordiamo che il nome di analisi semantica è un retaggio storico e che si tratta sempre di vincoli sintattici (contenuti), da non confondere con quello che chiameremo semantica nel Paragrafo 2.5.

**Generazione della forma intermedia** Un'opportuna visita dell'albero di derivazione aumentato permette una prima generazione di codice. Non è ancora opportuno generare codice nel linguaggio oggetto, dato che rimangono da fare molte ottimizzazioni che sono indipendenti dallo specifico linguaggio oggetto. Inoltre, spesso un compilatore viene realizzato per generare codice non in un solo linguaggio oggetto, ma per una serie di linguaggi (per esempio codice macchina di varie architetture diverse). È opportuno, pertanto, concentrare le scelte legate allo specifico linguaggio in un'unica fase (quella, successiva, della generazione del codice) e generare qui codice in una forma intermedia, scelta in modo da essere indipendente sia dal linguaggio sorgente che da quello oggetto.

**Ottimizzazione del codice** Il codice che si ottiene dalla fase precedente mediante visita dell'albero di derivazione, è un codice assai inefficiente. Vi sono molte ottimizzazioni che possono essere condotte prima di generare codice oggetto. Tipiche operazioni che possono essere effettuate sono:

- rimozione del codice inutile (*dead code*), cioè porzioni di codice che non saranno mai eseguite perché non vi sono sequenze d'esecuzione che le raggiungono;
- espansione *in-line* delle chiamate di funzione: alcune chiamate di funzione (procedura) possono essere sostituite con il corpo della relativa funzione, rendendo più efficiente l'esecuzione, rimuovendo un trasferimento del controllo e, quindi, rendendo possibili anche altre ottimizzazioni;
- fattorizzazione delle sottoespressioni: alcuni programmi calcolano più volte lo stesso valore. Se e quando questo fatto è scoperto dal compilatore, il valore dell'espressione “comune” può essere calcolato una sola volta e memorizzato;
- ottimizzazione dei cicli: le iterazioni sono i frammenti dove sono possibili le maggiori ottimizzazioni; tra queste, la più comune consiste nel rimuovere dall'interno del ciclo il calcolo di quelle sottoespressioni il cui valore rimane costante durante le varie iterazioni.

**Generazione del codice** A partire dalla forma intermedia ottimizzata, viene generato il codice oggetto finale. Segue, in genere, un'ulteriore fase di ottimizzazione, dipendente dalle specifiche caratteristiche del linguaggio oggetto. In un

### Rapporto tra parser e scanner

In un compilatore reale l'analizzatore lessicale e quello sintattico non lavorano in sequenza, con due passate distinte. Come vedremo nel Capitolo 4, comunemente il parser cerca di costruire un albero di derivazione analizzando il proprio input un simbolo (cioè, dal suo punto di vista: un token) alla volta. Il controllo del processo viene dunque mantenuto dall'analizzatore sintattico, che richiede all'analizzatore lessicale un nuovo token ogni volta che ha necessità di procedere in avanti. Lo scanner accede all'input vero e proprio, determina un token, e lo restituisce al parser. In questo modo si riducono le operazioni di I/O e la necessità di mantenere in memoria versioni diverse (come stringa, come lista di token, come albero di derivazione) dello stesso programma.

compilatore che generi codice macchina, una parte importante di quest'ultima fase è l'assegnamento dei registri: quali variabili è conveniente tenere nei registri del processore? Si tratta di una scelta di grande importanza per l'efficienza del programma finale.

## 2.5 Semantica

Come anche intuitivamente ci si aspetta, la descrizione della semantica di un linguaggio di programmazione è assai più complessa della sua trattazione sintattica. La complessità nasce sia da questioni tecniche, sia dall'esigenza di mediare tra due istanze contrapposte: la ricerca dell'esattezza, da una parte, e quella della flessibilità, dall'altra. Quanto all'esattezza, è necessaria una descrizione precisa e non ambigua di cosa ci si debba aspettare da ogni costrutto sintatticamente corretto, di modo che ogni utente conosca a priori (cioè prima dell'esecuzione del programma) e indipendentemente dall'architettura, quello che succederà durante l'esecuzione. Questa ricerca di esattezza, tuttavia, non deve impedire che dello stesso linguaggio esistano molteplici realizzazioni (implementazioni), tutte corrette rispetto alla semantica del linguaggio. La specifica semantica deve dunque essere anche flessibile, cioè non deve anticipare scelte che possono invece essere demandate al momento dell'implementazione (e che dunque non fanno parte della definizione del linguaggio stesso).

Non sarebbe difficile raggiungere l'esattezza a danno della flessibilità: basta dare la semantica di un linguaggio mediante uno specifico compilatore su una specifica architettura. Il significato ufficiale di un programma è dato dalla sua esecuzione su quella architettura dopo la traduzione con quel compilatore. A parte la difficoltà di parlare in generale della semantica dei costrutti del linguaggio, questa soluzione ha flessibilità nulla: per come abbiamo costruito questa (pretesa) semantica, esiste una sola implementazione (quella ufficiale); tutte le altre sono in tutto e per tutto equivalenti. Ma fino a quale livello di dettaglio, l'implementazione canonica è “normativa”? Il tempo di calcolo di un certo programma fa parte della sua definizione? E la messaggistica di errore? E come adattare su

architetture diverse il funzionamento dei comandi di ingresso/uscita, tipicamente assai dipendenti dalla specifica architettura? E quando la tecnologia di realizzazione della macchina fisica cambia, come si adatta la semantica, che abbiamo incutamente definito in termini di una specifica macchina?

Una delle difficoltà della definizione semantica è proprio quella di trovare il “giusto mezzo” tra esattezza e flessibilità, in modo da rimuovere ambiguità, ma anche da lasciare spazio all’implementazione. Questa situazione suggerisce l’utilizzo di metodi formali per la descrizione semantica. Metodi del genere esistono da lungo tempo per i linguaggi artificiali della logica matematica, e sono stati adattati dagli informatici per i loro scopi. Tuttavia, alcuni fenomeni semantici rendono spesso la descrizione formale complessa, e non facilmente utilizzabile da un utente che non abbia una formazione specifica. È per questo che la maggioranza delle definizioni ufficiali dei linguaggi di programmazione usa il linguaggio naturale per la descrizione della semantica. Ciò non toglie che metodi formali per la semantica siano molto utilizzati nelle fasi preparatorie del progetto di un linguaggio, oppure per descriverne alcune caratteristiche particolari, dove la necessità di evitare ambiguità è predominante rispetto alla semplicità.

I metodi formali per la semantica si dividono in due grandi famiglie: le semantiche *denotazionali* e quelle *operazionali*<sup>9</sup>. La semantica denotazionale è l’applicazione ai linguaggi di programmazione di tecniche sviluppate per la semantica del linguaggio logico-matematico. Il significato di un programma è dato da una funzione, che esprime il comportamento input/output del programma stesso. Dominio e codominio di questa funzione sono opportune strutture matematiche che corrispondono non solo ai dati di ingresso/uscita, ma anche ad alcune strutture interne del linguaggio, quali l’ambiente e la memoria. Raffinate tecniche matematiche (continuità, punti fissi ecc.) permettono di trattare agevolmente fenomeni quali l’iterazione e la ricorsione (anche tra tipi) (si veda [106]).

Nell’approccio operazionale, invece, non vi sono entità esterne (per esempio funzioni) da associare ai costrutti di un linguaggio. Usando tecniche opportune, una semantica operazionale specifica il comportamento della macchina astratta, ossia ne definisce formalmente l’interprete, facendo riferimento ad un formalismo (astratto) di più basso livello. Le varie tecniche operazionali differiscono (anche profondamente) nella scelta di tale formalismo; alcune semantiche fanno riferimento ad automi formali, altre a sistemi di regole logico-matematiche, altre ancora a sistemi di transizione che specificano le trasformazioni di stato indotte da un programma.

**Approfondimento**

2.1

<sup>9</sup>A rigore, dovremmo parlare anche delle semantiche algebriche e di quelle assiomatiche, ma motivi di concisione e semplicità ci spingono a sorvolare su di esse.

## 2.6 Pragmatica

Se della sintassi e della semantica di un linguaggio di programmazione si può (e si deve) dare una descrizione precisa, non è vero altrettanto della pragmatica di un linguaggio. Ricordiamo che, per i nostri scopi, la pragmatica di un linguaggio risponde alle domande “A cosa serve un certo costrutto?”, o anche “Come uso un certo comando?” È evidente, dunque, che la pragmatica di un linguaggio di programmazione non viene stabilita una volta per tutte al momento della definizione del linguaggio (come avviene per sintassi e semantica). Al contrario, essa evolve assieme all’uso che del linguaggio viene fatto. Rientrano nella pragmatica tutti i suggerimenti di stile di programmazione, per esempio quello di evitare i salti (`goto`) ogni volta che sia possibile, o quello di usare le variabili di controllo di un ciclo `for` solo per quello scopo. Ma sono questioni di pragmatica anche la scelta della modalità più appropriata di passaggio di parametri ad una funzione, o la scelta tra un’iterazione determinata e una indeterminata.

In questo senso, la pragmatica di un linguaggio di programmazione confina con l’ingegneria del software (la disciplina che studia i metodi di progettazione e produzione del software), e non ci interessa troppo in questa sede. Per molti altri aspetti, invece, chiarire a che serve e come si usa un certo costrutto è una parte essenziale dello studio di un linguaggio di programmazione; è per questo che, nel seguito, faremo spesso annotazioni di pragmatica, anche senza notarle esplicitamente.

## 2.7 Implementazione

Anche l’ultimo livello della descrizione di un linguaggio di programmazione non sarà trattato qui in maniera organica. Come abbiamo ampiamente visto nel capitolo precedente, implementare un linguaggio vuol dire scriverne un compilatore e realizzare una macchina astratta per il linguaggio oggetto del compilatore, oppure scriverne un interprete e realizzare la macchina astratta del linguaggio nel quale l’interprete stesso è scritto oppure, come avviene nella pratica, un mix di entrambe queste cose. Ancora una volta, non è questo il testo in cui possiamo affrontare esaustivamente questi problemi. Ma non sarebbe corretto presentare i costrutti dei linguaggi di programmazione senza alcun riferimento al loro “costo” implementativo. Pur senza dare una visione d’insieme della costruzione di un interprete, per ogni costrutto ci chiederemo sempre “Come viene implementato?”, “Con quale costo?”. La risposta a queste domande ci aiuterà a comprendere meglio anche l’aspetto linguistico (perché un certo costrutto è fatto in un certo modo) e quello pragmatico (come possiamo usare al meglio un certo costrutto).

## 2.8 Sommario del capitolo

Il capitolo ha introdotto e discusso le tecniche fondamentali per la descrizione e l’implementazione di un linguaggio di programmazione.

- *La distinzione tra sintassi, semantica, pragmatica e implementazione:* ciascuna di queste discipline descrive un aspetto cruciale di un linguaggio.
- *Le grammatiche libere:* un mezzo formale essenziale per definire la sintassi di un linguaggio.
- *Gli alberi di derivazione e l'ambiguità:* gli alberi di derivazione rappresentano la struttura logica di una stringa e l'esistenza di un unico albero per ogni stringa permette una sua interpretazione canonica.
- *La semantica statica:* non tutti gli aspetti sintattici di un linguaggio sono descrivibili mediante grammatiche; i controlli di semantica statica intervengono per eliminare dai programmi legali quelle stringhe che, pur corrette secondo la grammatica, non rispettano gli ulteriori vincoli contestuali.
- *La struttura di un compilatore.*
- *La semantica operazionale strutturata:* un mezzo formale per descrivere la semantica di un linguaggio di programmazione mediante sistemi di transizioni.

## 2.9 Nota bibliografica

La letteratura sugli argomenti trattati in questo capitolo è sterminata, anche solo limitandosi ai testi introduttivi. Ci limitiamo a citare [45], l'ultima edizione di un testo classico sui linguaggi formali su cui si sono formate generazioni di studenti. Sui metodi di costruzione di un compilatore, ricordiamo [4] e [8], che trattano anche della generazione del codice. Un'introduzione alla semantica operazionale è [53]; ad un livello più avanzato si veda [106], che tratta anche della semantica denotazionale.

## 2.10 Esercizi

1. Si consideri la grammatica  $G''$ , ottenuta da quella della Figura 2.6 sostituendo le produzioni relative al simbolo non terminale  $T$  con le seguenti

$$T \rightarrow A \mid E * T.$$

Si mostri che  $G''$  è ambigua.

2. Si dia l'ovvia grammatica ambigua per il comando condizionale `if, then else`.
3. Prendendo come riferimento il frammento di grammatica della Figura 2.7, si costruisca un albero di derivazione per la stringa

```
if (espressione1) if (espressione2) comando1
  else comando2
```

assumendo che esistano le seguenti derivazioni:

$Expression \Rightarrow^* espressione1,$

$Expression \Rightarrow^* espressione2,$

$StatementWithoutTrailingSubstatement \Rightarrow^* comando1,$

$StatementWithoutTrailingSubstatement \Rightarrow^* comando2.$

## Analisi lessicale: linguaggi regolari

Questo capitolo tratta delle tecniche per progettare ed implementare l'analisi lessicale, cioè la trasformazione della stringa di ingresso in una lista di token. Descriveremo un modo generale ed efficiente per descrivere la struttura dei token mediante un certo tipo di espressioni. Queste espressioni, a loro volta, corrispondono in modo naturale ad automi dai quali si può estrarre un programma di riconoscimento. Inquadreremo questi argomenti nel contesto della teoria dei linguaggi regolari, di cui vedremo alcuni importanti risultati.

### 3.1 Analisi lessicale

Scopo dell'analisi lessicale è quello di riconoscere nella stringa di ingresso alcuni gruppi di caratteri che corrispondono a specifiche categorie sintattiche. In tal modo la stringa di ingresso è trasformata in una sequenza di simboli astratti, detti *token*, che sono poi passati all'analizzatore sintattico.

Non c'è un motivo teorico per separare l'analisi sintattica da quella lessicale: tutta l'elaborazione del testo sorgente potrebbe essere condotta mediante le tecniche di analisi sintattica che tratteremo nel Capitolo 4. Tuttavia la separazione tra i due tipi di analisi è molto utile perché le tecnologie per effettuare l'analisi lessicale sono più efficienti di quelle (più generali) usate per l'analisi sintattica. Inoltre, la specifica e la costruzione di un compilatore sono più semplici se possono essere suddivise in parti (e fasi) distinte. Questa separazione porta in genere anche ad una semplificazione della *descrizione* del linguaggio, cioè della grammatica che sarà usata dai progettisti delle macchine astratte e dai generici programmati.

#### 3.1.1 Token

Un *token* è un'informazione astratta che rappresenta una stringa del testo in ingresso. Più precisamente, un *token* è una coppia: la sua prima componente è il *nome* del token, cioè un simbolo astratto che rappresenta una specifica categoria sintattica (ad esempio gli identificatori, gli operatori aritmetici, una parola

Token	Una coppia $\langle \text{Nome}, \text{Valore} \rangle$	$\langle \text{IDE}, \text{x1} \rangle$
Nome	Informazione che identifica una classe di token	IDE
Valore	Informazione che identifica uno specifico token	x1
Pattern	Descrizione generale della forma dei valori di un token	$(x y)(x y 0 1)^*$
Lessema	Una stringa istanza di un pattern	x1

Figura 3.1 Riepilogo: token, pattern, lessemi.

riservata, ecc.). La seconda componente del token è il *valore* del token ed è costituita da una sequenza di simboli del testo di ingresso (ad esempio lo specifico identificatore, lo specifico operatore, ecc.).

Specificare un analizzatore lessicale significa elencare e specificare una lista di token: per ogni nome di token si definiscono quali sequenze di caratteri di ingresso sono riconosciute come “istanze” (cioè valori) di quel token. Le sequenze di caratteri associate ad un token sono specificate mediante un *pattern*, cioè una descrizione generale della forma che le sequenze di caratteri possono assumere. Questa descrizione generale è data mediante *espressioni regolari*, un concetto che introdurremo tra breve.

Una stringa dell’alfabeto di ingresso che corrisponde ad un pattern (cioè che, in inglese, fa *match* con il pattern) si dice *lessema*. Vi sono (nomi di) token a cui possono corrispondere molti (anche infiniti) lessemi (per esempio al token IDE — per “identificatore” — corrispondono tutte le stringhe che sono specifici identificatori). Ad altri token corrisponde un solo lessema: per esempio, una parola riservata costituisce da sola un token, che ha quella parola come unico lessema.

Usando questa terminologia, possiamo dire che il valore di un certo token è un lessema istanza del pattern associato a quel token. Nel seguito indicheremo i nomi dei token in MAIUSCOLETTTO; un token di nome N e valore v lo scriveremo come  $\langle N, v \rangle$ . Quando il valore coincide con il nome (come nel caso delle parole riservate) scriveremo solo il nome:  $\langle N \rangle$ . Ci riferiremo ad un token col suo nome.

### Esempio 3.1 Nella stringa C

```
if (x==0) printf("Zero")
```

potrebbero essere riconosciuti i token seguenti:  $\langle \text{IF} \rangle$ ,  $\langle () \rangle$ ,  $\langle \text{IDE}, \text{x} \rangle$ ,  $\langle \text{OPREL}, == \rangle$ ,  $\langle \text{CONST-NUM}, 0 \rangle$ ,  $\langle () \rangle$ ,  $\langle \text{IDE}, \text{printf} \rangle$ ,  $\langle () \rangle$ ,  $\langle \text{CONST-STRING}, \text{Zero} \rangle$ ,  $\langle () \rangle$ .

## 3.2 Espressioni regolari

Prima di descrivere come specificare un pattern, è opportuno introdurre un po’ di teoria dei linguaggi formali, che è il contesto teorico nel quale sono nate le tecnologie che ci interessano.

### 3.2.1 Linguaggi formali e operazioni

Fissiamo un alfabeto  $A$ , cioè un insieme finito di simboli. Una *stringa* su  $A$  è una successione finita di elementi di  $A$ . La *lunghezza* di una stringa  $s$  (indicata con  $|s|$ ) è il numero di elementi che compongono  $s$ . La stringa vuota (che indichiamo con  $\epsilon$ ) ha lunghezza zero. Un *linguaggio* sull’alfabeto  $A$  è un insieme di stringhe su  $A$ .

Tra due stringhe sullo stesso alfabeto possiamo definire l’operazione di *concatenazione*, che consiste nel mettere le due stringhe in sequenza. Date le due stringhe  $s$  e  $t$ , indichiamo la loro concatenazione con  $st$ . Osserviamo che si tratta di un’operazione non commutativa, per la quale la stringa vuota costituisce l’elemento neutro:  $s\epsilon = \epsilon s = s$ .

Iterando la concatenazione otteniamo una forma di esponenziazione, che possiamo definire come  $s^0 = \epsilon$  e  $s^{n+1} = s^n s$ .

Sulla base delle definizioni che abbiamo appena dato, possiamo definire alcune operazioni tra *linguaggi* sullo stesso alfabeto. Sia  $A$  un alfabeto fissato e  $L$ ,  $M$  due linguaggi su  $A$ .

**Concatenazione:**  $LM = \{uv \mid u \in L \text{ e } v \in M\}$ . Nel caso in cui  $L$  sia un singoletto ( $L = \{a\}$ , per  $a \in A$ ) scriveremo  $aM$  invece di  $\{a\}M$ . Analogamente nel caso in cui  $M$  sia un singoletto.

**Unione:**  $L \cup M = \{s \mid s \in L \text{ oppure } s \in M\}$ .

**Chiusura:** Iterando la concatenazione possiamo definire la *chiusura di Kleene*, indicata con  $L^*$  e costituita da tutte le concatenazioni finite di elementi di  $L$ . Iniziamo col definire  $L^0 = \{\epsilon\}$  e quindi  $L^{n+1} = L^n L$ . Definiamo ora  $L^* = \bigcup_{i \geq 0} L^i$ .

**Chiusura positiva:**  $L^+ = \bigcup_{i \geq 1} L^i$ . Si osservi che  $\epsilon \in L^+$  sse  $\epsilon \in L$ .

**Esempio 3.2** Abbiamo già introdotto l’operatore di Kleene informalmente nel Paragrafo 2.2.1. Dato un insieme  $L$  (finito o infinito) di stringhe,  $L^*$  è sempre un insieme infinito che contiene la stringa vuota  $\epsilon$ , tutti gli elementi di  $L$ , e tutte le stringhe che si ottengono mettendo in sequenza un numero finito di volte gli elementi di  $L$ . Se  $L$  contiene solo simboli (cioè tutti i suoi elementi sono di lunghezza 1),  $L^*$  è l’insieme di tutte le possibili stringhe finite su  $L$ , cioè è il più grosso linguaggio (rispetto all’inclusione insiemistica) definito sui simboli di  $L$ .

### 3.2.2 Espressioni regolari

Possiamo ora introdurre la nozione di espressione regolare, che ci serve per specificare il pattern associato ad un token. Ad ogni espressione regolare è associato in modo univoco un linguaggio, che costituisce l’insieme dei lessemi corrispondenti a quella espressione regolare.

**Definizione 3.3** Fissato un alfabeto  $A$ , definiamo induttivamente e in modo simultaneo le nozioni di espressione regolare su  $A$  e di linguaggio associato a (o anche definito da) un’espressione regolare  $s$ , che indicheremo con  $L[s]$ .

1.  $\epsilon$  è un'espressione regolare;  $\mathcal{L}[\epsilon] = \{\epsilon\}$ .
2. Per ogni  $a \in A$ ,  $a$  è un'espressione regolare;  $\mathcal{L}[a] = \{a\}$ .
3. Se  $r$  è un'espressione regolare,  $(r)$  è un'espressione regolare;  $\mathcal{L}[(r)] = \mathcal{L}[r]$ .
4. Se  $r$  e  $s$  sono espressioni regolari,  $(r) | (s)$  è un'espressione regolare;  
 $\mathcal{L}[(r)|(s)] = \mathcal{L}[r] \cup \mathcal{L}[s]$ .
5. Se  $r$  e  $s$  sono espressioni regolari,  $(r) \cdot (s)$  è un'espressione regolare;  
 $\mathcal{L}[(r) \cdot (s)] = \mathcal{L}[r]\mathcal{L}[s]$ .
6. Se  $r$  è un'espressione regolare,  $(r)^*$  è un'espressione regolare;  
 $\mathcal{L}[(r)^*] = \mathcal{L}[r]^*$ .
7. Nient'altro è un'espressione regolare.

Per limitare l'uso delle parentesi, conveniamo che concatenazione ( $\cdot$ ), disgiunzione ( $|$ ) e ripetizione ( $*$ ) associno a sinistra e che la precedenza tra loro sia, in ordine decrescente:  $*, \cdot, |$ . Inoltre, denoteremo la concatenazione con la semplice giustapposizione, cioè scriveremo  $st$  invece di  $s \cdot t$ .

**Esempio 3.4** (i) L'espressione regolare  $((a)|(b))^*((a)|(b))|(a)$  potrà essere scritta più semplicemente come  $(a|b)^*(a|b)|a$ .  
(ii) Invece di  $((b)^*)(a)|(c)$  potremo scrivere  $b^*a|c$ .

Si osservi che uno stesso linguaggio può essere in genere definito da più espressioni regolari; ad esempio, per  $s = (a|b)^*$  e  $t = (a^*b^*)^*$ , si ha  $\mathcal{L}[s] = \mathcal{L}[t] = \{a, b\}^*$ .

**Esempio 3.5** Fissiamo l'alfabeto  $A = \{a, b\}$ .

- (i) L'espressione  $ab|b$  definisce il linguaggio  $\{ab, b\}$ .
- (ii)  $a(a|b)b$  definisce il linguaggio  $\{aab, abb\}$ .
- (iii)  $a^*$  definisce  $\{\epsilon, a, aa, aaa, \dots\} = \{a^n \mid n \geq 0\}$ , cioè il linguaggio di tutte le stringhe finite costituite da sole  $a$ .
- (iv)  $b^*a|c$  definisce  $\{c\} \cup \{b^n a \mid n \geq 0\}$ .

Possiamo estendere le espressioni regolari con altre operazioni, che definiamo in termini di quelle della Definizione 3.3:

**Ripetizione positiva:**  $r^+ = rr^* = r^*r$ .

**Possibilità:**  $r? = r|\epsilon$ .

**Elenco:** La notazione  $[a_1, \dots, a_n]$ , per  $a_1, \dots, a_n \in A$ , indica uno dei simboli elencati:  $[a_1, \dots, a_n] = a_1 | \dots | a_n$ .

Inoltre, se tra gli elementi di  $A$  sussiste una relazione d'ordine canonica tale che  $a_i < a_{i+1}$ , allora definiamo anche  $[a_1 - a_n] = a_1 | \dots | a_n$ .

**Definizione 3.6** Un linguaggio  $L$  è regolare sse  $L$  è vuoto, oppure esiste un'espressione regolare  $s$  tale che  $L = \mathcal{L}[s]$ .

A proposito della definizione, si osservi la (ovvia) differenza tra il linguaggio vuoto  $\emptyset$  e il linguaggio che contiene la sola stringa vuota  $\{\epsilon\}$ . Il linguaggio che

### Identità tra espressioni regolari

Possiamo identificare due espressioni regolari quando corrispondono allo stesso linguaggio. Scriveremo dunque  $s = t$  se e solo se  $\mathcal{L}[s] = \mathcal{L}[t]$ . Da questa definizione seguono diverse importanti uguaglianze, che possono essere usate per "semplificare" espressioni regolari. Ne elenchiamo alcune, da dimostrare per esercizio.

· è commutativa	$r s = s r$
· è associativa	$r (s t) = (r s) t$
· · è associativa	$r(st) = (rs)t$
· $\epsilon$ è l'elemento neutro di ·	$\epsilon r = r\epsilon = r$
· * è idempotente	$r^{**} = r^*$
· distribuisce a sinistra su	$r(s t) = rs rt$
· distribuisce a destra su	$(s t)r = sr tr$

contiene la sola stringa vuota corrisponde all'espressione regolare  $\epsilon$ ; non ci sono espressioni regolari il cui linguaggio associato è vuoto.

Indagheremo nel seguito le proprietà dei linguaggi regolari e vedremo anche altri modi equivalenti per caratterizzare questa importante classe.

**Esempio 3.7** Possiamo definire il linguaggio dei numeri decimali senza segno con l'espressione  $[0-9]^+(\epsilon) \cdot [0-9]^+$ . Si tratta di un linguaggio sull'alfabeto  $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, .\}$ .

**Definizioni regolari** Per scrivere espressioni complesse è utile poterle suddividere in sottoespressioni, ognuna definita separatamente con un nome associato. Fissato un alfabeto  $A$ , una *definizione regolare su A* è costituita da una lista di definizioni

$$\begin{aligned} d_1 &:= s_1 \\ d_2 &:= s_2 \\ &\dots \\ d_k &:= s_k \end{aligned}$$

nella quale tutti i  $d_i$  sono simboli nuovi, distinti tra loro e non appartenenti ad  $A$ , mentre ogni  $s_i$  è un'espressione regolare sull'alfabeto  $A \cup \{d_1, \dots, d_{i-1}\}$ .

**Esempio 3.8** Possiamo elaborare sul linguaggio dei numeri decimali e definire

$$\begin{aligned} \text{numconsegno} &:= \text{segno cifre}(\cdot\text{cifre})? \\ \text{segno} &:= -|+ \\ \text{cifre} &:= \text{cifra}^+ \\ \text{cifra} &:= 0-9 \end{aligned}$$

### 3.3 Automi finiti

Se le espressioni regolari servono a *specificare* (la forma dei lessemi che corrispondono ad) un token, rimane il problema di *riconoscere* i lessemi ed associarli ai relativi token. La forma delle espressioni regolari permette la costruzione di riconoscitori efficienti e semplici, che sono basati sul concetto di *automa finito*, che è l'oggetto di studio di questo paragrafo.

Intuitivamente un automa finito è un macchina (cioè un algoritmo) che in ogni momento si trova in un certo stato, tratto da un insieme finito di stati possibili. La macchina consuma un simbolo alla volta del proprio input e sulla base del simbolo consumato e dello stato in cui si trova, transita in un nuovo stato. Questo processo continua fino a quando vi sono simboli da consumare. Dopo che la macchina ha consumato l'ultimo simbolo dell'input, si guarda lo stato in cui si è fermata: se fa parte di un insieme speciale di stati (detti stati *finali*), l'automa *accetta*, o *riconosce*, l'input che ha consumato. In caso contrario (o, come vedremo, se l'automa si blocca prima di poter consumare tutto l'input) la macchina *rifiuta* l'input.

La descrizione più compatta ed efficace di un automa finito è quella mediante un *diagramma di transizione*, cioè un grafo orientato i cui nodi rappresentano gli stati, mentre gli archi, etichettati con i simboli dell'alfabeto dell'input, rappresentano le possibili transizioni tra stati. Alcuni nodi sono opportunamente contrassegnati come finali, mentre un unico nodo è designato quale stato *iniziale*.

La Figura 3.2 rappresenta il grafo di transizione di un automa per il linguaggio dei numeri decimali senza segno, che abbiamo introdotto nell'Esempio 3.7. Vi sono quattro nodi, corrispondenti a quattro stati e contrassegnati con le lettere  $A, B, C, D$ . Lo stato  $A$  è iniziale (ha un arco entrante che proviene dal ... nulla), mentre  $B$  e  $D$  sono finali (contrassegnati dal doppio bordo). Quando un arco è etichettato con un insieme di simboli (come tutti gli archi dell'esempio eccetto quello con etichetta ":"), si intende che vi sono tanti archi con stessa sorgente e stessa destinazione quanti sono i simboli dell'insieme. Se all'automa specificato da questo diagramma viene presentata, ad esempio, la stringa  $23$ , l'automa inizia nello stato  $A$ , consuma  $2$  per portarsi in  $B$ , consuma  $3$ , ritorna in  $B$  e si ferma. Siccome  $B$  è finale,  $23$  è accettata. Se all'automa viene presentata  $0 \cdot 9^*$ , dopo tre transizioni (che consumano  $0, \cdot$  e  $9$ ) l'automa si trova nello stato  $D$  e si blocca perché l'input da consumare è  $\cdot$ , ma non vi sono transizioni possibili nel diagramma. Per questo motivo, anche se lo stato  $D$  è finale, l'automa rifiuta la stringa in input.

**Esempio 3.9** L'automa della Figura 3.2 non è certamente l'unico che corrisponde al linguaggio dell'Esempio 3.7. Anche quello della Figura 3.3 è progettato per accettare le stesse stringhe. Si tratta di un automa più compatto, perché ha un numero di stati inferiore.

Consideriamo ora l'espressione regolare  $(a|b)^*ba$ , che costituirà il nostro esempio canonico per questo paragrafo. Se proviamo a progettare un diagramma di transizione, è probabile che il primo tentativo dia luogo all'automa della Figu-

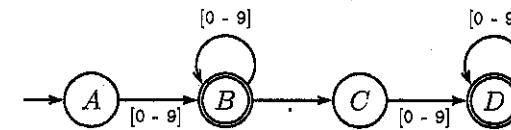


Figura 3.2 Un diagramma di transizione per  $[0-9]^+ (\epsilon) . [0-9]^+$ .

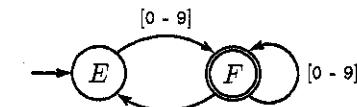


Figura 3.3 Un altro automa per  $[0-9]^+ (\epsilon) . [0-9]^+$ .

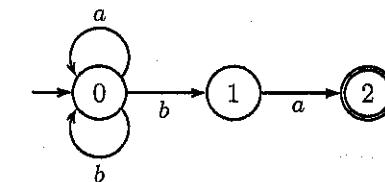


Figura 3.4 Un automa non deterministico per  $(a|b)^*ba$ .

ra 3.4. Si tratta di un automa finito secondo la definizione informale che abbiamo dato, ma a differenza dell'automa precedente presenta uno stato ( $0$ ) con due archi uscenti con la stessa etichetta ( $b$ ). In casi come questi diciamo che l'automa è *non deterministico*. Il problema è che la descrizione informale di funzionamento che abbiamo dato è ora incompleta: quale mossa viene scelta dall'automa quando si trova in  $0$  e l'input presenta  $b$ ? L'idea (che formalizzeremo tra breve) è che in questo caso il diagramma di transizione specifica *tutte* le possibilità che l'automa deve esplorare. Se c'è almeno un modo di proseguire consumando tutto l'input e terminando in uno stato finale, allora l'automa accetta. Rifiuta solo, pertanto, se non c'è alcun modo per scegliere la successione degli stati in corrispondenza dell'input e terminare in uno stato finale. Per esempio, l'automa della Figura 3.4 accetta  $aba$  perché c'è la successione di stati  $0, 1, 2$  che corrisponde ad una *possibile* computazione con quell'input (anche se lo stesso input permette una computazione che resta sempre in  $0$ ); accetta  $abba$ ; rifiuta  $aabb$  perché non c'è alcun modo per raggiungere uno stato finale con questa sequenza di input.

### 3.3.1 Automi finiti non deterministici

Dopo l'introduzione informale del paragrafo precedente, veniamo ora alla definizione formale di automa finito. Partiamo dalla definizione più generale, quella di automa non deterministico, per poi passare a quella di automa deterministico. Scopriremo che, come riconoscitori di linguaggi, automi deterministici e non deterministici hanno la stessa potenza: riconoscono esattamente la classe dei linguaggi regolari, cioè i linguaggi definiti da espressioni regolari.

**Definizione 3.10** Un automa finito (o a stati finiti) non deterministico (non deterministic finite state automaton, per brevità: NFA) è una quintupla  $(\Sigma, Q, \delta, q_0, F)$  dove

1.  $\Sigma$  è un alfabeto finito (i simboli di input);
2.  $Q$  è un insieme finito (gli stati);
3.  $\delta$  è la funzione di transizione con tipo

$$\delta : Q \times (\Sigma \cup \{\epsilon\}) \rightarrow \mathcal{P}(Q)$$

che dunque associa un insieme di stati ad ogni coppia  $(q, a)$  formata da uno stato e da un simbolo di input;

4.  $q_0 \in Q$  (lo stato iniziale);
5.  $F \subseteq Q$  (gli stati finali).

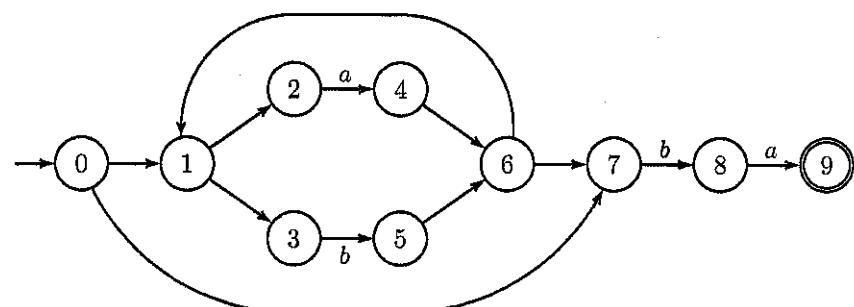
Un NFA può essere efficacemente rappresentato da un diagramma di transizione, dove un arco può anche essere etichettato da  $\epsilon$  (in tal caso nelle figure non scriveremo nessuna etichetta, a meno che non sia necessario per evitare confusione). I nodi del diagramma corrispondono a  $Q$ , mentre la funzione di transizione  $\delta$  corrisponde agli archi: se  $\delta(q, a) = \{q_1, \dots, q_n\}$ , per ogni nodo  $q_i \in \{q_1, \dots, q_n\}$  c'è un arco etichettato  $a$  uscente da  $q$  e diretto a  $q_i$ . Si osservi che qualche coppia  $(q, a) \in Q \times (\Sigma \cup \{\epsilon\})$  potrebbe avere  $\delta(q, a) = \emptyset$ , il che corrisponde al fatto che non vi sono archi etichettati  $a$  uscenti da  $q$ .

**Esempio 3.11** Il diagramma della Figura 3.4 corrisponde all'automa con

1. alfabeto:  $\{a, b\}$ ;
2. stati:  $\{0, 1, 2\}$ ;
3.  $\delta$  è definita dalla seguente tabella di transizione

Stato	$a$	$b$	$\epsilon$
0	$\{0\}$	$\{0, 1\}$	$\emptyset$
1	$\{2\}$	$\emptyset$	$\emptyset$
2	$\emptyset$	$\emptyset$	$\emptyset$

4. 0 è lo stato iniziale;
5.  $\{2\}$  è l'insieme degli stati finali.



**Figura 3.5** Un altro automa non deterministico per  $(a|b)^*ba$ .

La definizione più economica di accettazione di una stringa da parte di un NFA sfrutta il diagramma di transizione, sul quale, essendo un grafo orientato, è naturalmente definita una nozione di cammino.

**Definizione 3.12** Un NFA  $N = (\Sigma, Q, \delta, q_0, F)$  accetta la stringa  $x = a_1 \dots a_n$  se nel diagramma di transizione esiste un cammino che inizia in  $q_0$  e termina in uno stato di  $F$  nel quale la stringa che si ottiene concatenando le etichette degli archi percorsi è esattamente  $x$ .

#### Approfondimento 3.1

Si osservi che la definizione tiene conto del fatto che alcuni archi possono essere etichettati con la stringa vuota  $\epsilon$ : tali archi possono essere attraversati "silenziosamente", senza consumare nessun simbolo di input.

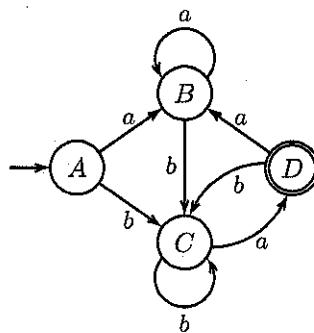
**Definizione 3.13** Il linguaggio accettato da un NFA  $N$  è l'insieme  $\mathcal{L}[N] = \{x \in \Sigma^* \mid N \text{ accetta } x\}$ .

**Esempio 3.14** La Figura 3.5 rappresenta un NFA con archi etichettati con  $\epsilon$ . L'automa accetta la stringa  $baba$  in virtù del cammino  $0, 1, 3, 5, 6, 1, 2, 4, 6, 7, 8, 9$ . Un po' di riflessione mostra che il linguaggio accettato da questo automa coincide con quello accettato dall'automa della Figura 3.4.<sup>1</sup>

### 3.3.2 Automi finiti deterministici

Un caso particolare di NFA si ha quando non vi sono archi etichettati con  $\epsilon$  e inoltre nella funzione di transizione si ha che  $\delta(q, a)$  è sempre un singoletto

<sup>1</sup>Nel Paragrafo 3.4 vedremo un modo canonico per ottenere automi da espressioni regolari.



**Figura 3.6** Un automa deterministico per  $(a|b)^*ba$ .

(cioè contiene esattamente un solo stato). In tal caso abbiamo un *automa finito deterministico*, per il quale possiamo dare la seguente definizione esplicita.

**Definizione 3.15** Un *automa finito deterministico* (*deterministic finite automaton*, per brevità: *DFA*) è una quintupla  $(\Sigma, Q, \delta, q_0, F)$  dove  $\Sigma, Q, q_0$  e  $F$  sono come in un NFA, mentre la funzione di transizione  $\delta$  ha tipo  $\delta : Q \times \Sigma \rightarrow Q$ .

Le definizioni di accettazione e di linguaggio accettato per un DFA sono ereditate da quelle per un NFA. Si osservi, tuttavia, che da uno stato di un DFA deve uscire *esattamente* una transizione per ciascun simbolo dell'alfabeto. A differenza di un NFA, quindi, un DFA non si blocca mai in corrispondenza di un carattere di input. Inoltre la successione di stati percorsa in corrispondenza di un certo input  $x$  (cioè il cammino etichettato  $x$  nel diagramma di transizione) è univocamente determinata da  $x$ . Dato un input  $x$ , insomma, un DFA ha uno ed un solo cammino etichettato  $x$  uscente dallo stato iniziale e che porterà il DFA in uno stato finale  $x$  appartiene al linguaggio accettato.

La Figura 3.6 riporta un DFA che riconosce il linguaggio  $(a|b)^*ba$ . Gli stati sono indicati con lettere anziché cifre, per un motivo che sarà chiaro tra breve.

### 3.3.3 DFA e NFA sono equivalenti

La situazione del nostro linguaggio d'esempio, che è accettato sia da NFA (come quelli delle Figure 3.4 e 3.5) sia da DFA (Figura 3.6) non è un caso. Mostreremo in questo paragrafo che a partire da un qualsiasi NFA  $N$  possiamo costruire un DFA  $M_N$  tale che  $\mathcal{L}[N] = \mathcal{L}[M_N]$ . Il viceversa (che ad un DFA corrisponde un NFA che accetta lo stesso linguaggio) è un'ovvietà: un DFA è infatti un caso particolare di NFA e le nozioni di accettazione e linguaggio accettato coincidono. DFA e NFA riconoscono dunque la stessa classe di linguaggi, che mostreremo coincidere proprio con la classe dei linguaggi regolari.

### Calcolare la $\epsilon$ -chiusura

La Definizione 3.16 è data in termini di una proprietà: l'insieme  $\epsilon\text{-clos}(q)$  deve godere delle due proprietà (i) e (ii) e si seleziona poi il più piccolo tra tutti gli insiemi che godono di quelle proprietà (si osservi che  $Q$ , l'insieme di tutti gli stati, soddisfa sempre (i) e (ii)). Non è difficile estrarre dalla definizione un algoritmo che effettivamente calcoli  $\epsilon\text{-clos}(P)$ .

```

Inizializza T = P;
Inizializza  $\epsilon\text{-clos}(P) = P$ ;
while T ≠ ∅ do{
    scegli un qualsiasi r da T e rimuovilo;
    foreach s ∈ δ(r, ε){
        if s ∉  $\epsilon\text{-clos}(q)$ {
            add s to  $\epsilon\text{-clos}(q)$ ;
            add s to T;
        }
    }
}
  
```

**Notazioni utili** In un NFA ci sono due cause di non determinismo. Una è data dalle  $\epsilon$ -transizioni, che sono eseguite senza consumare input. Un'altra è la possibilità che da uno stato partano più transizioni con la stessa etichetta.

Per trasformare un NFA in un DFA equivalente, è utile poter indicare in modo sintetico gli stati raggiungibili a partire da uno stato fissato, esplicitando le scelte non deterministiche. Iniziamo col definire la  $\epsilon$ -chiusura di uno stato  $q$ , cioè l'insieme degli stati che si possono raggiungere da  $q$   $\epsilon$ -transizioni.

**Definizione 3.16** (a) Fissato un NFA  $N = (\Sigma, Q, \delta, q_0, F)$  ed uno stato  $q \in Q$ , la  $\epsilon$ -chiusura di  $q$ , indicata con  $\epsilon\text{-clos}(q)$ , è il più piccolo insieme di stati tale che (i)  $q \in \epsilon\text{-clos}(q)$ ; (ii) se  $p \in \epsilon\text{-clos}(q)$  allora  $\delta(p, \epsilon) \subseteq \epsilon\text{-clos}(q)$ .

(b) Se  $P$  è un insieme di stati, definiamo  $\epsilon\text{-clos}(P) = \bigcup_{p \in P} \epsilon\text{-clos}(p)$ .

Per gestire l'altra forma di non determinismo, definiamo la seguente funzione, che, dato un insieme di stati  $P$  e un simbolo  $a$ , restituisce l'insieme degli stati in cui si può trovare l'automa partendo da uno stato in  $P$  e consumando  $a$ :

$$\begin{aligned} mossia : \mathcal{P}(Q) \times \Sigma &\rightarrow \mathcal{P}(Q) \\ mossia(P, a) &= \bigcup_{p \in P} \delta(p, a). \end{aligned}$$

Se un NFA si trova in un certo stato  $q$ , consumando il simbolo di input  $a$  potrà trovarsi in uno qualsiasi degli stati di  $\epsilon\text{-clos}(\text{mossa}(\{q\}), a)$ .

**Costruire un DFA a partire da un NFA** Dato un NFA  $N$ , costruiremo un DFA  $M_N$  che riconosce lo stesso linguaggio. L'idea di fondo è che uno stato di  $M_N$  può essere pensato come un insieme di stati del NFA.<sup>2</sup> In particolare, se

<sup>2</sup>Dunque se il NFA ha  $n$  stati, il DFA può avere, almeno in teoria, fino a  $2^n$  stati.

$N$ , partendo dallo stato iniziale e consumando l'input  $a_1 \dots a_k$  può trovarsi in uno qualsiasi degli stati  $q_1, \dots, q_m$  (seguendo ovviamente percorsi diversi, visto che si tratta di un NFA), allora  $M_N$ , iniziando nel suo stato iniziale e consumando (deterministicamente) lo stesso input  $a_1 \dots a_k$  si troverà nello stato che corrisponde all'insieme  $\{q_1, \dots, q_m\}$ . Per questo, l'algoritmo che vedremo è noto come *costruzione per sottinsiemi*.

Sia dunque  $N = (\Sigma, Q, \delta, q_0, F)$  un generico NFA. Applichiamo il seguente algoritmo, nel quale  $T$  conterrà alla fine l'insieme degli stati di  $M_N$  e  $\Delta$  costituirà la funzione di transizione di  $M_N$ .

```
Inizializza  $S = \epsilon\text{-clos}(q_0); // S$  insieme di stati
Inizializza  $T = \{S\}; // T$  insieme di insiemi di stati
                           // S non ha ancora marchi
finché c'è un  $P \in T$  non marcato{
    marca  $P$ ;
    foreach  $a \in \Sigma\{$ 
         $R = \epsilon\text{-clos}(\text{mossa}(P, a));$ 
        if  $R \notin T\{ \text{add } R \text{ to } T; \} // R$  non ha ancora marchi
        definisci  $\Delta(P, a) = R;$ 
    }
}
```

Il DFA  $M_N$  è ora definito come  $M_N = (\Sigma, T, \Delta, \epsilon\text{-clos}(q_0), F)$ , dove  $F$  è l'insieme di quegli elementi di  $T$  che contengono almeno uno stato finale dell'automa originale  $N$ . Formalmente:  $R \in F$  sse esiste  $q \in R$  con  $q \in F$ .

**Esempio 3.17** Applicando la costruzione per sottinsiemi al NFA della Figura 3.5, si ottiene l'automa della Figura 3.6, dove la corrispondenza tra nomi degli stati e insiemi del NFA è la seguente:

Stato DFA	Stati del NFA
$A$	$\{0, 1, 2, 3, 7\}$
$B$	$\{1, 2, 3, 4, 6, 7\}$
$C$	$\{1, 2, 3, 5, 6, 7, 8\}$
$D$	$\{1, 2, 3, 4, 6, 7, 9\}$

Lo stato  $A$  è iniziale, perché è la  $\epsilon$ -chiusura di 0;  $D$  è finale perché (è l'unico stato che) contiene lo stato 9, finale nel NFA.

**Approfondimento** 3.2

**Teorema 3.18** Sia  $N = (\Sigma, Q, \delta, q_0, F)$  un NFA e sia  $M_N$  l'automa ottenuto con la costruzione per sottinsiemi. Allora  $M_N$  è un DFA e si ha  $\mathcal{L}[N] = \mathcal{L}[M_N]$ .

**Approfondimento** 3.3

**Corollario 3.19** La classe dei linguaggi riconosciuti dagli NFA coincide con la classe dei linguaggi riconosciuti dai DFA.

### 3.4 Da espressioni regolari ad automi finiti

All'inizio del paragrafo precedente abbiamo introdotto gli automi finiti come possibili riconoscitori dei linguaggi definiti da espressioni regolari. È venuto il momento di vedere un modo canonico per costruire un NFA a partire da un'espressione regolare. È possibile anche il viceversa, ma quest'ultima trasformazione la otterremo più avanti, come sottoprodotto di altre costruzioni.

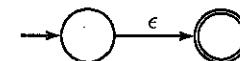
Diamo dunque un metodo generale che, data un'espressione regolare  $s$ , costruisce un NFA  $\mathcal{N}[s]$  che riconosce lo stesso linguaggio. La costruzione procede per induzione, secondo i casi della definizione induttiva di espressione regolare (Definizione 3.3). Dimostriamo dunque il seguente

**Teorema 3.20** Data un'espressione regolare  $s$ , possiamo costruire un NFA  $\mathcal{N}[s]$  con  $\mathcal{L}[s] = \mathcal{L}[\mathcal{N}[s]]$ .

**Dim.** Costruiremo  $\mathcal{N}[s]$  in modo da mantenere i seguenti invarianti, che ci semplificheranno la costruzione:

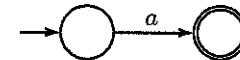
1. lo stato iniziale di  $\mathcal{N}[s]$  non ha archi entranti;
2.  $\mathcal{N}[s]$  ha un solo stato finale, che non ha archi uscenti.

$s \equiv \epsilon$  Definiamo  $\mathcal{N}[\epsilon]$  come:



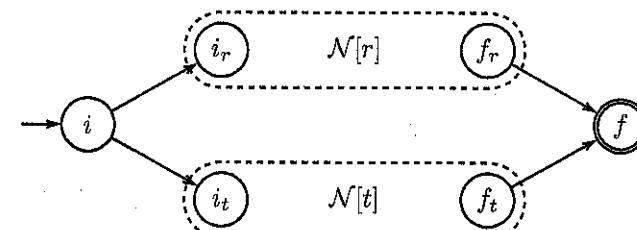
Che  $\mathcal{L}[\epsilon] = \mathcal{L}[\mathcal{N}[\epsilon]]$  è ovvio, come è ovvio che gli invarianti sono soddisfatti.

$s \equiv a$  Definiamo  $\mathcal{N}[a]$  come:



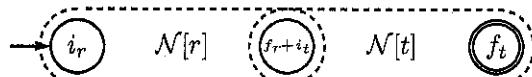
Anche in questo caso la tesi è immediata.

$s \equiv r|t$  Per induzione, possiamo assumere di aver costruito  $\mathcal{N}[r]$  e  $\mathcal{N}[t]$ , entrambi che soddisfano i due invarianti. Siano  $i_r$  e  $i_t$  gli stati iniziali, rispettivamente, di  $\mathcal{N}[r]$  e  $\mathcal{N}[t]$ ; analogamente, siano  $f_r$  e  $f_t$  i loro stati finali. Definiamo  $\mathcal{N}[r|t]$  come:



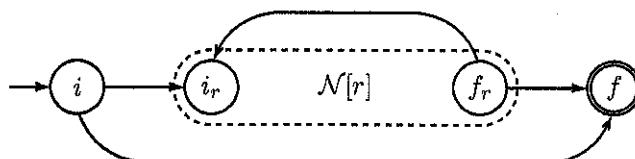
dove  $i$  e  $f$  sono nuovi stati (non appartenenti agli stati né di  $\mathcal{N}[r]$  né di  $\mathcal{N}[t]$ );  $i$  è lo stato iniziale, e  $f$  è lo stato finale, di  $\mathcal{N}[r|t]$ . I due invarianti sono ovviamente soddisfatti. In  $\mathcal{N}[r|t]$  si raggiunge lo stato finale solo seguendo un cammino in  $\mathcal{N}[r]$ , oppure seguendo un cammino in  $\mathcal{N}[t]$ . Quindi, sfruttando l'ipotesi induttiva che  $\mathcal{L}[r] = \mathcal{L}[\mathcal{N}[r]]$  e  $\mathcal{L}[t] = \mathcal{L}[\mathcal{N}[t]]$ , si ha la tesi che  $\mathcal{L}[r|t] = \mathcal{L}[\mathcal{N}[r|t]]$ .

$s \equiv rt$  Come nel caso precedente, e con la stessa notazione, assumiamo di aver costruito  $\mathcal{N}[r]$  e  $\mathcal{N}[t]$ . Definiamo  $\mathcal{N}[rt]$  come:



dove  $i_r$  è definito come stato iniziale del nuovo automa; il suo unico stato finale è  $f_t$ . Lo stato  $f_r + i_t$  è un nuovo stato risultato della fusione di  $f_r$  e  $i_t$  in un'unica stato, mantenendo gli archi entranti in  $f_r$  e gli archi uscenti da  $i_t$ . I due invarianti sono ovviamente soddisfatti. Per quanto riguarda la tesi sui linguaggi accettati, osserviamo che in  $\mathcal{N}[rt]$  si può raggiungere lo stato finale solo attraverso un cammino in  $\mathcal{N}[r]$  (dal suo stato iniziale a quello finale) seguito da un cammino in  $\mathcal{N}[t]$  (dal suo stato iniziale al suo stato finale), consumando quindi un input in  $\mathcal{L}[rt]$ . Si osservi che una volta entrati in  $\mathcal{N}[t]$  non è possibile rientrare in  $\mathcal{N}[r]$  (attraverso il suo stato finale  $f_r$  fuso in  $f_r + i_t$ ), perché l'invariante (1) assicura che non vi sono archi entranti in  $i_t$  provenienti da  $\mathcal{N}[r]$ .

$s \equiv r^*$  Sia  $\mathcal{N}[r]$  l'automa costruito induttivamente per  $r$ . Definiamo  $\mathcal{N}[r^*]$  come:



dove  $i$  è un nuovo stato, iniziale per  $\mathcal{N}[r^*]$ , e  $f$  è un nuovo stato, finale per  $\mathcal{N}[r^*]$ . I due invarianti sono soddisfatti. Per quanto riguarda i linguaggi, osserviamo che in  $\mathcal{N}[r^*]$  vi sono due tipi di cammini accettanti: il primo inizia in  $i$  e termina immediatamente in  $f$  attraverso la  $\epsilon$  transizione (riconoscendo dunque la stringa vuota  $\epsilon$ ). Gli altri cammini accettanti sono quelli che attraversano  $\mathcal{N}[r]$  un numero finito di volte e poi transitano silenziosamente in  $f$ . Siccome per induzione  $\mathcal{N}[r]$  accetta  $\mathcal{L}[r]$ , questa seconda classe di cammini accetta le stringhe in  $\mathcal{L}[r]^n$  per  $n \geq 1$ . Mettendo insieme il cammino del primo tipo, si ottiene che  $\mathcal{N}[r^*]$  accetta  $\mathcal{L}[r]^n$  per  $n \geq 0$ , cioè proprio  $\mathcal{L}[r^*]$ .

$s \equiv (r)$  Definiamo  $\mathcal{N}[(r)] = \mathcal{N}[r]$ . La tesi è ovvia.  $\square$

**Esempio 3.21** Prendiamo la nostra espressione regolare d'esempio,  $(a|b)^*ba$ . Applicando la costruzione del Teorema 3.20 otteniamo lo NFA della Figura 3.5.

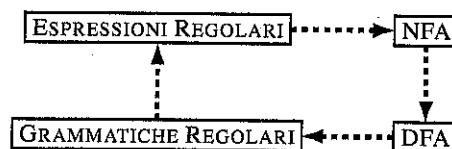


Figura 3.7 Una catena di equivalenze.

### 3.5 Automi finiti e grammatiche

I linguaggi regolari (cioè definiti da espressioni regolari, Definizione 3.6) possono essere definiti anche mediante grammatiche libere da contesto. D'altra parte una generica grammatica libera è uno strumento troppo potente per poterlo "ridurre" ad un'espressione regolare (vedremo questo fatto in modo preciso nel Paragrafo 3.8). Mettendo opportuni vincoli alla forma delle produzioni, tuttavia, è possibile definire una classe di grammatiche che corrisponde esattamente alla classe dei linguaggi regolari. In questo paragrafo studieremo questa corrispondenza, definendo in primo luogo la classe delle grammatiche regolari.

Nella Figura 3.7, sono riportati i principali concetti che abbiamo incontrato in questo capitolo. Le frecce che collegano due concetti corrispondono ai teoremi di corrispondenza che abbiamo dimostrato (o che dimostreremo): un'espressione regolare può essere trasformata in un NFA che riconosce lo stesso linguaggio; a sua volta un NFA può essere trasformato in un DFA per lo stesso linguaggio. In questo paragrafo prenderemo in considerazione le due altre frecce che "chiudono" il quadrato: mostreremo prima che ad ogni DFA corrisponde una grammatica regolare, per poi dimostrare che ogni grammatica regolare definisce un linguaggio corrispondente ad un'espressione regolare. In questo modo avremo dimostrato che le quattro nozioni elencate in figura sono tutte equivalenti: tutte e quattro definiscono la stessa classe di linguaggi.

Ricordiamo innanzitutto che la Definizione 2.2 introduce una *grammatica libera* come una quadrupla  $(NT, T, R, S)$ , dove  $R$  è un insieme di *produzioni*, cioè espressioni della forma  $V \rightarrow w$  con  $V \in NT$  e  $w \in (T \cup NT)^*$ . Le grammatiche regolari si ottengono restringendo drasticamente la forma della parte destra.

**Definizione 3.22** Una grammatica libera è regolare sse ogni produzione è della forma  $V \rightarrow aW$  oppure  $V \rightarrow a$ , dove  $V, W \in NT$  e  $a \in T$ . Per il simbolo iniziale è ammessa anche la produzione  $S \rightarrow \epsilon$ .

La definizione di grammatica regolare può essere data in molti modi equivalenti, nel senso che non cambia la classe dei *linguaggi definiti* da queste grammatiche. Alcuni di questi modi sono i seguenti:

- ogni produzione è della forma  $V \rightarrow aW$  oppure  $V \rightarrow a$ , dove  $V, W \in NT$  e  $a \in T$ ; per il simbolo iniziale è ammessa la produzione  $S \rightarrow \epsilon$  (Def. 3.22);

2. ogni produzione è della forma  $V \rightarrow Wa$  oppure  $V \rightarrow a$ , dove  $V, W \in NT$  e  $a \in T$ ; per il simbolo iniziale è ammessa la produzione  $S \rightarrow \epsilon$ ;
3. ogni produzione è della forma  $V \rightarrow Wx$  oppure  $V \rightarrow x$ , dove  $V, W \in NT$  e  $x \in T^*$  (queste grammatiche sono dette *lineari sinistre*);
4. ogni produzione è della forma  $V \rightarrow xW$  oppure  $V \rightarrow x$ , dove  $V, W \in NT$  e  $x \in T^*$  (queste grammatiche sono dette *lineari destre*).

**Esempio 3.23** La grammatica con  $NT = \{A, B, C\}$ ,  $T = \{a, b\}$ , simbolo iniziale  $A$  e produzioni

$$\begin{array}{l} A \rightarrow aA \mid bB \\ B \rightarrow bB \mid aC \mid a \\ C \rightarrow aA \mid bB \end{array}$$

è una grammatica regolare, che genera il linguaggio dell'espressione  $(a|b)^*ba$ .

**Teorema 3.24** Per ogni DFA  $M$  possiamo costruire una grammatica regolare  $G_M$  tale che  $\mathcal{L}[M] = \mathcal{L}[G_M]$ .

**Dim.** La costruzione non è difficile. Sia  $M = (\Sigma, Q, \delta, q_0, F)$  il DFA assegnato. La grammatica  $G_M = (Q, \Sigma, R, q_0)$  ha:

1. come non terminali, gli stati di  $M$ ;
2. come terminali, l'alfabeto di  $M$ ;
3. come simbolo iniziale, lo stato iniziale  $q_0$  di  $M$ ;
4. come produzioni  $R$ :
  - (a) per ogni transizione  $\delta(q_i, a) = q_j$  di  $M$ , la produzione  $q_i \rightarrow aq_j \in R$ ; inoltre, se  $q_j \in F$ , anche la produzione  $q_i \rightarrow a \in R$ ;
  - (b) se lo stato iniziale di  $M$  è anche finale (cioè  $q_0 \in F$ ), allora la produzione  $q_0 \rightarrow \epsilon \in R$ .

Per dimostrare che  $\mathcal{L}[M] = \mathcal{L}[G_M]$  osserviamo che ogni cammino di accettazione del DFA corrisponde esattamente ad una derivazione della grammatica.

**Approfondimento** 3.4

**Esempio 3.25** Al DFA della Figura 3.6 corrisponde la grammatica regolare con produzioni

$$\begin{array}{l} A \rightarrow aB \mid bC \\ B \rightarrow aB \mid bC \\ C \rightarrow aD \mid bC \mid a \\ D \rightarrow aB \mid bC. \end{array}$$

**Approfondimento** 3.5

Concludiamo la nostra catena di equivalenze mostrando che ad ogni grammatica regolare corrisponde un linguaggio regolare.

**Teorema 3.26** Il linguaggio definito da una grammatica regolare  $G$  è un linguaggio regolare. Cioè  $L$  è vuoto, oppure è possibile costruire un'espressione regolare  $s_G$  tale che  $\mathcal{L}[G] = \mathcal{L}[s_G]$ .

**Approfondimento** 3.6

## 3.6 Minimizzare un DFA

Nella generazione di un analizzatore lessicale, si parte dalla specifica dei (pattern associati ai) token mediante espressioni regolari. Queste espressioni sono poi usate per costruire gli NFA corrispondenti, dai quali, mediante la costruzione per sottinsiemi, si estrae un DFA. A questo punto un DFA può essere direttamente usato come programma riconoscitore sulla stringa da analizzare. La costruzione per sottinsiemi potrebbe generare un DFA con molti più stati del NFA da cui siamo partiti: il caso peggiore è un NFA con  $n$  stati che genera un DFA con  $2^n$  stati (anche se nei casi reali questo difficilmente accade).

La simulazione del DFA richiede che la tabella di transizione sia mantenuta in memoria: la cardinalità degli stati è dunque una misura dell'occupazione di memoria dell'analizzatore lessicale. Non è quindi del tutto inutile chiedersi se esistano tecniche per *minimizzare* un DFA, cioè per ridurre il numero degli stati, mantenendo inalterato il linguaggio accettato. Vedremo in questo paragrafo un metodo generale che, applicato ad un generico DFA, produce il *DFA minimo* (cioè con numero minimo di stati) che riconosce lo stesso linguaggio. È inoltre possibile mostrare che l'automa minimo è unico, a meno di isomorfismo del relativo diagramma di transizione.

L'idea che soggiace alla minimizzazione è che in un DFA non minimo vi sono certamente stati *equivalenti*, cioè che si comportano in modo indistinguibile rispetto all'accettazione. Quando questo accade, gli stati equivalenti possono essere fusi tra loro, riducendo così il numero totale di stati.

Iniziamo col precisare la nozione di equivalenza che ci interessa: è intuitivamente più semplice definire quando due stati *non* sono equivalenti.

**Definizione 3.27** Sia  $M$  un DFA sull'alfabeto  $\Sigma$  e  $x \in \Sigma^*$ ; siano  $q_1$  e  $q_2$  due stati di  $M$ . La stringa  $x$  distingue  $q_1$  e  $q_2$  sse (i) il cammino che parte in  $q_1$  e consuma  $x$  arriva in uno stato finale, mentre il cammino che parte in  $q_2$  e consuma  $x$  arriva in uno stato non finale; (ii) oppure, viceversa, il cammino che parte in  $q_1$  e consuma  $x$  arriva in uno stato non finale, mentre il cammino che parte in  $q_2$  e consuma  $x$  arriva in uno stato finale.

**Approfondimento** 3.7

**Esempio 3.28** Nell'automa della Figura 3.6, abbiamo:

	B	C	D		B	C	D		B	C	D	
				A				A	X	X	X	A
				B				B	X	X	X	B
(a)												
(b)												
(c)												

Figura 3.8 Calcolo degli stati indistinguibili per il DFA della Figura 3.6.

1.  $\epsilon$  distingue ogni stato in  $F$  da ogni stato non finale (questo accade sempre, per ogni DFA);
2.  $a$  distingue:  $B$  e  $C$ ;  $A$  e  $C$ ;  $C$  e  $D$ ;
3.  $b$  non distingue nessuna coppia di stati.

Si osservi che  $A$  e  $B$  non sono distinti da nessuna stringa: infatti le stringhe che iniziano con  $a$  portano in  $B$  sia partendo da  $A$  che da  $B$ ; le stringhe che iniziano con  $b$  portano in  $C$  sia da  $A$  che da  $B$ . I due stati  $A$  e  $B$  sono *indistinguibili*.

**Definizione 3.29** Sia  $M$  un DFA sull'alfabeto  $\Sigma$ ; due stati  $q_1$  e  $q_2$  di  $M$  sono indistinguibili (o equivalenti) sse nessuna  $x \in \Sigma^*$  distingue  $q_1$  e  $q_2$ .

**Teorema 3.30** La relazione di indistinguibilità è una relazione di equivalenza (ovvero è riflessiva, simmetrica e transitiva).

#### Approfondimento

3.8

Se in un DFA vi sono due stati equivalenti, questi possono essere fusi (cioè identificati). Ovviamente, in generale l'identificazione di due stati può portare confusione nella funzione di transizione: se  $\delta(q_1, a) = s$  e  $\delta(q_2, a) = t$ , come è definita  $\delta$  sul risultato della fusione di  $q_1$  e  $q_2$ , se  $s \neq t$ ? Il fatto che gli stati fusi sono equivalenti, tuttavia, significa proprio che non si genera confusione, perché  $s$  e  $t$  possono certo essere diversi, ma sono necessariamente equivalenti, e dunque saranno fusi anch'essi in un unico stato. Dimostreremo tutto questo dopo aver presentato un algoritmo per la minimizzazione di un DFA.

Un modo relativamente efficiente per determinare quando due stati sono distinguibili è quello di usare una *tabella a scala*, cioè la parte triangolare inferiore di una matrice quadrata  $|Q| \times |Q|$ . Iniziamo col descrivere un esempio, per poi dare il metodo generale.

**Esempio 3.31** Consideriamo ancora il DFA della Figura 3.6. La tabella a scala (vuota) per questo automa è riportata in Figura 3.8(a). La tabella ha un elemento per ogni coppia di stati distinti; l'elemento nella coppia  $(p, q)$  è destinato a contenere  $X$  se  $p$  e  $q$  sono distinguibili, altrimenti rimarrà vuoto. Il riempimento inizia mettendo  $X$  in ogni coppia  $(p, q)$  dove uno dei due stati è finale, mentre l'altro non è finale (Figura 3.8(b)). Adesso, si prendono in considerazione, una

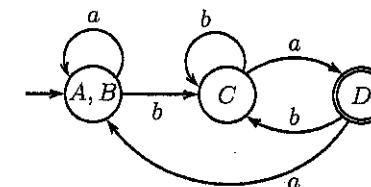


Figura 3.9 Automa deterministico minimo per  $(a|b)^*ba$ .

dopo l'altra, tutte le coppie  $(p, q)$ : si marca con  $X$  la coppia se esiste un simbolo  $a$  dell'alfabeto per cui la coppia  $(\delta(p, a), \delta(q, a))$  è già marcata con  $X$  (infatti in tal caso  $a$  distingue  $p$  da  $q$ ). Nel nostro caso, marchiamo  $(A, C)$  (perché  $a$  li distingue) e  $(B, C)$  (distinti anch'essi da  $a$ ), col risultato in Figura 3.8(c). Per la coppia  $(A, B)$ , invece, vediamo che con  $a$  si va nella coppia  $(B, B)$ , mentre  $b$  porta  $(A, B)$  in  $(C, C)$ : in questo caso  $A$  e  $B$  sono certo indistinguibili e lasciamo vuota la casella  $(A, B)$ . Avendo esaminato tutte le coppie, l'algoritmo termina. L'automa minimo è quello in cui  $A$  e  $B$  sono identificati (cioè la partizione  $\Pi$  è data da  $\{A, B\}$ ,  $\{C\}$ ,  $\{D\}$ ). L'automa minimo è riportato in Figura 3.9.

L'Esempio 3.31 è particolarmente semplice, perché è stato possibile determinare subito la marca da inserire in ogni casella. In generale, oltre ai due casi evidenziati nell'esempio, potrà capitare che partendo da due stati  $(p, q)$ :

1. non c'è nessun simbolo  $a$  che porta in una coppia già marcata con  $X$  (e dunque non è possibile marcare  $(p, q)$  con  $X$ ); e inoltre
2. non tutte le coppie  $(\delta(p, a), \delta(q, a))$  sono nella forma  $\delta(p, a) = \delta(q, a)$ : non è dunque possibile affermare con sicurezza che  $p$  e  $q$  sono indistinguibili.

Se siamo in questa situazione, c'è dunque un simbolo  $b$  per cui  $(\delta(p, b), \delta(q, b))$  non ha ancora nessuna marca e  $\delta(p, b) \neq \delta(q, b)$ . In questo caso la marca da inserire in  $(p, q)$  dipende da quella che sarà inserita in  $(\delta(p, b), \delta(q, b))$ . L'algoritmo tiene in conto questo fatto memorizzando in ogni casella la lista delle caselle che dipendono da essa; quando lo stato di quella casella sarà determinato, potrà essere aggiornato lo stato delle coppie nella lista.

**Algoritmo di riempimento della tabella a scala** Dato un DFA  $M = (\Sigma, Q, \delta, q_0, F)$ , costruisce la tabella a scala per  $Q$ . Durante il riempimento, ogni elemento  $(p, q)$  di tale tabella potrà contenere:

- $X$ : in tal caso  $p$  e  $q$  sono distinguibili;
- nulla: in tal caso  $p$  e  $q$  non sono ancora distinti;
- alcune coppie  $(p', q')$ : in tal caso, se si determinerà che  $p$  e  $q$  sono distinguibili, allora anche tutte le coppie  $(p', q')$  sono distinguibili.

Per riempire la tabella:

1. marca  $X$  ogni coppia  $(p, q)$  tale che  $p \in F$  e  $q \in Q - F$  o viceversa;
2. per ogni altra coppia  $(p, q)$ :
  - se esiste un  $a \in \Sigma$  con  $(\delta(p, a), \delta(q, a))$  già marcata  $X$ :
    - (a) marca  $(p, q)$  con  $X$ ;
    - (b) marca con  $X$  tutte le coppie elencate in  $(p, q)$ , e procedi ricorsivamente la marcatura su tutte le coppie marcate in questo modo;
  - altrimenti, per ogni  $a \in \Sigma$  tale che  $\delta(p, a) \neq \delta(q, a)$ , metti  $(p, q)$  nella lista di  $(\delta(p, a), \delta(q, a))$ .
3. Al termine del riempimento, sia  $J$  l'insieme delle coppie che non contengono  $X$ .
4. La relazione di indistinguibilità  $I$  (Definizione 3.29) è data dalla chiusura riflessiva e simmetrica di  $J$  (cioè dall'insieme di coppie  $I = J \cup \{(q, q) \mid q \in Q\} \cup \{(q, p) \mid (p, q) \in J\}$ ).

**Teorema 3.32** *Dato un DFA  $M = (\Sigma, Q, \delta, q_0, F)$ , l'algoritmo di riempimento della tabella a scala termina. Due stati  $p$  e  $q$  sono distinguibili se e solo se la casella  $(p, q)$  (o  $(q, p)$ ) è marcata con  $X$ .*

**Approfondimento** 3.9

**L'automa minimo** La relazione di indistinguibilità tra stati  $I$  (costruita al passo 4 dell'algoritmo) è l'ingrediente cruciale per la costruzione dell'automa minimo. Dato un DFA  $M = (\Sigma, Q, \delta, q_0, F)$ , l'automa minimo equivalente  $M_{\min} = (\Sigma, Q_{\min}, \delta_{\min}, [q_0], F_{\min})$  è dato da:

- $Q_{\min}$  è l'insieme delle classi di equivalenza  $[q]$  della relazione di indistinguibilità  $I$ , da cui sono rimosse le classi composte da soli stati irraggiungibili (in  $Q$ ) a partire da  $q_0$ ;
- $\delta_{\min}([q], a) = [\delta(q, a)]$ ;
- $F_{\min} = \{[q] \mid q \in F\}$ .

**Teorema 3.33** *Dato un DFA  $M = (\Sigma, Q, \delta, q_0, F)$ , l'automa  $M_{\min}$  è correttamente definito, riconosce lo stesso linguaggio di  $M$ , ed ha il minimo numero di stati tra tutti gli automi per questo linguaggio.*

**Approfondimento** 3.10

**Esempio 3.34** Prendiamo in considerazione il DFA sull'alfabeto  $\{0, 1\}$ , con  $A$  stato iniziale,  $D$  unico stato finale e funzione di transizione data dalla tabella:

	0	1
$A$	$B$	$A$
$B$	$A$	$C$
$C$	$D$	$B$
$D$	$D$	$A$
$E$	$D$	$F$
$F$	$G$	$E$
$G$	$F$	$G$
$H$	$G$	$D$

L'automa ha alcuni stati irraggiungibili a partire da  $A$ , ma questo non impedisce di applicare l'algoritmo di minimizzazione. Il riempimento della tabella a scala produce:

$B$	$X$					
$C$	$X$	$X$				
$D$	$X$	$X$	$X$			
$E$	$X$	$X$		$X$		
$F$	$X$		$X$	$X$	$X$	
$G$		$X$	$X$	$X$	$X$	$X$
$H$	$X$	$X$	$X$	$X$	$X$	$X$
$A$	$B$	$C$	$D$	$E$	$F$	$G$

Gli stati indistinguibili sono dunque  $\{A, G\}$ ,  $\{B, F\}$  e  $\{C, E\}$ . Lo stato/classe  $H$  è irraggiungibile dallo stato iniziale e viene pertanto rimosso. L'automa minimo è dato dalla tabella di transizione

	0	1
$A, G$	$B, F$	$A, G$
$B, F$	$A, G$	$C, E$
$C, E$	$D$	$B, F$
$D$	$D$	$A, G$

con  $A, G$  iniziale e  $D$  unico stato finale.

### 3.7 Generatori di analizzatori lessicali

Esistono molteplici strumenti per la realizzazione di analizzatori lessicali, tutti basati sulle tecniche che abbiamo delineato in questo capitolo. Si tratta di programmi che prendono come dati un insieme di pattern e restituiscono un programma che riconosce quei pattern nella sua stringa di ingresso. È possibile specificare che al riconoscimento di un determinato pattern sia eseguita una certa azione (per esempio, la sostituzione del lessema con un certo token), così da realizzare un analizzatore lessicale completo. Il cuore di questi strumenti è la costruzione del

DFA che riconosce i pattern, spesso con tecniche algoritmicamente più astute di quelle che abbiamo descritto. Di questi strumenti, uno dei più diffusi è Lex, disponibile su molte distribuzioni Unix (e su Linux nella sua versione *free* software flex). Descriveremo per sommi capi come descrivere un analizzatore lessicale con (f)Lex, rimandando al manuale per i dettagli.

Un sorgente Lex (cioè l'input di Lex) ha la seguente struttura generale:

```
definizioni
%%
regole
%%
funzioni ausiliarie
```

La parte concettualmente più importante di un sorgente sono le *regole*, cioè coppie espressione regolare      azione

dove azione è un frammento di codice C<sup>3</sup>, mentre espressione regolare può usare, oltre a quelli canonici che abbiamo introdotto nel Paragrafo 3.2.2, anche operatori addizionali, comuni ad altri comandi Unix (come ed o grep). Eseguendo flex su un sorgente della forma appena discussa viene generato un file lex.yy.c: si tratta di un programma C che implementa il DFA riconoscitore delle espressioni regolari contenute nelle regole. Il programma lex.yy.c filtra il proprio input: scandisce il testo alla ricerca di una stringa che corrisponda ad una delle espressioni regolari delle regole (cioè alla ricerca di un lessema). Quando trova un lessema, esegue l'*azione* specificata nella regola per quella espressione regolare, e passa in output il risultato dell'azione al posto del lessema. Passa in output inalterato tutto l'input che non corrisponde a nessuna espressione regolare.

### Esempio 3.35 Il seguente sorgente Lex

```
%%
a    printf("z");
bb*   printf("b");
cc    printf("c");
```

genera un programma lex.yy.c che, (compilato<sup>4</sup> ed) eseguito su una stringa sostituisce z al posto di a, una sola b al posto di una successione di b, una sola c al posto di una coppia di c e lascia inalterati tutti gli altri simboli. Ad esempio, sulla stringa abbbbbbbfgccc, restituisce zbgccc.

La regola può accedere al lessema attraverso yytext, un array di caratteri di lunghezza yyleng che contiene i simboli del lessema. In tal modo può costruire l'output in modo più sofisticato.

### Esempio 3.36 Il seguente sorgente Lex

<sup>3</sup>Esistono varianti di Lex capaci di generare codice in linguaggi diversi da C.

<sup>4</sup>Le modalità di compilazione dipendono ovviamente dall'architettura. Compilando con gcc è necessario includere alcune librerie: gcc lex.yy.c -l1.

```
cifra      [0-9]
cifre     [0-9]*
ide       [a-zA-Z] [a-zA-Z0-9]*
separatore [ \t\n]*
razionale {cifre}."{cifre}
bin        [+*]

%%
{cifre}          {printf("<NUM,%s>", yytext);}
{razionale}      {printf("<FRACT,%s>", yytext);}
":="|;"|while| "(" ")"|"{""}"
{ide}            {printf("<IDE,%s>", yytext);}
{separatore}    {printf("\n");}
{bin}|"-"
{printf("<OP2,%s>", yytext);}


```

Figura 3.10 Un semplice analizzatore lessicale in Lex.

```
%%
[0-9]    printf("*");
[a-z]+   printf("%c", yytext[yyleng-1]);
genera un programma che eseguito sulla stringa 12pippo45pappa2, restituisce
***o**a*.
```

La parte (opzionale) delle *definizioni* è una serie di definizioni regolari (vedi a pag. 55) nella forma

```
I1  espressione regolare1
I2  espressione regolare2
...
Ik  espressione regolarek
```

Ogni *I<sub>i</sub>* è un *nome* (identificatore che inizia con una lettera), che può essere usato nelle definizioni successive (con la sintassi {*I<sub>i</sub>*}) per denotare *espressione regolare<sub>i</sub>*.

Infine, la parte opzionale delle *funzioni ausiliarie* può contenere la definizione di funzioni da usare nelle azioni delle regole o anche la ridefinizione di funzioni usate da Lex che possono così essere personalizzate.

Abbiamo ora abbastanza elementi per mostrare in Figura 3.10 un sorgente Lex per un semplice analizzatore lessicale (che non contiene definizioni ausiliarie). Se lo eseguiamo sull'input

```
pigreco :=3.14;
temp1 := pigreco*(temp1+2);
temp2 := 6.0;
```

otteniamo in output la lista di token (formattata per comodità di lettura):

```
<IDE,pigreco><:=><FRACT,3.14><;>
```

## Lex e linguaggi non regolari

La possibilità che una regola Lex possa contenere codice arbitrario che viene inserito nel programma `lex.yy.c`, rende Lex molto flessibile. D'altra parte permette di generare programmi `lex.yy.c` che possono fare cose impossibili ad un DFA. Consideriamo il sorgente Lex

```
%{
    int par=0;
}

%%
\(
    {par++;}
\)
{par--;}
\n
{printf("%u", par);}

%
```

Se nella parte dedicata alle definizioni compare del testo racchiuso tra `%{ %}`, questo viene inserito inalterato in `lex.yy.c`. In questo caso, dichiariamo una variabile intera, che poi viene incrementata e decrementata se nell'input si incontrano, rispettivamente, parentesi aperte e chiuse (la barra \ è necessaria perché altrimenti le parentesi nelle espressioni regolari hanno il loro solito significato di raggruppamento). In corrispondenza di un carattere di *newline*, si stampa il valore di `par`. Il programma `lex.yy.c` può essere dunque usato come riconoscitore del linguaggio delle parentesi bilanciate (se `par` è 0, la stringa è bilanciata), un linguaggio non regolare (come dimostreremo nell'Esempio 3.39).

```
<IDE,temp1><::=><IDE,pigreco><OP2,*><(><IDE,temp1>
<OP2,+>2<) ><;>
<IDE,temp2><::=><FRACT,6.0><;>
```

Lex cerca sempre il lessema più lungo possibile che corrisponde a qualche espressione regolare. Quando più di un'espressione regolare può corrispondere all'input, Lex sceglie il match più lungo; a parità di lunghezza della stringa dell'input, ha preferenza la regola elencata per prima. Nell'esempio della Figura 3.10, l'input `whilefor` viene trasformato nel token `(IDE, whilefor)`, mentre `while` dà luogo a `(WHILE)`.

Per gestire il match in modo flessibile, Lex mette a disposizione altre due funzioni che permettono di intervenire su parte dell'input già riconosciuto. La funzione `yymore()` può essere invocata dentro un'azione quando si vuole estendere la stringa che ha causato il match: invocando `yymore()` la prossima porzione di input che effettuerà match sarà aggiunta allo `yytext` corrente (che sarebbe invece sovrascritto dal nuovo match in condizioni normali). Dualmente, `yyless(n)` viene usata per indicare che alcuni caratteri che hanno causato match non devono essere utilizzati e devono essere restituiti alla stringa di input. Il parametro intero `n` indica quanti caratteri di `yytext` devono essere conservati, mentre i restanti caratteri (che saranno `yylen - n`) sono restituiti all'input. Infine, altre funzioni permettono di accedere alle routine di I/O di Lex stesso: `input()` restituisce il prossimo carattere dell'input; `output(c)` scrive il carattere `c` sull'output; (iii)

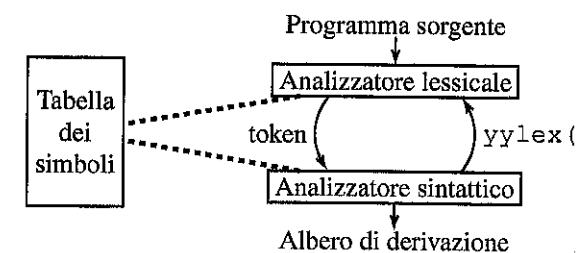


Figura 3.11 Rapporto tra parser e scanner.

`unput(c)` rimette il carattere `c` sull'input, in modo che possa essere letto da `input()` successive.

**L'operatore di lookahead** La sintassi delle espressioni regolari di Lex permette in alcune situazioni di specificare il carattere *che deve seguire* un certo pattern, senza per questo far parte del lessema. Ad esempio, l'espressione `r$` specifica i lessemi che corrispondono a `r`, purché si presentino al termine di una riga (ma il carattere *newline* non fa parte del lessema che viene così determinato). Analogamente, `^s` corrisponde ai match di `s`, purché si presentino ad inizio riga.

Questo modo di “guardare un carattere avanti” è presente nelle espressioni regolari di molti altri comandi Unix, ma ha in Lex una sua versione molto generale: l'operatore di lookahead `/`. Il pattern `s/t` specifica che il lessema deve essere un match per `s` e deve essere immediatamente seguito da un match per `t`; la parte dell'input che corrisponde a `t`, tuttavia, non fa parte del lessema. L'operatore di lookahead è molto espressivo e permette di gestire l'input in modo analogo, anche se formalmente diverso, a quello che è possibile attraverso l'uso di `yyless(n)` all'interno delle azioni.

### 3.7.1 Lex e i generatori di analizzatori sintattici

L'uso canonico di Lex nella generazione di un compilatore non è quello che abbiamo descritto sin qui. Come abbiamo già osservato nel riquadro di pag. 47, è in genere l'analizzatore sintattico a guidare l'analizzatore lessicale, chiedendo un token alla volta. La Figura 3.11 illustra questa architettura, evidenziando anche l'interazione con la tabella dei simboli. Esistono molti strumenti che permettono di generare analizzatori sintattici; l'analogo di Lex è Yacc (in una delle sue innumerevoli versioni), che discuteremo nel Paragrafo 4.10. I due strumenti sono pensati per essere usati insieme e generano codice che può condividere variabili e funzioni e quindi essere compilato assieme. Quando il codice prodotto da Lex è usato così, `lex.yy.c` è invocato da Yacc attraverso la funzione `yylex()` (che viene definita da Yacc e collegata con `lex.yy.c` al momento della compilazione congiunta). Le azioni definite nel sorgente Lex non agiscono allora sullo standard

```
%{
#define NUM 1
#define FRACT 2
#define IDE 3
#define OP2 4

#define PIU 41
#define PER 42
#define MEN 43
%}

cifra      [0-9]
cifre     [0-9]*
ide       [a-zA-Z][a-zA-Z0-9]*
separatore [ \t\n]*
razionale  {cifre}."{cifre}
bin        [+*]
%%
{separatore}  {}
{cifre}        {yyval = inConstTable(); return(NUM);}
{razionale}    {yyval = inConstTable(); return(FRACT);}
{ide}          {yyval = inSymbolTable(); return(IDE);}
+             {yyval = PIU; return(OP2);}
*             {yyval = PER; return(OP2);}
-             {yyval = MEN; return(OP2);}

int inConstTable() /*Converte la stringa puntata da yytext in
                   uno (NUM) o due (FRACT) interi, li inserisce
                   nella tabella delle costanti e restituisce
                   (come int) un puntatore all'elemento
                   inserito*/
{
int inSymbolTable() /*Inserisce nella tabella dei simboli la
                     stringa puntata da yytext e restituisce (come
                     int) un puntatore all'elemento inserito*/
}
```

**Figura 3.12** Un programma Lex che interagisce con Yacc.

output, ma restituiscono valori e controllo al chiamante (cioè l'analizzatore sintattico). La funzione `yylex()` restituisce in genere il nome del token, mentre il valore del token viene condiviso attraverso altre variabili, tra cui `yyval` di tipo `int`. Inoltre, le azioni dello scanner possono iniziare a riempire la tabella dei simboli, che poi sarà utilizzata (e ulteriormente aggiornata di informazioni) dal parser. La Figura 3.12 presenta un semplice programma Lex che realizza questo schema di interazione.

### 3.8 Dimostrare che un linguaggio non è regolare

Fin'ora non abbiamo *dimostrato* che esistono linguaggi non regolari. D'altra parte, se cercassimo un'espressione regolare (o un automa finito, o una grammatica regolare) per il linguaggio delle palindrome, col quale abbiamo iniziato il nostro incontro con le grammatiche nell'Esempio 2.1, dopo un po' di tentativi falliti ci convinceremmo che le espressioni regolari sono troppo povere per descrivere linguaggi del genere. Mostreremo in questo paragrafo un'importante proprietà di ogni linguaggio regolare, il che ci permetterà di dimostrare che alcuni linguaggi *non* sono regolari: se tutti i linguaggi regolari hanno quella proprietà e un certo linguaggio *non* possiede la proprietà, quel linguaggio non può essere regolare.

La proprietà in questione è quella espressa dal teorema che segue, che è tradizionalmente indicato come “lemma del pompaggio” (*pumping lemma*), perché mostra che in ogni linguaggio regolare infinito tutte le stringhe sufficientemente lunghe possono essere arbitrariamente “pompate” (cioè allungate o accorciate ripetendo più volte, o cancellando, una loro sottostringa) restando all'interno del linguaggio.

**Teorema 3.37 (Pumping lemma, PL)** *Sia  $L$  un linguaggio regolare. Esiste una costante  $N > 0$  tale che, per ogni  $z \in L$  con  $|z| \geq N$ , possiamo suddividere  $z$  in tre sottostringhe  $z = uvw$  tali che*

- $|uv| \leq N$ ;
- $|v| \geq 1$ ;
- per ogni  $k \geq 0$ ,  $uv^k w \in L$ .*

*Inoltre  $N$  è minore o uguale del numero di stati del DFA minimo che accetta  $L$ .*

**Dim.** Sia  $M$  un DFA che accetta  $L$  (possiamo prenderlo minimo, per l'ultima parte dell'enunciato) e sia  $N$  il numero degli stati di  $M$  (cioè  $N = |Q_M|$ ). Prendiamo una generica stringa  $z = a_1 \dots a_m \in L$ , con  $m \geq N$ , e consideriamo l'elenco ordinato degli  $m + 1$  stati attraverso i quali passa  $M$  durante il riconoscimento di  $z$ . Consideriamo i primi  $N + 1$  stati di tale elenco (siccome  $m \geq N$ , si ha anche  $m + 1 \geq N + 1$ ):  $q_0, \dots, q_N$ . Dal momento che  $m \geq N = |Q_M|$ , questi  $N + 1$  stati non possono essere tutti diversi tra loro<sup>5</sup>. Vi sono dunque nell'elenco due stati  $q_i$  e  $q_j$  con  $i < j$  e  $q_i = q_j$ . La Figura 3.13 illustra la situazione e lo svolgimento successivo della dimostrazione. Suddividiamo allora  $z$  in questo modo:  $u = a_1 \dots a_i$ ;  $v = a_{i+1} \dots a_j$ ;  $w = a_{j+1} \dots a_m$ . La condizione  $|uv| \leq N$  è soddisfatta, perché  $i$  e  $j$  sono scelti nei primi  $N + 1$  stati (coi quali si consumano  $N$  simboli di input); la condizione  $|v| \geq 1$  è verificata perché  $i \neq j$ .

Dobbiamo ora mostrare che  $v$  può essere arbitrariamente “pompata”, cioè la condizione (iii). Dal momento che  $q_i = q_j$ , il cammino accettante di  $M$  su  $z$  contiene un ciclo: da  $q_i$  si torna a  $q_i$  consumando la stringa  $v$ . Ma è evidente che essendo  $M$  deterministico, si ottengono altri cammini accettanti anche se questo

<sup>5</sup>Questo fatto è spesso chiamato “princípio della piccionaia” (*pigeonhole principle*): se si hanno  $N + 1$  piccioni e  $N$  piccionaie, vi sarà almeno una piccionaia che contiene (almeno) due piccioni.

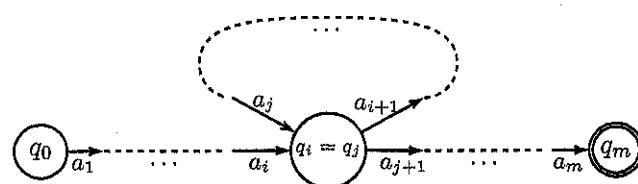


Figura 3.13 La computazione del pumping lemma.

ciclo è percorso zero volte (e dunque consumando l'input  $uv^0w = uv$ ) oppure un numero qualsiasi  $k \geq 1$  di volte (consumando l'input  $uv^k w$ ). In tutti questi casi, quando nello stato  $q_j$  l'automa incontrerà il primo simbolo  $a_{j+1}$  di  $w$ , proseguirà in  $q_{j+1}$ , per ricalcare da lì in poi il cammino seguito per  $w$ . E dunque terminerà in uno stato finale. Abbiamo dunque dimostrato che per ogni  $k \geq 0$ , la stringa  $uv^k w \in L$ , cioè la tesi.  $\square$

Il lemma è molto utile per dimostrare che alcuni linguaggi non sono regolari. Nei prossimi esempi vedremo come possiamo usarlo a questo scopo.

**Esempio 3.38** Il linguaggio  $L = \{a^n b^n \mid n \geq 0\}$  non è regolare.

La dimostrazione procede supponendo che  $L$  sia regolare, e utilizzando il pumping lemma (PL) per mostrare che allora apparterrebbero a  $L$  stringhe che, invece, certamente non vi appartengono. Supponiamo dunque che  $L$  sia regolare: sia  $N$  la costante di cui il lemma asserisce l'esistenza (la "costante del PL"). Per ogni stringa  $z$  che appartiene al linguaggio e che è sufficientemente lunga ( $|z| \geq N$ ), il PL asserisce l'esistenza di una scomposizione di  $z$  che permette pompaggi arbitrari. Siccome vogliamo dimostrare che in realtà  $L$  non è regolare, quello che dobbiamo fare è cercare un controsenso, cioè una stringa  $z$  di almeno  $N$  simboli che non possa essere mai scomposta in modo da soddisfare la conclusione del lemma. Dobbiamo prima "inventare" il controsenso, e poi argomentare che per ogni possibile scomposizione la tesi del lemma verrebbe contraddetta.

Scegliamo  $z = a^N b^N$  (si tratta di una stringa fissata: quella di lunghezza  $2N$ , per  $N$  costante del PL). Supponiamo ora che  $z$  possa essere scomposta come  $z = uvw$ : non sappiamo dove il PL farebbe iniziare  $v$  e  $w$ , ma sappiamo che questa scomposizione rispetta i vincoli (i) e (ii):  $|uv| \leq N$  e  $|v| \geq 1$ . La prima condizione, nel caso della nostra stringa  $z$ , implica che  $uv$  è composta da soli  $a$ , perché  $z$  inizia con  $N$  simboli  $a$  consecutivi. Dunque anche  $v$  è composta di soli  $a$ , ed ha almeno un  $a$ , per via della condizione (ii). In modo formale,  $u = a^i$  e  $v = a^j$ , con  $i + j \leq N$ ,  $i \geq 0$  e  $j \geq 1$ . Prendiamo ora in considerazione un qualsiasi pompaggio  $k \neq 1$ ; per esempio  $k = 2$ . Il PL ci dice che  $uv^2 w \in L$ , ma  $uv^2 w$  contiene certo  $N + j$  simboli  $a$ , con  $j \geq 1$ , mentre i simboli  $b$  sono rimasti  $N$ . Ne segue che  $uv^2 w$  non può appartenere a  $L$ , perché  $L$  è per definizione il linguaggio  $\{a^n b^n \mid n \geq 0\}$ . Ne segue che non potevamo applicare il PL a  $L$ , ovvero  $L$  non può essere regolare.

IL PL	LA NEGAZIONE DEL PL
Esiste $N \geq 0$	Per ogni $N$
Per ogni $z$ tale che	Esiste $z$ tale che
$ z  \geq N$	$ z  \geq N$
Esistono $u,v,w$ tali che	Per ogni $u,v,w$ tali che
(i) $ uv  \leq N$	(i) $ uv  \leq N$
(ii) $ v  \geq 1$	(ii) $ v  \geq 1$
Per ogni $k \geq 0$	Esiste $k \geq 0$
$uv^k w \in L$	$uv^k w \notin L$

Figura 3.14 La struttura logica dell'enunciato del pumping lemma.

In questo caso, *ogni* pompaggio  $k \neq 1$  sarebbe andato bene; in altri casi, solo alcuni valori di  $k$  potrebbero portarci fuori dal linguaggio. È sufficiente mostrare che esiste almeno un pompaggio che porta fuori del linguaggio per poter concludere che quel linguaggio non è regolare.

**Esempio 3.39** L'Esempio 3.38 è il capostipite di molti altri dello stesso tono. Un linguaggio importante è quello delle parentesi bilanciate. Formalmente,  $Pb$  è il linguaggio sull'alfabeto  $\{(, )\}$  costituito da quelle stringhe per le quali: (i) il numero di parentesi aperte è uguale a quello delle parentesi chiuse; (ii) in ogni loro sottostringa, il numero di parentesi aperte è maggiore o uguale di quello delle parentesi chiuse. Supponendo che  $Pb$  sia regolare, sia  $N$  la costante del PL. Scegliamo  $z = {}^{(N)}{}^N \in Pb$ . Esattamente lo stesso ragionamento dell'esempio precedente mostra che  $z$  non ammette pompaggi della forma prescritta dal PL e che rimangano all'interno di  $Pb$ . Dunque  $Pb$  non è regolare.

Il principiante trova spesso qualche difficoltà ad applicare correttamente il PL. Una delle difficoltà è costituita dalla struttura logica dell'enunciato, che alterna quantificatori esistenziali ("esiste un  $N$ ") e quantificatori universali ("per ogni stringa  $z$ "). La Figura 3.14 sintetizza la struttura dell'enunciato del PL (a sinistra) e della sua negazione (a destra).

Una dimostrazione che usa il PL per mostrare che un linguaggio  $L$  non è regolare, può essere schematizzata come un contraddirittorio tra il PL e noi stessi che cerchiamo di dimostrare che  $L$  non è regolare. Le mosse dei due contendenti sono scandite dai quantificatori: PL "gioca" sui quantificatori esistenziali, mentre noi "giochiamo" in corrispondenza dei quantificatori universali. Siccome vogliamo ottenere una contraddizione, le mosse di PL sono quelle di ipotizzare l'esistenza di un parametro ( $N$ , la scomposizione  $uvw$ ), che noi dobbiamo usare senza potervi fare ipotesi aggiuntive se non quelle date dai vincoli (i) e (ii). Quando invece è il nostro turno (giochiamo su un "per ogni" che dobbiamo falsificare: dobbiamo cioè produrre un controsenso), possiamo scegliere il parametro (la stringa  $z$ , il pompaggio  $k$ ) che meglio serve al nostro scopo. La Figura 3.15 schematizza in questo modo la dimostrazione dell'Esempio 3.38; sono indicate con  $\bullet$  le nostre

IL PL	LA DEMOSTRAZIONE CHE $L$ NON È REGOLARE
Esiste $N$	Sia $N \geq 0$ qualsiasi, ma fissato
Per ogni $z$ tale che $ z  \geq N$	• Scegliamo uno specifico $z$ tale che $ z  \geq N : z = a^N b^N$
Esistono $u, v, w$ tali che (i) $ uv  \leq N$ (ii) $ v  \geq 1$	Per ogni possibile scomposizione $u, v, w$ tale che (i) $ uv  \leq N$ (ii) $ v  \geq 1$
Per ogni $k \geq 0$ $uv^k w \in L$	• Scegliamo uno specifico $k \geq 0$ tale che $uv^k w \notin L : k = 2$ .

Figura 3.15 La dimostrazione che  $L = \{a^n b^n \mid n \geq 0\}$  non è regolare.

mosse, nelle quali sceglio il valore di uno specifico parametro. Tutta la difficoltà sta nello scegliere  $z$  (che dipenderà da  $N$ ) in modo che, senza ipotizzare su  $uvw$  null'altro che (i) e (ii), si possa poi scegliere un pompaggio  $k$  che falsifichi l'appartenenza a  $L$ . In genere quest'ultimo passo richiede di fare delle considerazioni (spesso per casi) sulla forma che possono assumere  $u$  e  $v$ , sfruttando (i) e (ii) (nell'esempio, osserviamo che  $uv$  è necessariamente composta solo da  $a$ ).

**Esempio 3.40** Dimostriamo che  $L = \{1^n \mid n \text{ è un quadrato perfetto}\}$  non è regolare, seguendo lo schema della Figura 3.14. Se  $L$  fosse regolare, avremmo  $N$ , costante del PL. Scegliamo  $z = 1^{N^2}$ . Sia ora  $z = uvw$  una suddivisione arbitraria, ma che soddisfa (i) e (ii). Certo  $v$  è fatta solo di 1 e vale  $1 \leq |v| \leq N$ . Consideriamo il pompaggio  $k = 2$ , cioè la stringa  $uvvw$ . Vogliamo mostrare che non può appartenere a  $L$ . Il quadrato perfetto immediatamente più grande di  $N^2$  è  $(N+1)^2 = N^2 + 2N + 1$ . D'altra parte, per quanto abbiamo detto su  $v$ , si ha  $|uvvw| \leq 2N + N < N^2 + 2N + 1$ . Dunque la stringa  $uvvw$  ha una lunghezza che non è un quadrato perfetto, e non può appartenere a  $L$ . Dunque  $L$  non è regolare.

Concludiamo osservando che la proprietà espressa dal PL è solo una proprietà *necessaria* di ogni linguaggio regolare. L'implicazione del lemma non può essere invertita: ci sono linguaggi che soddisfano la proprietà espressa dal PL ma che *non* sono regolari. Per trattare la non regolarità di linguaggi siffatti occorrono strumenti più selettivi del PL, che non possono essere discussi in questa sede.

### 3.9 Sommario del capitolo

In questo capitolo abbiamo trattato i principali argomenti teorici e implementativi collegati alla costruzione di un analizzatore lessicale.

- Il concetto di *token* e il suo ruolo in un compilatore.
- Le *espressioni regolari* e i linguaggi ad esse associati.

- Gli *automi finiti non deterministici (NFA)* e *deterministici (DFA)*, discutendone il ruolo come riconoscitori di linguaggi. Abbiamo dimostrato che ogni NFA può essere effettivamente trasformato in un DFA che riconosce lo stesso linguaggio.
- Abbiamo mostrato come ad ogni espressione regolare possiamo associare un NFA che riconosce lo stesso linguaggio associato all'espressione regolare.
- Le *grammatiche regolari*, una restrizione delle classi delle grammatiche libere; abbiamo mostrato che ogni DFA può essere trasformato in una grammatica regolare per lo stesso linguaggio. Abbiamo infine argomentato come trasformare ogni grammatica regolare in un'espressione regolare per lo stesso linguaggio.
- Come conseguenza abbiamo stabilito l'*equivalenza delle principali nozioni introdotte in questo capitolo*. Grammatiche regolari, automi finiti non deterministici, automi finiti deterministici, grammatiche regolari definiscono tutti la stessa classe di linguaggi: i *linguaggi regolari*.
- *Lex*: un generatore di analizzatori lessicali.
- Il *pumping lemma*, un'importante proprietà goduta da ogni linguaggio regolare, che può essere utilizzata per dimostrare che alcuni linguaggi *non* sono regolari.

### 3.10 Nota bibliografica

La teoria degli automi finiti e la loro applicazione all'elaborazione di stringhe è uno dei più interessanti contributi dell'informatica teorica degli anni sessanta. Un testo canonico che approfondisce la parte teorica è [45], mentre sul progetto di analizzatori lessicali si può vedere [4]; [55] è un manuale per Lex.

### 3.11 Esercizi

1. Fissiamo  $\Sigma = \{0, 1\}$ . Usare il pumping lemma per dimostrare che i seguenti linguaggi su  $\Sigma$  non sono regolari:
  - $L_1 = \{ww \mid w \in \Sigma^*\};$
  - $L_2 = \{w \mid w \text{ ha lo stesso numero di } 0 \text{ e } 1\};$
  - $L_3 = \{ww^R \mid w \in \Sigma^*\}$  ( $w^R$  è la stringa  $w$  rovesciata da destra a sinistra);
  - $L_4 = \{0^n 1^m \mid n \geq m \geq 0\};$
  - $L_5 = \{1^m \mid m \text{ è un numero primo}\}.$
2. Si consideri il DFA la cui funzione di transizione è data dalla tabella seguente

	0	1
A	B	A
B	A	C
C	D	B
D	D	A
E	D	F
F	G	E
G	F	G
H	G	D

- A è lo stato iniziale; D quello finale.
- Dire quali stati del DFA sono indistinguibili.
  - Si dia la tabella di transizione dell'automa minimo equivalente (gli stati di tale automa sono insiemi di stati indistinguibili).
3. Si dia il NFA canonico corrispondente all'espressione regolare  $ab^*(a \mid b)^*$ .
4. Date due stringhe  $x = a_1 \dots a_n$ ,  $y = b_1 \dots b_k$  su  $\Sigma = \{0, 1\}$ , si definisca  $x \bullet y = b_1 \dots b_k a_{k+1} \dots a_n$  (nel caso in cui  $n \leq k$  si ha quindi  $x \bullet y = y$ ). Si classifichi il linguaggio  $A = 0^* \bullet \{0^n 1^n \mid n \geq 0\}$ .

## Analisi sintattica: linguaggi liberi

Vedremo in questo capitolo alcuni elementi della teoria dei linguaggi libri da contesto, in particolare gli aspetti collegati alla costruzione di riconoscitori. Scopriremo un mondo più complesso di quello dei linguaggi regolari. Innanzitutto ci verrà a mancare una notazione sintetica ed espressiva come quella delle espressioni regolari, che non ha corrispettivi nel caso dei linguaggi libri. Studieremo una generalizzazione degli automi finiti (gli *automi a pila*) che corrisponde esattamente alla classe dei linguaggi libri, ma perderemo l'equivalenza tra la versione deterministica e quella non deterministica. Ciò nonostante, i linguaggi di programmazione ammettono per la gran parte riconoscitori deterministici e dedicheremo tutta la seconda parte del capitolo alle tecniche di generazione di automi riconoscitori, cioè di analizzatori sintattici. Tratteremo di due grandi famiglie: i riconoscitori *top down*, per i quali introduciamo la classe delle grammatiche  $LL(k)$ , e quelli *bottom up*, che corrispondono alle grammatiche  $LR(k)$ . Le grammatiche LALR (una sottoclasse delle  $LR(k)$ ) sono quelle utilizzate dai principali strumenti automatici per la generazione di analizzatori sintattici, di cui discuteremo il capostipite, YACC.

### 4.1 Linguaggi, derivazioni e alberi

Questo capitolo tratta dei linguaggi che possono essere generati dalle grammatiche libere da contesto (Definizione 2.2):

**Definizione 4.1** Un linguaggio  $L$  è libero da contesto se esiste una grammatica libera da contesto  $G$  con  $L = \mathcal{L}[G]$ .

Siccome ogni grammatica regolare (Definizione 3.22) è certo libera, i linguaggi regolari sono inclusi nei linguaggi libri da contesto. Si tratta di un'inclusione propria, perché il linguaggio  $L = \{a^n b^n \mid n \geq 0\}$  non è regolare (Paragrafo 3.8), ma è libero da contesto in quanto generato dalla grammatica

$$A \rightarrow aAb \mid \epsilon$$

Abbiamo dato nelle Definizioni 2.5 e 2.7 le nozioni di derivazione e di albero di derivazione, che ci sono servite per quella di ambiguità (Definizione 2.10).

Una derivazione si dice *sinistra* (rispettivamente: *destra*) se ad ogni passo viene riscritto il non terminale più a sinistra (a destra) della stringa. Un passo di una derivazione sinistra ha dunque la forma  $uA\alpha \Rightarrow u\beta\alpha$ , con  $u \in T^*$ ,  $A \in NT$  e  $\alpha, \beta \in (T \cup NT)^*$ ; la stringa  $u$  è il *prefisso* del passo di derivazione. La nozione di *suffisso* per un passo di derivazione destra è simmetrica. In una derivazione sinistra (o destra), il non terminale da riscrivere è dunque univocamente determinato, mentre rimane la scelta su *quale produzione* utilizzare se quel non terminale ne ammette più di una. È facile osservare che derivazioni sinistre (o destre) e alberi di derivazione sono in corrispondenza biunivoca: se due derivazioni sinistre per lo stesso non terminale sono diverse, allora ad un certo punto al non terminale più a sinistra si sono applicate due diverse produzioni, e questo si traduce in due alberi di derivazione distinti. E il ragionamento è invertibile: da due alberi di derivazione distinti si estraggono due derivazioni sinistre (destre) distinte, agendo sempre sul nodo più a sinistra nell'albero. Abbiamo dunque stabilito

**Teorema 4.2** Una grammatica  $G$  è ambigua se e solo esiste una stringa  $v \in \mathcal{L}[G]$  che ammette due diverse derivazioni sinistre (destre) dal simbolo iniziale.

Concludiamo con un paio di definizioni che useremo nel seguito. Una *forma sentenziale* di una grammatica  $G$  è una stringa  $\alpha \in (T \cup NT)^*$  tale che  $S \Rightarrow^* \alpha$ , dove  $S$  è il simbolo iniziale di  $G$ . Una forma sentenziale è una *sentenza* di  $G$  se è composta da soli terminali. Possiamo dunque dire, in modo equivalente alla Definizione 2.6, che il linguaggio generato da una grammatica è l'insieme delle sue sentenze.

## 4.2 Automi a pila

Prendiamo il linguaggio libero e non regolare  $\{a^n b^n \mid n \geq 0\}$ . Un automa finito non è in grado di riconoscerlo, perché dovrebbe essere in grado, consumando le  $a$ , di contare il loro numero, così da confrontarlo col numero delle  $b$ . Siccome  $L$  contiene stringhe con un numero arbitrario di  $a$ , questo non può essere fatto con un numero finito di stati. Gli automi riconoscitori dei linguaggi liberi, dunque, devono avere qualche forma di memorizzazione, ma questa possibilità di ricordare (o contare) non può essere illimitata, perché altrimenti potremmo rischiare di riconoscere linguaggi che non sono nemmeno liberi da contesto.<sup>1</sup> La nozione giusta è quella di un automa che, oltre all'input e ad un numero finito di stati, ha a disposizione anche una pila, nella quale può essere memorizzato un numero arbitrario di simboli secondo la disciplina *last in first out*. Una transizione dipende non solo dall'input e dallo stato, ma anche dal simbolo che si trova in cima alla pila, che viene rimosso. Per effetto della transizione l'automa oltre a passare in un nuovo stato (come nel caso di un NFA) può anche mettere alcuni simboli

sulla pila. Nella definizione seguente con  $\mathcal{P}_{fin}(X)$  indichiamo la collezione dei sottoinsiemi finiti dell'insieme  $X$ .

**Definizione 4.3** Un *automa a pila non deterministico* (push-down automaton, per brevità: PDA) è una 7-upla  $(\Sigma, Q, \Gamma, \delta, q_0, \perp, F)$  dove

1.  $\Sigma$  è un alfabeto finito (i simboli di input);
2.  $Q$  è un insieme finito (gli stati);
3.  $\Gamma$  è un insieme finito (i simboli della pila);
4.  $\delta$  è la funzione di transizione con tipo

$$\delta : Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \rightarrow \mathcal{P}_{fin}(Q \times \Gamma^*);$$

5.  $q_0 \in Q$  (lo stato iniziale);
6.  $\perp \in \Gamma$  (il simbolo di fondo pila);
7.  $F \subseteq Q$  (gli stati finali).

Per descrivere il funzionamento di un PDA è utile avere un modo compatto per fotografare un istante di computazione. Mentre in un NFA basta dare lo stato e l'input ancora da consumare, in un PDA occorre conoscere anche la situazione della pila. Una *configurazione* (o descrizione istantanea) di un PDA è una tripla  $(q, w, \beta)$  con  $q \in Q$  (lo stato in cui il PDA si trova),  $w \in \Sigma^*$  (l'input ancora non letto) e  $\beta \in \Gamma^*$  (la stringa sulla pila: per convenzione la cima è il simbolo a sinistra). Possiamo allora descrivere un passo di computazione (o mossa) come una transizione tra due configurazioni. Siano ora  $a \in \Sigma$  e  $X \in \Gamma$ ; se un PDA  $P$  si trova nella configurazione  $(q, aw, X\beta)$  (cioè il prossimo input è  $a$  e la cima della pila è  $X$ ), la funzione di transizione  $\delta(q, a, X)$  dà un insieme finito di coppie della forma  $(q', \alpha)$ ; scelta una di queste coppie,  $P$  transita nel nuovo stato  $q'$ , consuma il simbolo  $a$ , rimuove  $X$  dalla pila e lo sostituisce con la stringa  $\alpha$ . In simboli,

$$(q, aw, X\beta) \vdash_P (q', w, \alpha\beta) \quad \text{per } (q', \alpha) \in \delta(q, a, X).$$

Se  $I$  e  $J$  sono due configurazioni, scriveremo  $I \vdash_P^* J$  se  $J$  si ottiene da  $I$  con zero o più mosse.

Per determinare il funzionamento di un PDA come riconoscitore di linguaggi, rimane da definire come un automa a pila accetta il proprio input. Per analogia con gli automi finiti, possiamo stabilire di accettare quando l'automa ha consumato tutto l'input ed è in uno stato finale, indipendentemente da cosa sia rimasto sulla pila. Oppure, possiamo accettare nel momento in cui la pila venga completamente svuotata (cioè sia rimosso anche il simbolo speciale di fondo pila), indipendentemente dallo stato finale.

**Definizione 4.4** Sia  $P = (\Sigma, Q, \Gamma, \delta, q_0, \perp, F)$  un PDA.

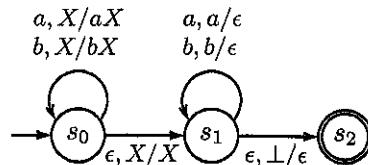
(i) Il linguaggio accettato per stato finale da  $P$  è

$$\mathcal{L}[P] = \{w \in \Sigma^* \mid (q_0, w, \perp) \vdash_P^* (q, \epsilon, \alpha) \text{ con } q \in F\};$$

(ii) Il linguaggio accettato per pila vuota da  $P$  è

$$\mathcal{N}[P] = \{w \in \Sigma^* \mid (q_0, w, \perp) \vdash_P^* (q, \epsilon, \epsilon)\}.$$

<sup>1</sup> Mostreremo nel Paragrafo 4.3 che tali linguaggi esistono:  $\{a^n b^n c^n \mid n \geq 0\}$  è un linguaggio non libero (Esempio 4.13).



**Figura 4.1** Un PDA per  $\{ww^R \mid w \in \Sigma^*\}$ .

**Esempio 4.5** Fissato  $\Sigma = \{a, b\}$ , sia  $L_{wur} = \{ww^R \mid w \in \Sigma^*\}$ , generato dalla grammatica

$$A \rightarrow aAa \mid bAb \mid \epsilon$$

Non è difficile dimostrare che  $L_{wur}$  non è regolare.<sup>2</sup> Un automa a pila per  $L_{wur}$  potrebbe iniziare a consumare il proprio input, inserendolo via via sulla pila. Non deterministicamente l'automa può decidere di iniziare a svuotare la pila (supponendo di trovarsi a metà della stringa di input), confrontando via via i simboli sull'input con quelli sulla pila. Se riesce a consumare tutto l'input, la stringa è della forma richiesta.

La Figura 4.1 presenta un PDA  $P$  con  $\Sigma = \{a, b\}$  e  $\Gamma = \{a, b, \perp\}$  che accetta  $L_{wur}$  sia per stato finale che per pila vuota, cioè  $L_{wur} = \mathcal{L}[P] = \mathcal{N}[P]$ . Nei diagrammi dei PDA etichettiamo l'arco  $(q, q')$  con  $a, Z/\beta$  se  $(q', \beta) \in \delta(q, a, Z)$ . Inoltre, usiamo la metavariable  $X$  per indicare un generico simbolo della pila: ad esempio nell'automa della figura l'etichetta  $a, X/aX$  sta per le tre etichette (e tre archi)  $[a, \perp/a\perp], [a, a/aa], [a, b/ba]$ .

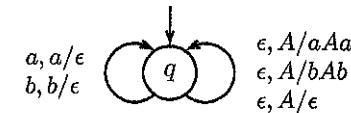
A differenza del PDA dell'esempio, in generale  $\mathcal{L}[P] \neq \mathcal{N}[P]$ . Come esempio banale, prendiamo il PDA della Figura 4.1 e semplicemente dichiariamo che lo stato  $s_2$  non è più finale (cioè prendiamo un automa  $P'$  in tutto identico, ma con insieme di stati finali vuoto). Ovviamente  $\mathcal{L}[P'] = \emptyset$ , mentre  $\mathcal{N}[P'] = \mathcal{N}[P] = L_{wur}$ . Tuttavia, se siamo interessati alla classe dei linguaggi accettati dall'insieme di tutti i PDA, questa non varia a seconda del tipo di accettazione. Vale infatti il seguente

**Teorema 4.6** (i) Se  $L = \mathcal{N}[P]$ , possiamo costruire un PDA  $P'$  con  $L = \mathcal{L}[P']$ .  
(ii) Se  $L = \mathcal{L}[P]$ , possiamo costruire un PDA  $P'$  con  $L = \mathcal{N}[P']$ .

**Approfondimento** 4.1

Il risultato più importante sulle relazioni tra PDA e linguaggi è che la loro capacità di accettazione coincide esattamente con i linguaggi libri da contesto.

<sup>2</sup>Si usa il PL, con la stringa  $a^N b^N b^N a^N$ .



**Figura 4.2** Un altro PDA che riconosce  $\{ww^R \mid w \in \Sigma^*\}$ .

**Esempio 4.7** Dalla grammatica per  $L_{wur}$  dell'Esempio 4.5

$$A \rightarrow aAa \mid bAb \mid \epsilon$$

possiamo ottenere un altro PDA, che accetta  $L_{wur}$  per pila vuota, in Figura 4.2. Si osservi come le etichette del ciclo a destra corrispondano in modo canonico alle produzioni della grammatica, mentre il ciclo a sinistra ha un'etichetta per ogni simbolo terminale. La costruzione di un PDA con un solo stato e con transizioni che corrispondono alle produzioni della grammatica è del tutto generale ed è un ingrediente primario nella dimostrazione del seguente importante risultato.

**Teorema 4.8** Un linguaggio  $L$  è libero da contesto sse è accettato da un PDA.

**Approfondimento** 4.2

È naturale a questo punto chiedersi se possiamo eliminare il non determinismo dai PDA. Iniziamo col chiarire quando un PDA è deterministico.

**Definizione 4.9** Un PDA  $P = (\Sigma, Q, \Gamma, \delta, i, \perp, F)$  è deterministico sse

1. per ogni  $q \in Q$  e  $Z \in \Gamma$ , se  $\delta(q, \epsilon, Z) \neq \emptyset$ , allora  $\delta(q, a, Z) = \emptyset$  per ogni  $a \in \Sigma$ ;
2. per ogni  $q \in Q$ ,  $Z \in \Gamma$  e  $a \in \Sigma \cup \{\epsilon\}$ ,  $|\delta(q, a, Z)| \leq 1$ .

Diremo che un linguaggio è *libero deterministico* se è accettato per stato finale da un PDA deterministico. A differenza degli automi finiti, un PDA non deterministico non è sempre equivalente ad un automa deterministico. Diamo il seguente enunciato senza dimostrazione.

**Teorema 4.10** La classe dei linguaggi libri deterministici è inclusa propriamente nella classe dei linguaggi libri. Ad esempio, il linguaggio libero  $L_{wur} = \{ww^R \mid w \in \Sigma^*\}$  non è accettato da alcun PDA deterministico.

**Esempio 4.11** L'essenziale non determinismo di  $L_{wur}$  viene dalla necessità di "indovinare" dove si trova il centro della stringa (cioè quando l'automa della Figura 4.1 deve transitare dallo stato  $s_0$  ad  $s_1$ ). Consideriamo il linguaggio sull'alfabeto  $\{a, b, c\}$  che si ottiene esplicitando il centro della stringa palindroma:

$L' = \{wcw^R \mid w \in \Sigma^*\}$ .  $L'$  è un linguaggio non regolare libero deterministico, come si dimostra facilmente transformando il PDA della Figura 4.1 in un PDA deterministico per  $L'$  (basta modificare la transizione da  $s_0$  ad  $s_1$ , etichettandola con  $c, X/X$ ).

### 4.3 Dimostrare che un linguaggio non è libero

Il pumping lemma per i linguaggi regolari è un caso particolare di un risultato più generale (e un po' più complesso) che vale per i linguaggi liberi. Come il suo analogo per i regolari, anche il pumping lemma per i libri è un importante strumento per dimostrare che una larga classe di linguaggi non è libera da contesto.

**Teorema 4.12 (Pumping lemma, PL)** *Sia  $L$  un linguaggio libero da contesto. Esiste una costante  $N > 0$  tale che, per ogni  $z \in L$  con  $|z| \geq N$ , possiamo suddividere  $z$  in cinque sottostringhe  $z = uvwxy$  tali che*

- (i)  $|vwx| \leq N$ ;
- (ii)  $|vx| \geq 1$ ;
- (iii) per ogni  $k \geq 0$ ,  $uv^kwx^ky \in L$ .

Approfondimento

4.3

**Esempio 4.13** Il linguaggio  $L = \{a^n b^n c^n \mid n \geq 0\}$  non è libero.

La dimostrazione procede supponendo che  $L$  sia libero, utilizzando quindi il PL per ottenere una contraddizione. Sia dunque  $N$  la costante di cui il PL asserisce l'esistenza. Dobbiamo esibire (scegliere) una stringa  $z \in L$  che, comunque suddivisa in accordo ai vincoli del PL, abbia almeno un pompaggio che la porta fuori di  $L$ . Sceglieremo  $z = a^N b^N c^N$ . Sia ora  $x = uvwxy$  una suddivisione qualsiasi di  $z$ , con il vincolo che  $|vx| \geq 1$  e  $|vwx| \leq N$ . Siccome  $|vwx| \leq N$ , le due estremità di questa sottostringa non possono essere una  $a$  e l'altra  $c$ . Si presentano dunque solo i seguenti casi:

1.  $vwx$  non contiene  $c$  (è fatta solo di  $a$  e/o di  $b$  e i  $c$  stanno tutti in  $y$ ): in tal caso, siccome almeno uno tra  $v$  e  $x$  non sono vuote ( $|vx| \geq 1$ ), ogni pompaggio di  $v$  e  $x$  modifica il numero di  $a$  e/o di  $b$ , ma lascia invariato il numero di  $c$ . Dunque, per  $k \neq 1$ ,  $uv^kwx^ky \notin L$ ;
2.  $vwx$  non contiene  $a$ : analogamente al caso (1).

Possiamo dunque concludere che  $L$  non è libero, perché abbiamo trovato una stringa di  $L$  che non ammette sottostringhe che rispettino i vincoli del PL e che possano essere pompatate.

Un altro interessante linguaggio che non è libero da contesto è  $L = \{ww \mid w \in \{a, b\}^*\}$ .<sup>3</sup> Si tratta di un linguaggio importante per i progettisti di linguaggi di programmazione perché identifica una situazione molto comune. In  $L$  infatti una sottostringa può comparire solo se *prima* essa è già comparsa. Nell'astrattezza dei linguaggi formali,  $L$  esprime una situazione tipica: un identificatore può essere usato solo dopo essere stato dichiarato. Questo vincolo (che nel Paragrafo 2.3 abbiamo chiamato non a caso "contestuale", o di "semantica statica") può essere espresso mediante linguaggi simili a  $L$ . Siccome  $L$  non è libero, tuttavia, non possiamo dare grammatiche (libere) per questi linguaggi (si veda il Paragrafo 4.4). È per questo che un compilatore verifica questi vincoli contestuali non mediante derivazioni grammaticali, ma attraverso controlli *ad hoc*.

### 4.4 Oltre i linguaggi libri

Nei nostri percorsi abbiamo iniziato con le grammatiche libere da contesto, per poi identificare la restrizione delle grammatiche regolari, che si ottengono con forti vincoli sulla parte destra delle produzioni. Possiamo ora dare un veloce sguardo alle generalizzazioni delle grammatiche libere, nelle quali è la parte *sinistra* delle produzioni a permettere più flessibilità.

**Definizione 4.14** *Sia  $G = (NT, T, R, S)$  una quadrupla in cui  $NT$ ,  $T$  e  $S$  sono come in una grammatica libera. La nozione di derivazione è definita in modo analogo, a partire dalle produzioni (elementi di  $R$ ), che possono però avere forma più generale di quella delle grammatiche libere:*

1.  $G$  è una grammatica dipendente dal contesto (o *contestuale*, o di tipo 1) quando tutte le produzioni di  $R$  hanno la forma  $uAw \rightarrow uvw$ , con  $A \in NT$ ,  $u, w \in (T \cup NT)^*$  e  $v \in (T \cup NT)^+$ . La produzione  $S \rightarrow \epsilon$  è ammessa solo se  $S$  non compare a destra di nessuna produzione. Un linguaggio  $L$  è dipendente dal contesto se esiste una grammatica contestuale  $G$  per cui  $L = L[G]$ .
2.  $G$  è una grammatica a struttura di frase (o *generale*, o di tipo 0), quando tutte le produzioni di  $R$  hanno la forma  $u \rightarrow v$ , con  $u, v \in (T \cup NT)^*$ . Un linguaggio è semidecidibile se esiste un grammatica generale  $G$  per cui  $L = L[G]$ .

È immediato osservare che i vincoli sulle grammatiche sono via via meno stringenti: ogni grammatica regolare è libera da contesto; ogni grammatica libera è contestuale; ogni grammatica contestuale è a struttura di frase. Corrispondentemente, abbiamo una gerarchia di linguaggi (detta *gerarchia di Chomsky*): i linguaggi regolari sono inclusi nei libri, che sono inclusi nei contestuali, che sono a loro volta semidecidibili. Più interessante è che questa gerarchia è *propria*:

<sup>3</sup>La dimostrazione può usare il PL con la stringa  $z = a^N b^N a^N b^N$ , ma il ragionamento per casi è abbastanza complesso.

**Teorema 4.15** Esistono: linguaggi liberi non regolari; linguaggi contestuali non liberi; linguaggi semidecidibili non contestuali.

- $L_{wwr} = \{ww^R \mid w \in \Sigma^*\}$  è libero non regolare;
- $L = \{a^n b^n c^n \mid n \geq 0\}$  è contestuale non libero;
- i risultati del Capitolo 5 permettono di costruire linguaggi semidecidibili non contestuali.

Come ai linguaggi regolari corrispondono i DFA, e ai liberi i PDA, così è possibile caratterizzare anche le due classi superiori mediante automi. Se in un PDA sostituiamo la pila con un nastro su cui l'automa può liberamente leggere, scrivere e spostarsi avanti e indietro, con l'unico vincolo che non può mai usare un numero di celle del nastro superiore a quello dei caratteri dell'input, otteniamo un *automa limitato linearmente*. Se rimuoviamo anche il vincolo sulla lunghezza del nastro otteniamo un automa generale, una nozione equivalente a quella di *macchina di Turing* (riquadro a pag. 139). Queste due nozioni caratterizzano esattamente le due classi di linguaggi che abbiamo appena introdotto:

- un linguaggio è contestuale sse è accettato da un automa limitato linearmente;
- un linguaggio è semidecidibile sse è accettato da un automa generale (ovvero da una macchina di Turing).

## 4.5 Analizzatori sintattici

Nel contesto dei linguaggi di programmazione, un riconoscitore di un linguaggio libero è chiamato *analizzatore sintattico*, o *parser*. Si tratta di programmi, in genere basati su un PDA per il linguaggio riconosciuto, che usano l'input per decidere le produzioni da utilizzare e che costruiscono un albero di derivazione (se esiste). Come abbiamo visto nel Paragrafo 4.1, ciò è equivalente a determinare una derivazione sinistra (o destra), perché ogni derivazione sinistra (destra) corrisponde biunivocamente ad un albero. Una prima importante distinzione è tra parser *deterministici* e parser non deterministici. I secondi usano l'informazione dell'input per scegliere le produzioni da utilizzare, ma se in futuro si scopre che la scelta è improduttiva e non porta a nessun albero, il parser può tornare sui propri passi, disfare parte della derivazione appena costruita, e scegliere un'altra produzione, tornando a leggere (parte de) l'input. I parser deterministici, al contrario, leggono l'input una sola volta ed ogni loro decisione è definitiva, nel senso che se in futuro si scopre che non si è in grado di costruire un albero di derivazione, ciò accade perché l'albero non esiste, cioè l'input non appartiene al linguaggio generato dalla grammatica. È evidente che i parser deterministici sono quelli più interessanti per i nostri scopi, ma affinché possano essere costruiti occorre che sia il linguaggio che la sua grammatica godano di determinate proprietà, che studieremo in dettaglio nel seguito.

Una seconda importante distinzione è quella tra parser *top-down* e parser *bottom-up*. I parser top-down sfruttano l'informazione dell'input per costruire una

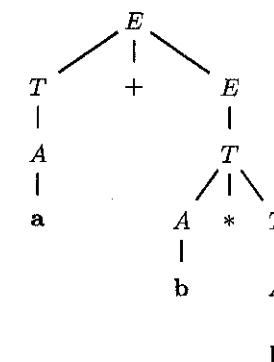


Figura 4.3 Albero di derivazione per  $a + b * b$ .

derivazione a partire dal simbolo iniziale della grammatica. In genere ricercano una derivazione sinistra, e costruiscono incrementalmente l'albero, espandendo ad ogni passo una foglia, fino a quando tutte le foglie non siano etichettate con terminali. Un parser bottom-up, al contrario, cerca di costruire una derivazione "alla rovescia", partendo dall'input e cercando quali produzioni possono aver generato la parte di input già analizzata. In genere questi parser ricercano una derivazione destra, e costruiscono incrementalmente l'albero partendo dalle foglie etichettate con terminali, cercando di far crescere l'albero verso la radice.

**Esempio 4.16** Consideriamo la seguente grammatica per espressioni aritmetiche:

$$G = (\{E, T, A\}, \{a, b, +, -, *, (), ()\}, R, E)$$

$$\begin{aligned} E &\rightarrow T \mid T + E \mid T - E \\ T &\rightarrow A \mid A * T \\ A &\rightarrow a \mid b \mid (E) \end{aligned}$$

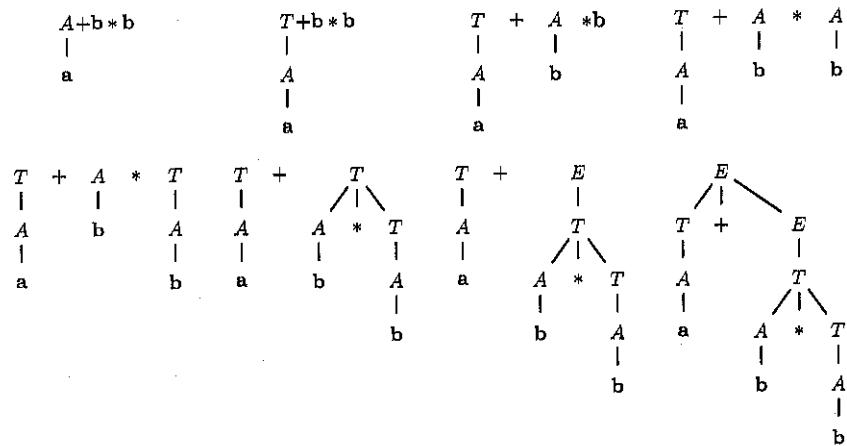
La stringa  $a + b * b$  ammette la (unica) derivazione sinistra

$$\begin{aligned} E &\Rightarrow T + E \Rightarrow A + E \Rightarrow a + E \Rightarrow a + T \\ &\Rightarrow a + A * T \Rightarrow a + b * T \Rightarrow a + b * A \Rightarrow a + b * b \end{aligned}$$

e la (unica) derivazione destra (scritta al contrario)

$$\begin{aligned} a + b * b &\Leftarrow A + b * b \Leftarrow T + b * b \Leftarrow T + A * b \\ &\Leftarrow T + A * A \Leftarrow T + A * T \Leftarrow T + T \\ &\Leftarrow T + E \Leftarrow E \end{aligned}$$

Entrambe le derivazioni corrispondono all'albero di derivazione della Figura 4.3; la derivazione sinistra lo costruisce a partire dalla radice, nell'ovvio ordine di



**Figura 4.4** Costruzione bottom-up di un albero di derivazione.

applicazione delle produzioni. Più interessante è la ricostruzione che corrisponde alla derivazione destra, i cui passi sono riportati in Figura 4.4.

Una derivazione destra “al contrario” lavora sulla testa della stringa di input, cercando di *ridurre* porzioni di input ad un non terminale. Per prima cosa il terminale **a** è ridotto ad  $A$  e, quindi, a  $T$ . A questo punto il terminale **+** non può essere ancora gestito: viene letto, memorizzato (nella pila del riconoscitore) e la costruzione dell’albero procede con il terminale **b**. Ad un certo punto, anche  $*$  sarà messo sulla pila, fino a che sia  $*$  che, poi,  $+$  possano essere usati per completare l’albero.

Nel seguito vedremo classi importanti di grammatiche che permettono la realizzazione di parser deterministici:

- le grammatiche  $LL(k)$  danno luogo a parser top-down: la prima L sta per *left-to-right*, riferito a come viene consumato l’input; la seconda L sta per *left derivation*; si tratta di parser top-down abbastanza semplici da realizzare, anche senza l’ausilio di strumenti automatici sofisticati.
- le grammatiche  $LR(k)$  danno luogo a parser bottom-up: la L sta, come nel caso precedente, per input consumato da sinistra, mentre la R sta per *right derivation*; si tratta di parser potenti, ma la cui costruzione è laboriosa e necessita in genere di strumenti automatici.

In entrambi i casi il parametro  $k$  indica quanti simboli dell’input occorre vedere (senza consumare) prima di poter prendere una decisione (nei casi pratici è sempre  $k = 1$ ). Tratteremo dei parser  $LL(k)$  e  $LR(k)$ , rispettivamente, nei Paragrafi 4.7 e 4.8. Il prossimo paragrafo discute alcune manipolazioni delle grammatiche, utili per generare un parser deterministico: può anche essere saltato in prima lettura, tornando indietro al bisogno.

### Forma normale di Chomsky

Una grammatica libera  $G = (NT, T, R, S)$  è in *forma normale di Chomsky* se tutte le sue produzioni sono della forma  $A \rightarrow BC$ ,  $A \rightarrow a$ , oppure  $S \rightarrow \epsilon$ , con  $A, B, C \in NT$  e  $a \in T$ . La produzione  $S \rightarrow \epsilon \in R$  sse  $\epsilon \in \mathcal{L}[G]$ ; inoltre  $S$  non compare mai a destra di una produzione.

Le grammatiche in forma di Chomsky sono molto utili perché non hanno  $\epsilon$ -produzioni (eccetto l’unica banale  $S \rightarrow \epsilon$  se  $\epsilon \in \mathcal{L}[G]$ ). Inoltre, non hanno produzioni unitarie, della forma  $A \rightarrow B$ . Ogni grammatica  $G$  può essere trasformata in una grammatica in forma normale di Chomsky  $G'$  con  $\mathcal{L}[G] = \mathcal{L}[G']$ .

### Forma normale di Greibach

Una grammatica libera  $G = (NT, T, R, S)$  è in *forma normale di Greibach* se tutte le sue produzioni sono della forma  $A \rightarrow aBC$ ,  $A \rightarrow aB$ ,  $A \rightarrow a$ , oppure  $S \rightarrow \epsilon$ , con  $A, B, C \in NT$  e  $a \in T$ . La produzione  $S \rightarrow \epsilon \in R$  sse  $\epsilon \in \mathcal{L}[G]$ ; inoltre  $S$  non compare mai a destra di una produzione.

Le grammatiche in forma di Greibach, come quelle in forma di Chomsky, non hanno né  $\epsilon$ -produzioni (eccetto l’eventuale  $S \rightarrow \epsilon$ ), né produzioni unitarie. Il loro interesse principale sta nel fatto che ogni produzione genera sempre almeno un simbolo terminale: nel caso di una derivazione sinistra, ogni produzione allunga il prefisso di almeno un simbolo. Ne segue che una grammatica in forma di Greibach non è mai ricorsiva sinistra. Ogni grammatica  $G$  può essere trasformata in una grammatica in forma normale di Greibach  $G'$  con  $\mathcal{L}[G] = \mathcal{L}[G']$ .

## 4.6 Manipolazioni delle grammatiche

Le forma delle produzioni di una grammatica libera è ancora troppo generale se vogliamo utilizzare una grammatica per progettare un riconoscitore di linguaggi. Vedremo in questo paragrafo alcune manipolazioni delle produzioni, per rendere la grammatica più semplice, o più adatta alla realizzazione di un parser deterministico, il tutto senza modificare il linguaggio generato. Le trasformazioni più generali sono quelle che consentono di trasformare una grammatica in forma normale di Chomsky o di Greibach, ma, in specie la seconda, sono troppo generali per trattarle in questo testo introduttivo. Vedremo dunque solo le trasformazioni che ci saranno utili per il seguito della trattazione.

### 4.6.1 Eliminare le $\epsilon$ -produzioni

Una produzione della forma  $A \rightarrow \epsilon$  è una  *$\epsilon$ -produzione*. Mostriamo come le  $\epsilon$ -produzioni possono sempre essere eliminate da una grammatica, senza modificare il potere espressivo. L’unica eccezione si ha quando  $\epsilon \in \mathcal{L}[G]$ : è evidente che la stringa vuota può essere derivata solo con una  $\epsilon$ -produzione; ma possiamo limitarci alla sola produzione  $S \rightarrow \epsilon$ , con  $S$  simbolo iniziale.

Iniziamo con l'individuazione dei simboli annullabili: un non terminale  $A$  è *annullabile* sse  $A \Rightarrow^+ \epsilon$ . Data una grammatica  $G$ , l'insieme dei suoi simboli annullabili  $N(G)$  è calcolato come segue:

1.  $N_0(G) = \{A \in NT \mid A \rightarrow \epsilon \in R\};$
2.  $N_{i+1}(G) = N_i(G) \cup \{B \in NT \mid B \rightarrow C_1 \cdots C_k \in R \text{ e } C_1, \dots, C_k \in N_i(G)\}.$

È facile vedere che  $N_i(G) \subseteq N_{i+1}(G)$ ; inoltre, essendo  $NT$  finito, per un certo  $i_c$  si avrà  $N_{i_c}(G) = N_{i_c+1}(G)$ .

**Fatto 4.17** L'insieme  $N(G) = N_{i_c}(G)$  è esattamente l'insieme di tutti i simboli annullabili di  $G$ .

#### Approfondimento

4.4

Costruiamo ora la grammatica  $G' = (NT, T, R', S)$ . Per ogni produzione  $A \rightarrow \alpha$  di  $G$  con  $\alpha \neq \epsilon$ , siano  $Z_1, \dots, Z_k$  le occorrenze di non terminali annullabili che occorrono in  $\alpha$ .<sup>4</sup> Inseriamo in  $R'$  tutte le produzioni che si ottengono da  $A \rightarrow \alpha$  cancellando tutti i possibili sottoinsiemi di  $Z_1, \dots, Z_k$  (incluso l'insieme vuoto). Unica eccezione: se  $\alpha$  è composta da soli simboli annullabili, non inseriamo la produzione che si ottiene omettendo tutti i simboli di  $\alpha$ .

**Teorema 4.18** Data una grammatica  $G$ , la grammatica  $G'$  determinata come sopra non ha  $\epsilon$ -produzioni e  $\mathcal{L}[G'] = \mathcal{L}[G] - \{\epsilon\}$ .

**Esempio 4.19** Data la grammatica con  $T = \{a, b, c\}$  e produzioni

$$\begin{aligned} A &\rightarrow ABA \mid C \mid a \\ B &\rightarrow b \\ C &\rightarrow cC \mid \epsilon \end{aligned}$$

I simboli annullabili sono  $N(G) = \{A, C\}$ . Le produzioni della grammatica  $G'$  sono

$$\begin{aligned} A &\rightarrow ABA \mid BA \mid AB \mid B \mid C \mid a \\ B &\rightarrow b \\ C &\rightarrow cC \mid c \end{aligned}$$

## 4.6.2 Eliminare i simboli inutili

Fissata una grammatica  $G = (NT, T, R, S)$ , un simbolo  $X \in T \cup NT$  è *utile* quando  $X$  compare in almeno una derivazione di una stringa del linguaggio

<sup>4</sup>Se un simbolo compare più di una volta in una stringa, le sue occorrenze sono multiple e diverse. Ad esempio, nella parte destra di  $A \rightarrow BaBA$ , vi sono tre occorrenze di non terminali.

generato. Formalmente: esiste una derivazione  $S \Rightarrow^* \alpha X \beta \Rightarrow^* w \in \mathcal{L}[G]$  ( $\alpha X \beta$  potrebbe coincidere con  $S$ , o con  $w$ ). Un simbolo che non è utile è *inutile*. Possiamo caratterizzare l'utilità in termini di due proprietà più semplici.

**Definizione 4.20** Un simbolo  $X \in T \cup NT$  è:

1. un generatore sse esiste  $w \in T^*$  con  $X \Rightarrow^* w$ ;
2. raggiungibile sse  $S \Rightarrow^* \alpha X \beta$  per qualche  $\alpha, \beta \in (T \cup NT)^*$ .

È evidente che tutti i terminali sono generatori e che  $X$  è utile sse è sia un generatore che raggiungibile. Per eliminare i simboli inutili, possiamo prima eliminare i simboli che non sono generatori, e quindi i non raggiungibili.

**Esempio 4.21** Consideriamo le produzioni seguenti (con le usuali convenzioni su terminali ecc.)

$$S \rightarrow AB \mid a \quad B \rightarrow b$$

L'unico simbolo che non è generatore è  $A$ . Possiamo cancellare ogni produzione in cui compare, ottenendo

$$S \rightarrow a \quad B \rightarrow b$$

Ora  $B$  non è raggiungibile, per cui possiamo rimuovere tutte le sue produzioni, ottenendo

$$S \rightarrow a$$

Si osservi che eliminare *prima* i simboli irraggiungibili e poi i non generatori non funziona. Nel nostro caso, la grammatica di partenza non ha simboli irraggiungibili; dopo aver rimosso i non generatori ci fermeremmo alla seconda grammatica, che comprende ancora l'inutile  $B$ .

**Determinare i generatori** Data una grammatica  $G$ , l'insieme dei suoi simboli generatori è calcolato iterativamente come segue:

1.  $G_0(G) = T;$
2.  $G_{i+1}(G) = G_i(G) \cup \{B \in NT \mid B \rightarrow C_1 \cdots C_k \in R \text{ e } C_1, \dots, C_k \in G_i(G)\}.$

Nel passo (2) è contemplato anche il caso  $k = 0$  (cioè se  $B \rightarrow \epsilon$ ,  $B$  è un generatore).

Si ha ovviamente  $G_i(G) \subseteq G_{i+1}(G)$  e come nel caso dei simboli annullabili, essendo  $NT$  finito, per un certo  $i_c$  si avrà  $G_{i_c}(G) = G_{i_c+1}(G)$ .

**Fatto 4.22** L'insieme  $G(G) = G_{i_c}(G)$  è esattamente l'insieme dei simboli generatori di  $G$ .

**Determinare i simboli raggiungibili** Data una grammatica  $G$ , l'insieme dei suoi simboli raggiungibili è calcolato iterativamente come segue:

$$\begin{aligned} 1. \quad R_0(G) &= \{S\}; \\ 2. \quad R_{i+1}(G) &= R_i(G) \cup \\ &\quad \{X_i \in T \cup NT \mid B \rightarrow X_1 \dots X_k \in R \\ &\quad \text{per qualche } B \in R_i(G)\}. \end{aligned}$$

Essendo  $T \cup NT$  finito, per un certo  $i_c$  si avrà  $R_{i_c}(G) = R_{i_c+1}(G)$ .

**Fatto 4.23** L'insieme  $R(G) = R_{i_c}(G)$  è esattamente l'insieme dei simboli raggiungibili di  $G$ .

Possiamo ora mettere assieme i due passi appena discussi per eliminare i simboli inutili, come già discusso poc'anzi.

**Teorema 4.24** Sia  $G = (NT, T, R, S)$  una grammatica con  $\mathcal{L}[G] \neq \emptyset$ .

1. Sia  $G_1$  la grammatica che si ottiene da  $G$  eliminando tutti i simboli che non appartengono a  $R(G)$ , e tutte le produzioni che fanno uso di almeno uno di tali simboli.
2. Sia  $G_2$  la grammatica che si ottiene da  $G_1$  eliminando tutti i simboli che non appartengono a  $R(G_1)$  e tutte le produzioni che fanno uso di almeno uno di tali simboli.

$G_2$  non ha simboli inutili e  $\mathcal{L}[G] = \mathcal{L}[G_2]$ .

### 4.6.3 Eliminare le produzioni unitarie

Una produzione è *unitaria* se ha la forma  $A \rightarrow B$ , con  $A$  e  $B$  entrambi non terminali. Sebbene si tratti di produzioni spesso utili (in specie per risolvere l'ambiguità), in alcune costruzioni è necessario partire da grammatiche che non hanno produzioni unitarie. Rimuovere una sola produzione unitaria  $A \rightarrow B$  non è difficile: basta sostituire la sua parte destra ("espanderla") con le parti destre delle produzioni per  $B$ .

**Esempio 4.25** Partiamo dalle produzioni

$$\begin{array}{l|l} S \rightarrow A & SaA \\ A \rightarrow B & AbB \\ B \rightarrow C & BcC \\ C \rightarrow d & e \\ & | \\ & f \end{array}$$

che presentano tre produzioni unitarie. Rimuoviamo la produzione unitaria  $S \rightarrow A$ , sostituendola con le due produzioni  $S \rightarrow B$  e  $S \rightarrow AbB$  (che si ottengono dalle produzioni che hanno  $A$  come testa). Così facendo abbiamo però introdotto una *nuova* produzione unitaria:  $S \rightarrow B$ . Eliminiamola nello stesso modo, cioè

espandendola in  $S \rightarrow C$  e  $S \rightarrow BcC$ . Ora dobbiamo trattare  $S \rightarrow C$ , espandendola in  $S \rightarrow d$ ,  $S \rightarrow e$  e infine  $S \rightarrow f$ . Questa volta non abbiamo introdotto nessuna nuova produzione unitaria. Le produzioni per  $S$  sono dunque

$$S \rightarrow d \mid e \mid f \mid BcC \mid AbB \mid SaA$$

mentre le produzioni degli altri non terminali sono inalterate. Con lo stesso metodo potremmo ora eliminare le altre due produzioni unitarie di questa grammatica ( $A \rightarrow B$  e  $B \rightarrow C$ ).

Questo semplice metodo non funziona, tuttavia, nel caso generale, perché le produzioni unitarie potrebbero avere dei cicli tra loro, ovvero potremmo avere una derivazione  $A \Rightarrow^+ A$  fatta tutta di produzioni unitarie. In questa situazione dopo un po' di espansioni ci ritroveremmo con la produzione unitaria da cui siamo partiti. Per gestire questo caso e rompere questi cicli, determiniamo per prima cosa le *copie unitarie* di non terminali  $A$  e  $B$ , cioè quelle copie per le quali  $A \Rightarrow^* B$  con solo produzioni unitarie.

Data una grammatica  $G$ , l'insieme delle sue copie unitarie è calcolato iterativamente come segue:

$$\begin{aligned} 1. \quad U_0(G) &= \{(A, A) \mid A \in NT\}; \\ 2. \quad U_{i+1}(G) &= U_i(G) \cup \\ &\quad \{(A, C) \mid B \rightarrow C \in R \text{ con } C \in NT \text{ e } (A, B) \in U_i(G)\}. \end{aligned}$$

Essendo  $NT$  finito, per un certo  $i_c$  si avrà  $U_{i_c}(G) = U_{i_c+1}(G)$ .

**Fatto 4.26** L'insieme  $U(G) = U_{i_c}(G)$  è esattamente l'insieme delle copie unitarie di  $G$ .

Possiamo ora usare le copie unitarie in un procedimento generale per la rimozione delle produzioni unitarie.

**Teorema 4.27** Sia  $G = (NT, T, R, S)$  una grammatica e sia  $U(G)$  l'insieme delle sue copie unitarie. Sia  $G_1 = (NT, T, R_1, S)$  la grammatica dove per ogni  $(A, B) \in U(G)$ , l'insieme delle produzioni  $R_1$  contiene tutte le produzioni non unitarie  $B \rightarrow \alpha$  di  $R$ .  $G_1$  non ha produzioni unitarie e  $\mathcal{L}[G] = \mathcal{L}[G_1]$ .

Nell'enunciato del teorema, si osservi che siccome le copie  $(A, A)$  sono tutte unitarie, in  $R_1$  vengono certo inserite tutte le produzioni non unitarie di  $R$ .

### 4.6.4 Mettere insieme le cose

L'ordine con il quale le trasformazioni sono applicate è importante, perché alcune delle costruzioni che abbiamo visto interagiscono con le altre. Un metodo che garantisce un risultato complessivo è il seguente:

1. elimina le  $\epsilon$ -produzioni;

2. elimina le produzioni unitarie (e dunque i cicli);
3. elimina i simboli inutili.

In questo modo la grammatica risultante è garantita non avere né  $\epsilon$ -produzioni, né produzioni unitarie, né simboli inutili.

#### 4.6.5 Eliminare la ricorsione sinistra

Una grammatica è *ricorsiva sinistra* se ammette una derivazione  $A \Rightarrow^+ A\alpha$  per qualche  $\alpha \in (T \cup NT)^*$ . I riconoscitori sintattici top-down non sono in grado di trattare grammatiche ricorsive sinistre: daremo quindi un modo generale per trasformare la ricorsione sinistra in ricorsione destra.

**Ricorsione immediata** La ricorsione sinistra immediata, cioè quella che si presenta applicando una sola produzione, è semplice da rimuovere. Consideriamo infatti una grammatica con produzioni ricorsive sinistre immediate per un non terminale  $A$ :

$$A \rightarrow A\alpha_1 | \dots | A\alpha_n | \beta_1 | \dots | \beta_m$$

(le stringhe  $\beta_i$  non iniziano con  $A$ ). Queste produzioni possono essere rimpiazzate da

$$\begin{aligned} A &\rightarrow \beta_1 A' | \dots | \beta_m A' \\ A' &\rightarrow \alpha_1 A' | \dots | \alpha_n A' | \epsilon \end{aligned}$$

Non è difficile convincersi che i due insiemi di produzioni sono equivalenti quanto a stringhe di terminali generate. Se infatti nella prima deriviamo  $A \Rightarrow^* w \in T^*$ , ciò significa che la ricorsione su  $A$  ad un certo punto è stata interrotta con un  $\beta_i$ ; ad esempio con una derivazione sinistra:

$$A \rightarrow A\alpha_{i_1} \rightarrow A\alpha_{i_2}\alpha_{i_1} \rightarrow \dots \rightarrow A\alpha_{i_k} \dots \alpha_{i_2}\alpha_{i_1} \rightarrow \beta_i\alpha_{i_k} \dots \alpha_{i_2}\alpha_{i_1} \Rightarrow^* w$$

Questa derivazione può essere simulata con il secondo insieme di produzioni nel modo ovvio:

$$A \rightarrow \beta_i A' \rightarrow \beta_i\alpha_{i_k} A' \rightarrow \dots \rightarrow \beta_i\alpha_{i_k} \dots \alpha_{i_2}\alpha_{i_1} A' \rightarrow \beta_i\alpha_{i_k} \dots \alpha_{i_2}\alpha_{i_1} \epsilon \Rightarrow^* w$$

Il viceversa è analogo.

**Ricorsione non immediata** L'eliminazione della ricorsione non immediata è più laboriosa e può essere ottenuta con l'algoritmo seguente. Sia  $G$  una grammatica senza  $\epsilon$ -produzioni e senza produzioni unitarie.<sup>5</sup>

<sup>5</sup>Per esser precisi, è solo necessario che  $G$  non abbia cicli, cioè derivazioni  $A \Rightarrow^+ A$ .

```
sia  $NT = \{A_1, A_2, \dots, A_n\}$ , in un ordine fissato.
for  $i$  from 1 to  $n$ 
  for  $j$  from 1 to  $i-1$ 
    sostituisci ogni produzione della forma  $A_i \rightarrow A_j\alpha$ 
    con le produzioni  $A_i \rightarrow \beta_1\alpha | \dots | \beta_k\alpha$ 
    dove  $A_j \rightarrow \beta_1 | \dots | \beta_k$  sono tutte le produzioni correnti per  $A_j$ 
  }
  elimina la ricorsione immediata su  $A_i$ .
}
```

L'eliminazione della ricorsione immediata può inserire  $\epsilon$ -produzioni, che rimarranno nella grammatica finale prodotta dall'algoritmo.

**Esempio 4.28** Applichiamo la procedura di eliminazione alle produzioni

$$\begin{aligned} S &\rightarrow Ba | b \\ B &\rightarrow Bc | Sc | d \end{aligned}$$

Ordiniamo  $NT = \{S, B\}$ . Per  $i = 1$ , il ciclo interno non viene eseguito ( $j$  varia da 1 a 0); siccome non c'è ricorsione immediata su  $S$ , non succede nulla. Sia ora  $i = 2$  (cioè  $A_i = B$  e il ciclo interno si esegue solo per  $A_j = A_1 = S$ ): sostituiamo  $B \rightarrow Sc$  con  $B \rightarrow Bac | bc$ . Le produzioni correnti per  $B$  sono dunque

$$B \rightarrow Bc | Bac | bc | d$$

dalle quali dobbiamo eliminare la ricorsione immediata. Otteniamo

$$\begin{aligned} B &\rightarrow bcB' | dB' \\ B' &\rightarrow cB' | acB' | \epsilon \end{aligned}$$

La grammatica finale è dunque quella con produzioni

$$\begin{aligned} S &\rightarrow Ba | b \\ B &\rightarrow bcB' | dB' \\ B' &\rightarrow cB' | acB' | \epsilon \end{aligned}$$

#### 4.6.6 Fattorizzazione sinistra

Oltre alla ricorsione sinistra, un'altra situazione che impedisce la realizzazione di alcuni parser si presenta quando più produzioni per uno stesso non terminale iniziano con la stessa stringa, come ad esempio

$$A \rightarrow aBbC | aBd$$

Un parser che cerchi di determinare una derivazione analizzando l'input da sinistra a destra non è in grado di scegliere tra le due produzioni vedendo il solo simbolo  $a$ . *Fattorizzare* una grammatica significa ristrutturarne le produzioni,

```

sia  $N = NT$ ;
ripeti il ciclo seguente, finché nessuna modifica
è più possibile a  $N$  o alle produzioni {
    foreach non terminale  $A$  {
        sia  $\alpha$  il prefisso comune più lungo alle parti destre delle
            produzioni di  $A$ ;
        if  $\alpha \neq \epsilon$  {
            sia  $A'$  un nuovo non terminale: aggiungi  $A'$  a  $N$ ;
            rimpiazza tutte le produzioni per  $A$ 
                 $A \rightarrow \alpha\beta_1 | \dots | \alpha\beta_k | \gamma_1 | \dots | \gamma_h$ 
            con le produzioni
                 $A \rightarrow \alpha A' | \gamma_1 | \dots | \gamma_h$ 
                 $A' \rightarrow \beta_1 | \dots | \beta_k$ 
        }
    }
}
}

```

**Figura 4.5** Fattorizzazione sinistra di una grammatica.

“raccogliendo” la parte comune delle parti destre delle produzioni. Nel caso d’ esempio, potremmo introdurre un nuovo non terminale  $A'$  e riscrivere le produzioni come

$$\begin{aligned} A &\rightarrow aBA' \\ A' &\rightarrow bC \mid d \end{aligned}$$

Il caso generale non è difficile ed è trattato dall’ algoritmo della Figura 4.5.

**Esempio 4.29** La grammatica dell’ Esempio 4.16 ha produzioni che possono essere fattorizzate. Applicando la tecnica appena discussa si ottengono le produzioni

$$\begin{aligned} E &\rightarrow TE' \\ E' &\rightarrow \epsilon \mid +E \mid -E \\ T &\rightarrow AT' \\ T' &\rightarrow \epsilon \mid *T \\ A &\rightarrow a \mid b \mid (E) \end{aligned}$$

## 4.7 Parser top-down

I parser top-down costruiscono un albero di derivazione a partire dal simbolo iniziale, usando informazioni della stringa di ingresso per decidere quale produzione applicare. I parser più semplici di questo tipo sono non deterministici e si ottengono direttamente dalla grammatica: si tratta dei parser *a discesa ricorsiva*.

```

void A() {
    Scegli non deterministicamente  $h$  tra 1 e  $k$ ,          2
    ovvero una produzione  $A \rightarrow X_1^h \dots X_{n_h}^h$ ;          4
    for ( $i = 1$  to  $n_h$ ) {
        if ( $X_i^h \in NT$ )  $X_i^h()$ ;
        else if ( $X_i^h$  = simbolo corrente in input)          6
            avanza di un simbolo nell’ input;
        else fail();                                      8
    }
    return;                                              10
}

```

**Figura 4.6** Parser a discesa ricorsiva: funzione per il non terminale  $A$ .

### 4.7.1 Parser a discesa ricorsiva

Un parser a discesa ricorsiva si ottiene associando ad ogni non terminale  $A$  una funzione ricorsiva  $A()$ , che funge da riconoscitore delle stringhe derivabili da  $A$ . Nel caso in cui  $A()$  (o una funzione chiamata da essa) non riesca a determinare una possibile derivazione, si fa uso di *backtracking*, cioè si torna indietro nella computazione tentando altre strade. Supponiamo che al non terminale  $A$  siano associate le  $k$  produzioni

$$A \rightarrow X_1^1 \dots X_{n_1}^1 \mid \dots \mid X_1^k \dots X_{n_k}^k,$$

con  $X_j^i \in T \cup NT$ . La funzione associata ad  $A$  avrà la struttura descritta nella Figura 4.6. Scelta una produzione alle linee 2-3,  $A()$  cerca di determinare se l’ input è una stringa che si deriva da  $A$ : se un terminale della produzione coincide con il terminale sull’ input (linea 6), si avanza nell’ input e si procede con il prossimo elemento della produzione ( $i$  verrà incrementato); se la produzione contiene un non terminale (linea 5), si chiama la funzione relativa. Alla linea 8 si arriva dunque solo se il simbolo nella produzione è un terminale diverso da quello che si vede sull’ input: in questo caso la funzione termina con “fallimento” (la chiamata a *fail()*). Se invece il *for* viene completato (cioè tutte le chiamate ricorsive ai non terminali terminano con successo e i simboli terminali della produzione sono in accordo con quelli dell’ input), la funzione termina con successo (linea 10).

La terminazione con fallimento della linea 8, tuttavia, non è definitiva (per la stringa in input): il fallimento potrebbe dipendere dall’ aver scelto una produzione sbagliata alla linea 2. La chiamata a *fail()*, dunque, implica backtracking. Ad ogni scelta non deterministica delle linee 2-3 viene fissato un punto di ritorno; ogni volta che si verifica un *fail()* si determina il punto di ritorno più vicino e si tenta una scelta diversa. Solo dopo che *tutte* le scelte possibili sono state tentate il fallimento diviene definitivo (la stringa non appartiene al linguaggio generato dalla grammatica).

I parser a discesa ricorsiva sono concettualmente molto semplici, ma sono computazionalmente inefficienti, per la necessità di esplorare, nel caso peggiore, tutte le possibili alternative.<sup>6</sup>

#### 4.7.2 First e Follow

L'inefficienza dei parser a discesa ricorsiva dipende in modo cruciale dal fatto che la scelta di quale produzione usare (alle linee 2-3 della Figura 4.6) viene fatta alla cieca, senza sfruttare l'input. Supponiamo, ad esempio, che le sole produzioni per il non terminale  $A$  siano

$$A \rightarrow a\alpha \mid b\beta$$

con  $a, b$  terminali distinti. Se sull'input vediamo  $b$ , è evidente che non c'è motivo di tentare la prima produzione: solo la seconda (che inizia proprio con il simbolo che vediamo sull'input) potrà portarci ad una derivazione. Per dare condizioni generali che sfruttino questa idea, è opportuno introdurre un paio di nozioni, che saranno utili anche per i parser bottom-up. Da qui in poi supporremo che il simbolo speciale  $\$$  non faccia parte dei simboli di nessuna grammatica: utilizzeremo  $\$$  come segnalatore di fine input durante un'analisi sintattica.

**First** Fissiamo una grammatica  $G$  e sia  $\alpha$  una stringa su  $T \cup NT$ . Definiamo  $\text{FIRST}(\alpha)$  come l'insieme dei terminali che possono stare in prima posizione in una stringa che si deriva da  $\alpha$ : per  $a \in T$ ,  $a \in \text{FIRST}(\alpha)$  sse  $\alpha \Rightarrow^* a\beta$  per qualche  $\beta \in (T \cup NT)^*$ . Inoltre, se da  $\alpha$  si può derivare la stringa vuota, anche la stringa vuota appartiene a  $\text{FIRST}(\alpha)$ : se  $\alpha \Rightarrow^* \epsilon$ , allora  $\epsilon \in \text{FIRST}(\alpha)$ .

Se ora abbiamo due produzioni per lo stesso non terminale

$$A \rightarrow \alpha_1 \mid \alpha_2,$$

un parser top-down può guardare al simbolo corrente di input  $a$ : sceglierà la prima produzione se  $a \in \text{FIRST}(\alpha_1)$ , mentre sceglierà la seconda se  $a \in \text{FIRST}(\alpha_2)$ . Se, inoltre,  $\text{FIRST}(\alpha_1)$  e  $\text{FIRST}(\alpha_2)$  sono *disgiunti* la scelta è deterministica e non c'è bisogno di backtracking, almeno per la coppia di produzioni considerate.

Tuttavia, il solo insieme  $\text{FIRST}$  può non essere sufficiente a scegliere la produzione corretta. Consideriamo infatti la semplice grammatica seguente

$$\begin{aligned} S &\rightarrow Ab \mid c \\ A &\rightarrow aA \mid \epsilon \end{aligned}$$

e calcoliamo i  $\text{FIRST}$  delle parti destre di tutte le produzioni:

$$\begin{aligned} \text{FIRST}(Ab) &= \{a, b\} & \text{FIRST}(c) &= \{c\} \\ \text{FIRST}(aA) &= \{a\} & \text{FIRST}(\epsilon) &= \{\epsilon\}. \end{aligned}$$

<sup>6</sup>Sono dunque esponenziali nella lunghezza della stringa; la base dell'esponente è data dal massimo numero di produzioni per lo stesso non terminale.

Supponiamo di dover riconoscere la stringa  $ab$ . Partendo da  $S$ , siccome il primo simbolo di input è  $a$ , viene scelta la prima produzione per  $S$ , perché  $a \in \text{FIRST}(Ab)$ . Ora si deve scegliere una produzione per  $A$ : l'input corrente è ancora  $a$  e in base ai  $\text{FIRST}$  si sceglie la prima produzione di  $A$ , ottenendo la derivazione sinistra parziale  $S \Rightarrow Ab \Rightarrow aAb$ . Siccome ora il simbolo corrente di input si accorda ("fa match") con il primo simbolo della produzione (linee 6-7 della Figura 4.6), si sposta il simbolo corrente su  $b$  e si cerca una produzione per  $A$ . Ma gli insiemi  $\text{FIRST}$  non ci aiutano, perché nessuno dei due contiene il simbolo corrente  $b$ . Eppure una derivazione esiste: è quella che sostituisce  $A$  con  $\epsilon$ .

Una situazione del genere si ha tutte le volte che la parte destra di una produzione può derivare la stringa vuota (nel nostro semplice esempio, la produzione contiene direttamente  $\epsilon$ ). In questo caso, la scelta deve esser fatta andando a vedere quello che può *seguire* la testa della produzione (nel caso dell'esempio, in una derivazione sinistra  $A$  è sempre seguita da  $b$ : se l'input corrente è  $b$ , occorre scegliere la produzione  $A \rightarrow \epsilon$ ).

**Follow** Fissiamo una grammatica  $G$  e sia  $A$  un suo non terminale. Definiamo  $\text{FOLLOW}(A)$  come l'insieme dei terminali che possono comparire immediatamente a destra di  $A$  in una forma sentenziale: per  $a \in T$ ,  $a \in \text{FOLLOW}(A)$  sse  $S \Rightarrow^* \alpha A a \beta$  per qualche  $\alpha, \beta \in (T \cup NT)^*$ .<sup>7</sup> Inoltre, se  $A$  può essere il simbolo più a destra di una forma sentenziale, allora il simbolo speciale  $\$$  appartiene a  $\text{FOLLOW}(A)$ : se  $S \Rightarrow^* \alpha A$ , allora  $\$ \in \text{FOLLOW}(A)$ .

**Calcolare First e Follow** Per calcolare in modo ordinato i  $\text{FIRST}$  e i  $\text{FOLLOW}$  per una grammatica  $G$  si può innanzitutto determinare i simboli annullabili  $N(G)$  (cioè i non terminali dai quali si deriva  $\epsilon$ ), seguendo la procedura del Paragrafo 4.6.1.

Per calcolare  $\text{FIRST}(X)$  per ogni simbolo  $X$  della grammatica si procede nel modo seguente.

Per ogni  $X \in T$ , inizializza  $\text{FIRST}(X) = \{X\}$ ;  
Per ogni  $X \in NT$ , inizializza  $\text{FIRST}(X) = \emptyset$ ;

Ripeti il ciclo seguente, finché nessun  $\text{FIRST}(X)$  viene più modificato in una iterazione:

Per ogni produzione  $X \rightarrow Y_1 \dots Y_k$   
Per ogni  $i$  da 1 a  $k$   
if  $(Y_1, \dots, Y_{i-1} \in N(G))$  /\* true se  $i = 1$  \*/  
 $\text{FIRST}(X) := \text{FIRST}(X) \cup \text{FIRST}(Y_i)$ ;

Per ogni  $X \in N(G)$ ,  $\text{FIRST}(X) := \text{FIRST}(X) \cup \{\epsilon\}$ ;

Il ciclo interno aggiunge a  $\text{FIRST}(X)$  innanzitutto tutto  $\text{FIRST}(Y_1)$ ; se  $Y_1$  è annullabile, allora aggiunge anche  $\text{FIRST}(Y_2)$ , e così via. L'ultimo aggiorna-

<sup>7</sup>Si osservi che tra  $A$  e  $a$  ci possono essere stati altri simboli durante la derivazione, ma da questi simboli si è poi derivata la stringa vuota, cosicché  $A$  e  $a$  sono diventati contigui.

mento, fuori del ciclo, si prende cura di aggiungere  $\epsilon$  al FIRST di tutti i simboli annullabili.

Il calcolo di FIRST( $\alpha$ ), per  $\alpha \in (T \cup NT)^*$  è una semplice induzione:

$$\begin{aligned} \text{FIRST}(\epsilon) &= \{\epsilon\} \\ \text{FIRST}(X\beta) &= \text{FIRST}(X) && \text{se } X \notin N(G) \\ \text{FIRST}(X\beta) &= (\text{FIRST}(X) - \{\epsilon\}) \cup \text{FIRST}(\beta) && \text{se } X \in N(G). \end{aligned}$$

Il calcolo di FOLLOW è un pò più macchinoso:

Per ogni  $X \in NT$ , inizializza  $\text{FOLLOW}(X) = \emptyset$ ;  
 $\text{FOLLOW}(S) := \{\$\}$ ; /\*  $S$  è il simbolo iniziale \*/

Ripeti il ciclo seguente, finché nessun  $\text{FOLLOW}(X)$  viene più modificato in una iterazione:

Per ogni produzione  $X \rightarrow \alpha Y \beta$   
 $\text{FOLLOW}(Y) := \text{FOLLOW}(Y) \cup (\text{FIRST}(\beta) - \{\epsilon\})$ ;  
 Per ogni produzione  $X \rightarrow \alpha Y$  e  
 per ogni produzione  $X \rightarrow \alpha Y \beta$  con  $\epsilon \in \text{FIRST}(\beta)$   
 $\text{FOLLOW}(Y) := \text{FOLLOW}(Y) \cup \text{FOLLOW}(X)$ ;

Nel secondo “if”, la condizione  $\epsilon \in \text{FIRST}(\beta)$  è un modo sintetico per esprimere che  $\beta \Rightarrow^* \epsilon$ , ovvero che tutti i simboli di  $\beta$  sono annullabili.

**Esempio 4.30** Gli insiemi FIRST e FOLLOW per la grammatica introdotta nell’Esempio 4.29 sono i seguenti.

	FIRST()	FOLLOW()
$E$	a, b, (	\$, )
$E'$	$\epsilon, +, -$	\$, )
$T$	a, b, (	\$, ), +, -
$T'$	$\epsilon, *$	\$, ), +, -
$A$	a, b, (	\$, ), +, -, *

### 4.7.3 Grammatiche LL(1)

Possiamo usare le informazioni degli insiemi FIRST e FOLLOW per guidare un parser a discesa ricorsiva. Il modo più economico è quello di usare una *tabella di parsing LL*, cioè una matrice bidimensionale  $M$  le cui righe sono indicizzate sui non terminali e le cui colonne variano sui terminali (più \$): la casella  $M[A, a]$  contiene le produzioni che possono essere scelte alle linee 2-3 della Figura 4.6, nel corpo della funzione per il non terminale  $A$  e nel momento in cui l’input corrente è  $a$ . La tabella è riempita secondo il metodo seguente.

#### Riempire una tabella di parsing top-down

Per ogni produzione  $A \rightarrow \alpha$ :

- per ogni  $a \in T$  e  $a \in \text{FIRST}(\alpha)$ , inserisci  $A \rightarrow \alpha$  in  $M[A, a]$ ;

	a	b	(	)	+	-	*	\$
$E$	$E \rightarrow TE'$	$E \rightarrow TE'$	$E \rightarrow TE'$					
$E'$				$E' \rightarrow \epsilon$	$E' \rightarrow +E$	$E' \rightarrow -E$		$E' \rightarrow \epsilon$
$T$	$T \rightarrow AT'$	$T \rightarrow AT'$	$T \rightarrow AT'$					
$T'$				$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$	$T' \rightarrow *T$	$T' \rightarrow \epsilon$
$A$	$A \rightarrow a$	$A \rightarrow b$	$A \rightarrow (E)$					

Figura 4.7 Tabella di parsing per la grammatica dell’Esempio 4.29.

- se  $\epsilon \in \text{FIRST}(\alpha)$ , inserisci  $A \rightarrow \alpha$  in tutte le caselle  $M[A, x]$ , per  $x \in \text{FOLLOW}(A)$  (ricorda che in generale  $x \in T \cup \{\$\}$ ).

Ogni casella rimasta vuota dopo aver elaborato tutte le produzioni deve essere considerata una casella di *errore*, cioè in corrispondenza della quale la funzione ricorsiva chiama `fail()`.

Usando una tabella di parsing la scelta non deterministica delle linee 2-3 è ristretta: non più una produzione a caso, ma una tra quelle presenti nell’opportuna casella. Ovviamente questo metodo non assicura, in generale, che il parser che ne risulta sia deterministico: se anche una sola casella della tabella di parsing contiene più produzioni, queste devono essere analizzate tutte con backtracking.

Per alcune grammatiche, tuttavia, capiterà che le caselle contengono tutte al più una produzione: in tal caso il parser è deterministico e viene detto *parser predittivo* perché ricostruisce l’albero predicendo quale produzione usare (tra le molte possibili) guardando sull’input. Le grammatiche per le quali la tabella di parsing top-down contiene al più una produzione per casella sono dette *grammatiche LL(1)*. In modo equivalente, esplicitando le condizioni per le quali il riempimento è unico, possiamo dare la seguente definizione.

**Definizione 4.31** Una grammatica  $G$  è LL(1) sse per ogni coppia di produzioni distinte con la stessa testa

$$A \rightarrow \alpha \mid \beta$$

si ha

- $\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \emptyset$ ;
- (a) se  $\epsilon \in \text{FIRST}(\alpha)$ , allora  $\text{FIRST}(\beta) \cap \text{FOLLOW}(A) = \emptyset$ ;  
 (b) se  $\epsilon \in \text{FIRST}(\beta)$ , allora  $\text{FIRST}(\alpha) \cap \text{FOLLOW}(A) = \emptyset$ .

La Figura 4.7 mostra la tabella di parsing per la grammatica dell’Esempio 4.29, per la quale abbiamo calcolato i FIRST e FOLLOW nell’Esempio 4.30. Siccome non vi sono caselle riempite con più di una produzione, si tratta di una grammatica LL(1).

**Esempio 4.32** La seguente grammatica non è LL(1):

$S \rightarrow A$	$B$
$A \rightarrow ab$	$cd$
$B \rightarrow ad$	$cb$

Infatti si calcola facilmente

$$\begin{aligned}\text{FIRST}(A) &= \{a, c\} \\ \text{FIRST}(B) &= \{a, c\} \\ \text{FIRST}(S) &= \text{FIRST}(A) \cup \text{FIRST}(B).\end{aligned}$$

Siccome  $a, c \in \text{FIRST}(A) \cap \text{FIRST}(B)$  la tabella di parsing avrà conflitti sia in  $M[S, a]$  che in  $M[S, c]$  che conterranno entrambe  $S \rightarrow A$  e  $S \rightarrow B$ . Possiamo però dare una grammatica LL(1) equivalente: prima di tutto espandiamo le produzioni per  $A$  e  $B$ :

$$S \rightarrow ab \mid cd \mid ad \mid cb$$

Fattorizzando e riorganizzando si ottiene

$$\begin{array}{l|l|l} S & \rightarrow & aT \\ T & \rightarrow & b \quad | \quad d \end{array}$$

che è LL(1), come si vede immediatamente.

#### 4.7.4 Parser LL(1) non ricorsivi

Lo schema ricorsivo generale della Figura 4.6 corrisponde ad un automa a pila non deterministico: la pila è implicita nella ricorsione.<sup>8</sup> Se la grammatica è LL(1) possiamo esplicitare la pila e trasformare quello schema in un semplice programma iterativo, il cui controllo deterministico è dettato dalla tabella di parsing (che è riempita in modo univoco perché la grammatica è LL(1)). Il programma è descritto nella Figura 4.8;  $\$$  è usato anche come simbolo di fondo pila.

**Esempio 4.33** Le tabelle che seguono sono, nell'ordine, le produzioni di una grammatica, i suoi insiemi FIRST e FOLLOW, la tabella di parsing top-down.

		FIRST()	FOLLOW()
$S \rightarrow aAB$	$S$	$a$	$\$$
$A \rightarrow C \mid D$	$A$	$c, d, \epsilon$	$b$
$B \rightarrow b$	$B$	$b$	$\$$
$C \rightarrow c \mid \epsilon$	$C$	$c, \epsilon$	$b$
$D \rightarrow d$	$D$	$d$	$b$

	$a$	$b$	$c$	$d$	$\$$
$S$	$S \rightarrow aAB$				
$A$		$A \rightarrow C$	$A \rightarrow C$	$A \rightarrow D$	
$B$		$B \rightarrow b$			
$C$		$C \rightarrow \epsilon$	$C \rightarrow c$		
$D$				$D \rightarrow d$	

<sup>8</sup>Quando alla linea 5 si chiama una nuova funzione  $X_i^h()$ , la funzione corrente  $A()$  viene sospesa in pila, per essere ripresa quando  $X_i^h()$  restituirà il controllo al chiamante. I dettagli dell'implementazione della ricorsione mediante pila sono discussi nel Capitolo 7.

Input: una stringa di terminali  $w$ ; una tabella di parsing  $M$ .  
Output: se  $w \in L[G]$ , una derivazione sinistra per  $w$ ;  
altrimenti un errore.

```
Inizializza la pila con  $S\$$  (la cima della pila è a sinistra);  
Inizializza  $X$  con  $S$  (top della pila);  
Inizializza input con  $w\$$ ;  
Inizializza  $ic$  con il primo carattere di input;  
  
while ( $X \neq \$$ ) { /* finché la pila non è vuota */  
    if ( $X$  è un terminale)  
        if ( $X = ic$ ) { /*  $X$  fa match con  $ic$  */  
            pop  $X$  dalla pila;  
            avanza  $ic$  su input;  
        }  
        else errore();  
    else /*  $X$  è un non terminale */  
        if ( $M[X, ic] = X \rightarrow Y_1 \dots Y_n$ ){  
            pop  $X$  dalla pila;  
            push  $Y_1 \dots Y_n$  sulla pila ( $Y_1$  in cima);  
            output la produzione  $X \rightarrow Y_1 \dots Y_n$ ;  
        }  
        else errore();  
    }  
    if ( $ic \neq \$$ ) errore();
```

Figura 4.8 Parser LL(1) non ricorsivo.

La Figura 4.9(a) riporta le mosse del parser LL(1) guidato da questa tabella durante l'analisi della stringa  $adb\$$ . La computazione può essere descritta da una tavola, nella quale le righe riportano, nell'ordine: l'output generato dal parser; la configurazione della pila (la cima è a sinistra); l'input ancora da consumare. Il simbolo sulla cima della pila e il primo simbolo dell'input (detto *simbolo di lookahead*) sono usati per cercare nella tabella di parsing la produzione da utilizzare. Le righe con output vuoto corrispondono alle azioni di match. La tabella (b) riporta le mosse dello stesso parser sulla stringa  $abb\$$ , che non appartiene al linguaggio della grammatica. Si osservi che, nonostante la pila sia vuota, il parser non accetta la stringa, perché l'input non è stato consumato tutto.

#### 4.7.5 Grammatiche LL(k)

Un parser LL(1) usa un simbolo dell'input (insieme alla pila) per effettuare le sue scelte, prima che quel simbolo sia attualmente consumato (il che avviene solo quando in cima alla pila c'è un terminale che fa match con l'input corrente). Si dice dunque che il parser "guarda avanti un simbolo", o anche che usa un *simbolo di lookahead*: questo è il significato di 1 in LL(1).

OUTPUT	PILA	INPUT	OUTPUT	PILA	INPUT
Start	\$	adb\$	Start	\$	abb\$
$S \rightarrow aAB$	aAB\$	adb\$	$S \rightarrow aAB$	aAB\$	abb\$
	AB\$	db\$		AB\$	bb\$
$A \rightarrow D$	DB\$	db\$	$A \rightarrow C$	CBS	bb\$
$D \rightarrow d$	dB\$	db\$	$C \rightarrow \epsilon$	B\$	bb\$
	B\$	b\$	$B \rightarrow b$	b\$	bb\$
$B \rightarrow b$	b\$	b\$		\$	b\$
	\$	\$	Errore!		
OK!			(a)		
			(b)		

Figura 4.9 Due sequenze di parsing: (a) con successo; (b) con fallimento.

Non è concettualmente difficile progettare un parser top-down che basi le sue scelte su più di un simbolo di lookahead. Fissato  $k \geq 1$ , si definiscono gli insiemi  $\text{FIRST}_k(X)$  e  $\text{FOLLOW}_k(X)$ , che generalizzano quelli che abbiamo già visto e contengono (invece di singoli simboli) stringhe lunghe al più  $k$ . Con questi elementi si definisce opportunamente un parser che usa  $k$  simboli di lookahead, la cui tabella di parsing ha come colonne tutte le stringhe di terminali di lunghezza al più  $k$ . Analogamente a quanto abbiamo fatto per le grammatiche  $\text{LL}(1)$ , diciamo che una grammatica è  $\text{LL}(k)$  se il parser risultante è deterministico. Per quanto si possa mostrare che, per ogni  $k$ , esistono linguaggi che sono generati da grammatiche  $\text{LL}(k+1)$  ma non da grammatiche  $\text{LL}(k)$ , si tratta di una situazione di scarso rilievo pratico. Gli unici parser utilizzati sono  $\text{LL}(1)$ : se una grammatica di interesse pratico non è  $\text{LL}(1)$  spesso la si può manipolare (fattorizzare, eliminare la ricorsione sinistra, disambiguare, ecc.), trasformandola in una equivalente  $\text{LL}(1)$ . Vale comunque il seguente importante risultato.

**Teorema 4.34** (i) Una grammatica ricorsiva sinistra non è  $\text{LL}(k)$ , per nessun  $k$ .  
(ii) Una grammatica ambigua non è  $\text{LL}(k)$ , per nessun  $k$ .

## 4.8 Parser bottom-up

L'interesse dei parser top-down predittivi (cioè  $\text{LL}(k)$ ) sta nella loro semplicità: anche per linguaggi di una certa complessità è possibile riempire manualmente una tabella di parsing  $\text{LL}(1)$ , senza l'ausilio di strumenti automatici. La loro debolezza, tuttavia, sta proprio nel loro dover *predire* la scelta della produzione da utilizzare, guardando solo  $k$  simboli della parte destra. I parser che introdurremo in questo paragrafo sono più potenti, perché, lavorando dal basso verso l'alto, sono in grado di posticipare la decisione sulla produzione da usare fino a quando non hanno visto sull'input tutta la parte destra di una produzione (e, nel caso di lookahead, anche  $k$  simboli più avanti). Si tratta tuttavia di parser più complessi,

Input: una stringa di terminali  $w$ ; una grammatica  $G$ ;  $S$  è il simbolo iniziale di  $G$ .  
Output: se  $w \in L[G]$ , una sequenza di esecuzione produce una derivazione destra al contrario per  $w$ .  
Inizializza la pila con  $\$$ ;  
Inizializza input con  $w\$$ ;  
Inizializza  $ic$  con il primo carattere di input;  
while ( $\text{Top}(\text{pila}) \neq S$  and  $ic \neq \$$ ) {  
 Scegli non deterministicamente una delle seguenti azioni:  
 SHIFT: push  $ic$  sulla pila;  
 avanza  $ic$  su input;  
 REDUCE: if ( $\alpha$  è in cima alla pila) and  
 ( $A \rightarrow \alpha$  è una produzione di  $G$ ) {  
 pop  $\alpha$  dalla pila;  
 push  $A$  sulla pila;  
 output la produzione  $A \rightarrow \alpha$ ;  
 }  
 }  
}

Figura 4.10 Parser shift-reduce non deterministico.

la cui generazione richiede l'uso di strumenti automatici, vista la dimensione delle strutture dati che sono necessarie.

I parser bottom-up che consideriamo sono detti parser *a spostamento e riduzione* (*shift-reduce*) e sono automi a pila non deterministici che alternano tra due possibili azioni:

1. *shift*: un simbolo terminale è spostato dall'input sulla pila;
2. *reduce*: una serie di simboli (terminali e non terminali) sulla cima della pila corrisponde alla parte destra di una produzione  $A \rightarrow \alpha$ : la stringa  $\alpha$  viene rimossa dalla pila e sostituita con  $A$  (“ $\alpha$  viene ridotta ad  $A$ ”).

La Figura 4.10 riporta lo schema generale di un parser non deterministico di questo tipo: se esiste una sequenza di scelte che permette la terminazione del ciclo, l'input è accettato (e l'esecuzione ha prodotto una derivazione destra al contrario).

È utile a questo punto rivedere l'Esempio 4.16 e, in particolare, la ricostruzione bottom-up di un albero di derivazione che abbiamo dato nella Figura 4.4. La Figura 4.11 riporta la grammatica di quell'esempio, insieme ad una possibile sequenza d'esecuzione di un parser shift-reduce sulla stringa  $a + b * b$  (la pila ha la sua cima a destra). Si osservi che su ogni riga concatenando pila e input otteniamo una serie di forme sentenziali che danno esattamente la derivazione destra dal basso verso l'alto.

È evidente che lo schema è non deterministico in due modi distinti: in primo luogo c'è *non determinismo shift/reduce*: quando sulla pila c'è una stringa che è

$E \rightarrow T \mid T + E \mid T - E$			
$T \rightarrow A \mid A * T$			
$A \rightarrow a \mid b \mid (E)$			
PILA	INPUT	AZIONE	OUTPUT
1 \$	a + b * b\$	shift	
2 \$a	+ b * b\$	reduce	$A \rightarrow a$
3 \$A	+ b * b\$	reduce	$T \rightarrow A$
4 \$T	+ b * b\$	shift	
5 \$T +	* b\$	shift	
6 \$T + b	* b\$	reduce	$A \rightarrow b$
7 \$T + A	* b\$	shift	
8 \$T + A *	b\$	shift	
9 \$T + A * b	\$	reduce	$A \rightarrow b$
10 \$T + A * A	\$	reduce	$T \rightarrow A$
11 \$T + A * T	\$	reduce	$T \rightarrow A * T$
12 \$T + T	\$	reduce	$E \rightarrow T$
13 \$T + E	\$	reduce	$E \rightarrow T + E$
14 \$E	\$	stop	OK!

Figura 4.11 Una sequenza di esecuzione di un parser shift-reduce.

la parte destra di una produzione il parser può sia ridurre con quella produzione sia spostare l'input (uno shift è sempre possibile se l'input non è stato tutto consumato). Ad esempio, alla linea 7 la cima della pila ha  $A$ , che è la parte destra della produzione  $T \rightarrow A$ ; ma il parser invece sposta l'input  $*$  sulla pila, a differenza di quanto aveva fatto alla linea 3, quando la testa della pila era ancora  $A$ . C'è, poi, un *non determinismo reduce/reduce*: può capitare che la testa della pila corrisponda alla parte destra di più produzioni. Nell'esempio questo accade alla linea 11: si può sia ridurre il solo  $T$  (il simbolo in testa alla pila) con la produzione  $E \rightarrow T$ , sia ridurre  $A * T$  con la produzione  $T \rightarrow A * T$  (che è la scelta seguita nell'esempio alla linea 11; mentre alla linea 12 si sceglie di ridurre la  $T$  a  $E$ ). Dovremo dunque determinare con quali informazioni il non determinismo shift/reduce e quello reduce/reduce possano essere risolti. In estrema sintesi ciò avverrà sia usando il prossimo simbolo dell'input (e questo non è molto diverso dai parser LL) sia mediante informazioni abbastanza sofisticate sulla configurazione della pila.

Prima di abbandonare l'esempio osserviamo un altro paio di aspetti importanti. In primo luogo, una riduzione avviene sempre sulla cima della pila: il parser non riduce mai "saltando" qualche simbolo sulla pila. Inoltre, anche se in un determinato momento sulla pila non c'è la parte destra di una produzione (nell'esempio accade alle linee 1, 5 e 8), c'è comunque il prefisso di una parte destra, che può essere completato mediante una serie di spostamenti. Questo invarianto è uno dei cardini dei parser LR, che introdurremo tra poco.

#### 4.8.1 Prefissi e maniglie

L'informazione fondamentale che un parser shift-reduce deve elaborare consiste nella struttura delle stringhe che si trovano sulla testa della pila. Alla linea 11 della Figura 4.11 il non determinismo reduce/reduce è risolto scegliendo la produzione  $T \rightarrow A * T$  invece di  $E \rightarrow T$ . Il motivo della scelta è che essa produce sulla pila una nuova testa che è a sua volta la parte destra di una produzione. Se il parser avesse scelto di ridurre  $T$  a  $E$ , la pila si sarebbe trovata nella configurazione  $$T + A * E$ : un vicolo cieco perché non ha sulla testa la parte destra di nessuna produzione, e inoltre non è neppure il prefisso di una parte destra che possa essere completato con qualche spostamento dell'input.

Fissata una grammatica  $G$  senza simboli inutili, un suo *prefisso viabile* (*viable prefix*) è una stringa di terminali e non terminali che può apparire sulla pila del parser shift-reduce per  $G$  durante una computazione che accetta l'input. Equivalentemente, possiamo dare la definizione usando direttamente la grammatica (e non il parser): una stringa  $\gamma \in (T \cup NT)^*$  è un prefisso viabile per  $G$  sse esiste una derivazione destra

$$S \Rightarrow^* \delta A y \Rightarrow \delta \alpha \beta y = \gamma \beta y$$

per qualche stringa  $\delta \in (T \cup NT)^*$  e  $y \in T^*$  e per una produzione  $A \rightarrow \alpha \beta$ . Inoltre, anche  $S$  è un prefisso viabile. Un prefisso viabile è *completo* se  $\beta = \epsilon$ ; in tal caso  $\alpha$  è una *maniglia* (*handle*) di  $\gamma y$ , ed è unica se la grammatica non è ambigua. Laddove dunque abbiamo detto sin qui che "sulla cima della pila si trova la parte destra di una produzione" avremmo potuto dire con più precisione: la cima della pila presenta una maniglia.

Se vogliamo ridurre (e potenzialmente eliminare) il non determinismo di un parser shift-reduce dobbiamo quindi: (i) riconoscere le maniglie sulla cima della pila e ridurle; (ii) scegliere solo quelle riduzioni che producono sulla pila un nuovo prefisso viabile; (iii) scegliere spostamenti che completino i prefissi viabili sulla pila e facciano comparire una nuova maniglia.

A questo scopo ci viene in soccorso uno dei risultati più importanti della teoria dei parser bottom-up: i prefissi viabili di un linguaggio libero costituiscono un linguaggio regolare. Il parser può quindi usare un opportuno automa finito per riconoscere i prefissi viabili e decidere quali azioni compiere. Questo DFA non lavora sull'input (che sappiamo essere inutile: gli automi finiti non sono in grado di riconoscere un linguaggio libero), ma sulla pila. Tutte le volte che si mette sulla pila un nuovo simbolo di input (un token), il parser richiama l'automa, che analizza la stringa sulla pila. Se la pila contiene un prefisso viabile completo, il parser riduce (la scelta della produzione potrà dipendere dai simboli in input non ancora consumati, in caso di lookahead  $k > 0$ ). Se la pila contiene un prefisso viabile incompleto, il parser sposta. Se la pila non contiene un prefisso viabile, si genera un errore, perché l'input non appartiene al linguaggio generato dalla grammatica. Ovviamente usando questa tecnica non è detto che il parser divenga deterministico: per alcune grammatiche il DFA darà informazioni conflittuali (sia shift che reduce; o più di una produzione mediante cui ridurre). Vedremo però

che alcune interessanti classi di grammatiche sono in grado di dare parser deterministici, sfruttando in modo opportuno le informazioni di FOLLOW e i simboli di lookahead.

Anche se concettualmente ad ogni modifica della pila il DFA la riscandidisce da capo, questo non è necessario, perché ogni prefisso di un prefisso viabile è anch'esso un prefisso viabile. La pila viene modificata in due modi:

1. Con uno spostamento, la pila passa da  $\$ \gamma$  a  $\$ \gamma x$ . In tal caso il DFA si trovava nello stato  $s$  (corrispondente ad un prefisso viabile incompleto) dopo aver elaborato  $\$ \gamma$ : basta far ripartire il DFA da  $s$  con l'input  $x$ .
2. Con una riduzione mediante la produzione  $A \rightarrow \alpha$ , la pila passa da  $\$ \gamma \alpha$  a  $\$ \gamma A$ . In tal caso il DFA si trovava nello stato  $s$  (corrispondente ad un prefisso viabile completo) dopo aver elaborato  $\$ \gamma \alpha$ : non c'è bisogno di far ripartire il DFA dalla base della pila; basta ripristinare lo stato in cui si esso trovava subito prima di elaborare il primo simbolo di  $\alpha$  e fornirgli il simbolo  $A$ .

Gli stati dell'automa finito, insomma, codificano tutta l'informazione racchiusa nella pila sotto al simbolo che viene elaborato dal DFA. Per realizzare un parser, dunque, è conveniente mettere in pila non i simboli della grammatica, ma gli stati del DFA (uno stato corrisponde sempre ad un unico simbolo, mentre più stati possono essere associati allo stesso simbolo: stati diversi corrispondono a ruoli diversi del simbolo nei prefissi viabili). In questo modo non c'è bisogno di scandire la pila ad ogni modifica.

Vedremo ora come avviene la costruzione del DFA associato ai prefissi viabili e, quindi, come derivare da esso una tabella per un parser bottom-up.

#### 4.8.2 Item e automa LR(0)

Un *item LR(0)* (diremo semplicemente *item*, quando non ci sarà confusione) per una grammatica  $G$  è una produzione di  $G$  nella quale è indicata esplicitamente una posizione della sua parte destra. Se indichiamo una posizione con un "punto", la produzione  $A \rightarrow XYZ$  genera i quattro item

$$\begin{aligned} A &\rightarrow .XYZ \\ A &\rightarrow X.YZ \\ A &\rightarrow XY.Z \\ A &\rightarrow XY.Z. \end{aligned}$$

Uno stato del DFA dei prefissi viabili (che chiameremo *automa canonico LR(0)*) è costituito da un insieme di item: intuitivamente, il punto indica quanta parte della produzione è già stata analizzata. Se lo stato dell'automa contiene  $A \rightarrow \alpha\beta$ , significa che il DFA sta cercando di riconoscere un prefisso viabile che termina in  $\alpha\beta$ ; di questo prefisso,  $\alpha$  è già sulla pila mentre ci si aspetta che l'input contenga (o possa essere ridotto a)  $\beta$ . Un item con il punto in fondo a destra indica, dunque, la presenza sulla pila di una maniglia.

**Automa canonico LR(0)** Vedremo prima come ottenere l'automa canonico  $LR(0)$  passando per un NFA; in seguito daremo la procedura diretta. Fissiamo una grammatica  $G = (NT, T, R, S)$ : aggiungiamo in primo luogo un nuovo stato iniziale  $S' \notin NT$  e la nuova produzione  $S' \rightarrow S$ . La grammatica risultante si dice *grammatica aumentata* e serve a distinguere un uso di  $S$  come passo intermedio di una derivazione dal suo uso come inizio. Costruire un NFA che riconosca i prefissi viabili per  $G$  non è difficile: gli stati coincidono con gli item e le transizioni sono definite facendo "viaggiare" il punto sugli item in tutti i modi possibili. L'unica attenzione si deve porre quando l'automa si trova in uno stato  $[A \rightarrow \alpha.X\beta]$  con  $X$  non terminale: in questo caso il prefisso può continuare sia con  $X$  stesso, sia *con una qualsiasi stringa derivabile da X*; per trattare questo caso aggiungiamo delle  $\epsilon$ -transizioni verso gli stati  $[X \rightarrow .\gamma]$ . Riassumendo, lo NFA dei prefissi viabili di  $G$  è dato da:

- gli stati sono costituiti dagli item  $LR(0)$  della grammatica aumentata;
- $[S' \rightarrow .S]$  è lo stato iniziale;
- dallo stato  $[A \rightarrow \alpha.X\beta]$  c'è una transizione allo stato  $[A \rightarrow \alpha X.\beta]$  etichettata  $X$ , per  $X \in T \cup NT$ ;
- dallo stato  $[A \rightarrow \alpha.X\beta]$ , per  $X \in NT$  e per ogni produzione  $X \rightarrow \gamma$ , c'è una  $\epsilon$ -transizione verso lo stato  $[X \rightarrow .\gamma]$ ;
- non è necessario definire gli stati finali: questo NFA non è usato per riconoscere un linguaggio, ma solo come ausilio del parser.

L'*automa canonico LR(0) per G* è il DFA che si ottiene rimuovendo dal NFA appena definito tutti gli stati irraggiungibili e applicando quindi la costruzione dei sottinsiemi (Paragrafo 3.3.3). In modo diretto, possiamo ottenere l'automa  $LR(0)$  applicando le seguenti funzioni, nelle quali  $I$  e  $J$  indicano insiemi di item e  $X \in T \cup NT$ : la prima funzione calcola la  $\epsilon$ -chiusura; la seconda l'insieme di stati del NFA raggiungibili con un fissato simbolo a partire da un insieme di stati.

```
Clos(I) {
    ripeti finché I è modificato{
        per ogni item  $A \rightarrow \alpha.X\beta \in I$ 
            per ogni produzione  $X \rightarrow \gamma$ 
                aggiungi  $X \rightarrow .\gamma$  a I;
    }
    return I;
}
```

```
Goto(I, X) {
    inizializza J = {};
    per ogni item  $A \rightarrow \alpha.X\beta \in I$ 
        aggiungi  $A \rightarrow \alpha X.\beta$  a J;
    return Clos(J);
```

Possiamo ora costruire l'automa canonico  $LR(0)$  iniziando dallo stato iniziale e procedendo incrementalmente. Useremo  $\mathcal{S}$  per l'insieme degli stati;  $\delta$  è la funzione di transizione.

```
inizializza  $\mathcal{S} = Clos(\{S' \rightarrow .S\})$ ;
inizializza  $\delta = \emptyset$ ;
ripeti finché  $\mathcal{S}$  e  $\delta$  vengono modificati
    per ogni stato  $I \in \mathcal{S}$ 
        per ogni item  $A \rightarrow \alpha.X\beta \in I$ {
            sia  $J = Goto(I, X)$ ;
            aggiungi  $J$  a  $\mathcal{S}$ ;
            aggiungi  $[A \rightarrow \alpha X.\beta] \in \delta$  con  $I \rightarrow J$  come etichetta;
```

```

aggiungi  $\delta(I, X) = J$  a  $\delta$ ;
}
return  $S$  e  $\delta$ ;

```

**Esempio 4.35** Descriviamo la costruzione dell'automa canonico LR(0) su di un semplice esempio, anticipando via via anche le mosse che il parser farà in corrispondenza alle transizioni dell'automa. Consideriamo la grammatica con terminali  $\{a, (, )\}$ , simbolo iniziale  $S$  e produzioni

$$1 \quad S \rightarrow (S) \quad 2 \quad S \rightarrow A \quad 3 \quad A \rightarrow a$$

e aumentiamola aggiungendo il nuovo simbolo iniziale  $S'$  e la produzione

$$0 \quad S' \rightarrow S$$

Lo stato iniziale è la chiusura di  $S' \rightarrow .S$ :

$$\text{Stato 1: } S' \rightarrow .S \quad S \rightarrow .(S) \quad S \rightarrow .A \quad A \rightarrow .a$$

Si osservi che la chiusura inserisce tutti gli item “iniziali” di  $S$ , tra cui anche  $S \rightarrow .A$ , che quindi porta con sé anche l’item iniziale di  $A$ . Ora determiniamo le transizioni che originano dallo stato 1. Negli item che fanno parte dello stato 1, il punto può attraversare i terminali  $a$  e “(”, oppure i non terminali  $S$  e  $A$ . Con l’etichetta  $a$  si passa a

$$\text{Stato 2: } A \rightarrow a.$$

Attraversando la parentesi aperta, lo stato di arrivo conterrà certo l’item  $S \rightarrow (:S)$ , che si porta dietro, per chiusura, gli item iniziali di  $S$  e, quindi, di  $A$ :

$$\text{Stato 3: } S \rightarrow (.S) \quad S \rightarrow .(S) \quad S \rightarrow .A \quad A \rightarrow .a$$

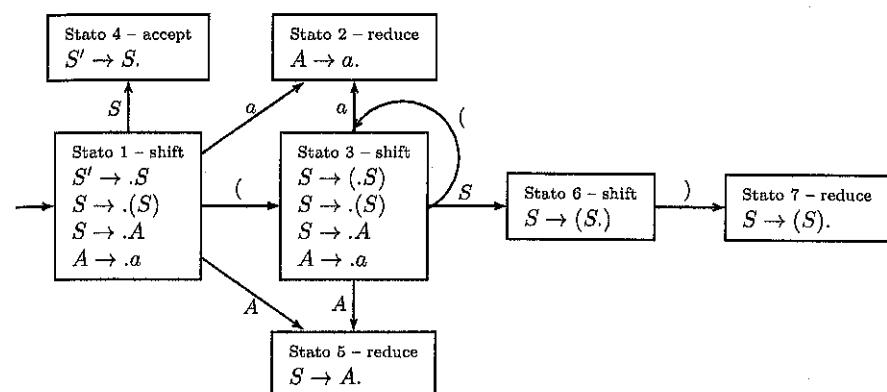
Le due mosse che abbiamo esaminato sin qui dallo stato 1 (agli stati 2 e 3 mediante terminali) corrispondono entrambe a *spostamenti*: il parser consuma l’input e mette sulla pila lo stato di destinazione (che corrisponde al terminale consumato). Ancora dallo stato 1 abbiamo una transizione etichettata  $S$ , che conduce a

$$\text{Stato 4: } S' \rightarrow S.$$

Quando l’automa è nello stato 1 e può effettuare questa transizione significa che il parser ha appena ridotto una produzione con testa  $S$ : la transizione consuma  $S$  e prepara il parser ad una nuova azione. Mosse del parser di questo tipo (cioè che corrispondono a transizioni dell’automa LR(0) etichettate con un non terminale) sono dette *goto*. Una transizione analoga è quella con etichetta  $A$  che dallo stato 1 porta a

$$\text{Stato 5: } S \rightarrow A.$$

Adesso dobbiamo analizzare le transizioni che originano negli stati 2, 3, 4 e 5. Lo stato 2 contiene un solo item, che corrisponde ad una maniglia sulla pila (il punto



**Figura 4.12** Automa canonico LR(0) per la grammatica dell’Esempio 4.35.

è all'estrema destra). Non ci sono transizioni che originano da 2 nell'automa LR(0). In corrispondenza di questo stato il parser *riduce* secondo la produzione  $A \rightarrow a$ : toglie dalla pila tanti stati quanti sono i simboli a destra della produzione (toglie tutta la maniglia: nel nostro caso un solo stato) e si prepara per una nuova mossa (che sarà una goto).

Dallo stato 3 origina una transizione goto (data dal primo item) verso

$$\text{Stato 6: } S \rightarrow (S.)$$

Il secondo item dello stato 3 genera uno spostamento di “(” verso lo stato 3 stesso; il terzo item dà una transizione goto (attraverso  $A$ ) verso lo stato 5; e infine il quarto item, spostando  $a$ , dà una transizione verso lo stato 2.

Dagli stati 4 e 5 non ci sono transizioni uscenti. Lo stato 5 è uno stato che provoca una riduzione. Lo stato 4 è uno stato particolare, perché corrisponde alla maniglia della produzione del nuovo simbolo iniziale. Se si arriva in questo stato, e l’input è stato tutto consumato, cioè il prossimo simbolo è \$, significa che è stata completata una derivazione destra: il parser accetta la stringa.

L’ultimo stato da considerare è lo stato 6, da cui origina una transizione di spostamento (consumando la parentesi chiusa) verso

$$\text{Stato 7: } S \rightarrow (S.)$$

che a sua volta non ha transizioni uscenti. La Figura 4.12 riporta il diagramma dell’automa canonico LR(0) che abbiamo costruito. In corrispondenza degli stati sono state anche annotate le corrispondenti azioni del parser.

### 4.8.3 Tabella e parser LR

Le informazioni codificate da un automa canonico LR(0) possono essere disposte in forma tabellare, creando una *tabella di parsing LR*. I parser LR hanno tutti la

stessa struttura e utilizzano tutti lo stesso tipo di tabella. Ciò che cambia passando da un parser LR(0) a parser più potenti è solo il modo in cui la tabella è riempita. Una tabella di parsing LR è una matrice bidimensionale, le cui righe sono indicate sugli stati dell'automa LR(0) (o LR( $k$ ), ecc.) e le cui colonne variano sui simboli della grammatica (più \$). La casella  $M[s, X]$  contiene le azioni che il parser LR compie quando si trova nello stato  $s$  in corrispondenza del simbolo  $X$ .<sup>9</sup>

Vediamo come la tabella è riempita nel semplice caso LR(0):

### Riempire una tabella di parsing LR(0)

Per ogni stato  $s$ :

1. se  $x \in T$  e  $s \xrightarrow{x} t$  nell'automa LR(0), inserisci SHIFT  $t$  in  $M[s, x]$ ;
2. se  $A \rightarrow \alpha \in s$  e  $A \neq S'$ , inserisci REDUCE  $A \rightarrow \alpha$  in  $M[s, x]$  per tutti gli  $x \in T \cup \{\$\}$ ;
3. se  $S' \rightarrow S \in s$ , inserisci ACCEPT in  $M[s, \$]$ ;
4. se  $A \in NT$  e  $s \xrightarrow{A} t$  nell'automa LR(0), inserisci GOTO  $t$  in  $M[s, A]$ .

Ogni casella rimasta vuota deve essere considerata una casella di *errore*. In generale questo procedimento potrà inserire più azioni in una stessa casella della tabella: in tal caso siamo in presenza di un *conflitto*. Ovviamente non possiamo avere un conflitto tra due SHIFT né tra due GOTO, perché l'automa LR(0) è deterministico. I conflitti possibili sono dunque SHIFT/REDUCE (il parser non ha informazioni deterministiche per scegliere tra spostare o ridurre) oppure REDUCE/REDUCE (il parser riduce ma non ha sufficienti informazioni per decidere con quale produzione). Una grammatica è LR(0) se ogni casella della tabella ha al più un elemento.

La Figura 4.13 riporta la tabella di parsing LR(0) per l'automa della Figura 4.12. Secondo l'uso, la tabella è divisa in una parte sinistra (la parte "azione", le cui colonne variano sui terminali) e in una parte destra (la parte "goto": colonne che variano sui non terminali). Nella tabella  $sn$  sta per SHIFT  $n$ ;  $rn$  per REDUCE con la produzione numero  $n$ ;  $gn$  per GOTO  $n$ ;  $ac$  per ACCEPT. Siccome la tabella non presenta conflitti, la grammatica dell'Esempio 4.35 è LR(0).<sup>10</sup>

Abbiamo finalmente tutti gli ingredienti per dare la struttura generale di un parser LR, che viene specializzato su di una grammatica mediante una tabella di parsing  $M$ .

Input: una stringa di terminali  $w$ ; una tabella di parsing LR  $M$ .

<sup>9</sup>Se  $X \in T$  questo è il simbolo di input; se  $X \in NT$  si tratta sempre di un'azione goto.

<sup>10</sup>Nel semplice caso LR(0), non essendoci lookahead ( $k = 0$ ), è solo lo stato a determinare se la prossima azione sarà SHIFT o REDUCE. Nella tabella, questo significa che, fissata una riga (uno stato), la parte azione contiene solo SHIFT, o solo REDUCE. I simboli terminali sulle colonne servono solo a codificare gli archi dell'automa LR(0), cioè il prossimo stato dell'automa. Nell'esempio, lo stato 1 causa SHIFT, e le colonne distinguono se, consumando l'input, si passa nello stato 2 o nello stato 3. In parser più sofisticati, che vedremo tra breve, il simbolo di lookahead (sulla colonna) giocherà il suo ruolo anche nel determinare l'azione da compiere.

	$a$	(	)	\$	$S$	$A$
1	$s_2$	$s_3$			$g_4$	$g_5$
2	$r_3$	$r_3$	$r_3$	$r_3$		
3		$s_3$			$g_6$	$g_5$
4					$ac$	
5	$r_2$	$r_2$	$r_2$	$r_2$		
6				$s_7$		
7	$r_1$	$r_1$	$r_1$	$r_1$		

Figura 4.13 Tabella di parsing LR(0) per l'automa della Figura 4.12.

Output: se  $w \in L[G]$ , una derivazione destra al contrario per  $w$ ; altrimenti un errore.

```
Inizializza la pila con  $\$s_0$ ;
/* cima della pila a destra;
    $s_0$  stato iniziale dell'automa */;
Inizializza input con  $w\$$ ;
Inizializza ic con il primo carattere di input;

while (true) {
    s = Top(pila);
    /* Top non rimuove la testa */

    case  $M[s, ic]$  of
        SHIFT  $t$ :{
            push  $t$  sulla pila;      /*  $t$  codifica  $ic$  */
            avanza ic su input;
        }

        ACCEPT:{output('accept'); break;}

        REDUCE  $A \rightarrow \alpha$ :{
            pop  $|\alpha|$  stati dalla pila; /* codificano la maniglia  $\alpha$  */
             $s_1 = \text{Top}(pila)$ ;          /*  $s_1$  contiene  $B \rightarrow \gamma \cdot A\delta$  */
            sia  $s_2 = M[s_1, A]$ ;          /*  $M[s_1, A]$  è GOTO  $s_2$  */
            push  $s_2$  sulla pila;
            output la produzione  $A \rightarrow \alpha$ ;
        }
        else errore();
    }
```

### 4.8.4 Parser SLR(1)

Un parser LR(0) è molto generoso nelle azioni REDUCE: non avendo lookahead, se si trova in uno stato che contiene un item completo, riduce comunque. Questa sua caratteristica lo rende incline a generare conflitti. Consideriamo la seguente grammatica aumentata, che genera il linguaggio delle *parentesi bilanciate*,

$$0 \quad S' \rightarrow S$$

$$1 \quad S \rightarrow (S)S$$

$$2 \quad S \rightarrow \epsilon$$

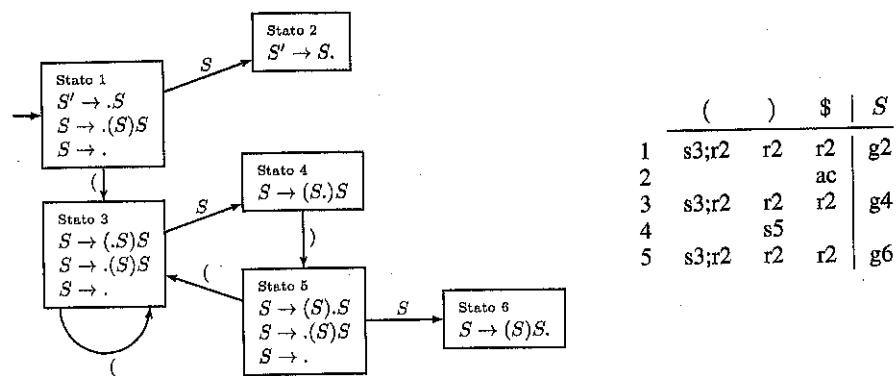


Figura 4.14 Automa e tabella LR(0) per la grammatica delle parentesi bilanciate.

il cui automa canonico LR(0) è riportato in Figura 4.14. Si vede subito che gli stati 1, 3 e 5 presentano un conflitto SHIFT/REDUCE: tutti e tre hanno una transizione uscente etichettata “(” (che comporta uno spostamento), ma contengono anche l’item  $S \rightarrow \cdot$ , che comporta una riduzione con la produzione 2. La grammatica non è dunque LR(0).

Se ammettiamo un lookahead, tuttavia, non è difficile limitare il numero di riduzioni. Supponiamo che il parser possa ridurre con una produzione  $A \rightarrow \alpha$ : ha senso ridurre solo se la derivazione può essere proseguita, il che avviene solo se sull’input c’è un simbolo che effettivamente può *seguire*  $A$ . Una tabella di parser SLR(1) (che sta per *simple LR* con un simbolo di lookahead) viene riempita in modo analogo alla tabella LR(0), con l’unica eccezione per le azioni REDUCE che sono inserite solo in corrispondenza dei terminali che appartengono al FOLLOW della testa della produzione.

### Riempire una tabella di parsing SLR(1)

Tutto come per una tabella LR(0) (pag. 116), con l’unica eccezione al punto

2. se  $A \rightarrow \alpha \in s$  e  $A \neq S'$ , inserisci REDUCE  $A \rightarrow \alpha$  in  $M[s, x]$  per tutti gli  $x \in \text{FOLLOW}(A)$ .

Una grammatica è SLR(1), o più semplicemente SLR, se ogni casella della tabella ha al più un elemento. Molti costrutti dei linguaggi di programmazione ammettono semplici grammatiche SLR(1).

Riprendendo la grammatica delle parentesi bilanciate, si calcola facilmente  $\text{FOLLOW}(S) = \{\}, \$\}$ , da cui si ottiene la seguente tabella SLR(1), nella quale non ci sono più conflitti: la grammatica è dunque SLR(1).

	(	)	\$		S
1	s3	r2	r2		g2
2			ac		
3	s3	r2	r2		g4
4			s5		
5	s3	r2	r2		g6

Il fatto che si stia usando del lookahead è evidente dal fatto che le righe della parte sinistra della tabella contengono azioni diverse (per esempio sia SHIFT sia REDUCE): il tipo di azione non è più determinato solo dallo stato, ma anche dal simbolo di lookahead.

### 4.8.5 Parser LR(1)

In un parser SLR(1) il lookahead è aggiunto solo alla fine della costruzione, per il riempimento della tabella. Parser più potenti possono essere ottenuti usando il lookahead *durante la costruzione dell’automa canonico*.

Un item  $LR(1)$  per una grammatica  $G$  è una coppia formata da un item  $LR(0)$  per  $G$  (detto *nucleo*, o *core*, dell’item  $LR(1)$ ) e un simbolo di lookahead, appartenente a  $T \cup \{\$\}$ ; per un item  $LR(1)$  useremo la notazione  $[A \rightarrow \alpha.\beta, x]$ .<sup>11</sup> Intuitivamente, quando l’automa si trova in uno stato che contiene l’item  $[A \rightarrow \alpha.\beta, x]$ , esso sta cercando di riconoscere la maniglia  $\alpha\beta$ ; di questa maniglia,  $\alpha$  è già sulla pila e si aspetta che sull’input si trovi una stringa derivabile da  $\beta x$ .

L’informazione di lookahead gioca il suo ruolo durante il calcolo della chiusura di uno stato. Ricordiamo che nell’automa non deterministico che genera l’automa canonico  $LR(0)$  si inseriscono  $\epsilon$ -transizioni quando il punto si trova a sinistra di un non terminale:

nell’automa canonico  $LR(0)$ :

dallo stato  $[A \rightarrow \alpha.X\beta, a]$ , per  $X \in NT$  e per ogni produzione  $X \rightarrow \gamma$ , c’è una  $\epsilon$ -transizione verso lo stato  $[X \rightarrow \cdot\gamma, a]$ .

Nel caso  $LR(1)$ , se siamo in uno stato (che contiene)  $[A \rightarrow \alpha.X\beta, a]$ , con  $X$  non terminale, ci si aspetta sull’input una stringa che si deriva da  $X\beta a$ ; in particolare potrebbe essere una stringa che si deriva da  $X$  mediante un’opportuna produzione  $X \rightarrow \gamma$ , il cui simbolo di lookahead  $b$  dovrà essere uno dei simboli iniziali di  $\beta a$ . Riassumendo:

nell’automa canonico  $LR(1)$ :

dallo stato  $[A \rightarrow \alpha.X\beta, a]$ , per  $X \in NT$  e per ogni produzione  $X \rightarrow \gamma$ , c’è una  $\epsilon$ -transizione verso lo stato  $[X \rightarrow \cdot\gamma, b]$  per ogni  $b \in \text{FIRST}(\beta a) \cap T$ .

L’automa canonico  $LR(1)$ , dunque, avrà (molti) più stati (perché gli item sono moltiplicati dalla presenza dei lookahead), ma sarà anche molto più selettivo

<sup>11</sup>Come nel caso LL, possiamo generalizzare la nozione in quella di item  $LR(k)$ , che contiene  $k$  simboli di lookahead. Le relative grammatiche sono dette grammatiche  $LR(k)$ ; in tutti i casi di interesse pratico,  $k = 1$ .

nelle transizioni, permettendo di ridurre il numero dei conflitti rispetto ad un automa LR(0). Le seguenti funzioni permettono di costruire l'automa canonico LR(1) in modo diretto.

```
Clos(I) {
    ripeti finché I è modificato{
        per ogni item  $[A \rightarrow \alpha.X\beta, a] \in I$ 
        per ogni produzione  $X \rightarrow \gamma$ 
        per ogni  $b \in \text{FIRST}(\beta a) \cap T$ 
            aggiungi  $[X \rightarrow .\gamma, b]$  a I;
    }
    return I;
}

Goto(I, X) {
    inizializza  $J = \emptyset$ ;
    per ogni item  $[A \rightarrow \alpha.X\beta, a] \in I$ 
        aggiungi  $[A \rightarrow \alpha.X.\beta, a]$  a J;
    return Clos(J);
}
```

La costruzione incrementale dell'automa canonico LR(1) è la stessa che abbiamo già visto nel caso LR(0) a pag. 113; lo stato iniziale è  $\text{Clos}([S' \rightarrow .S, \$])$ . Un parser LR(1) si ottiene specializzando quello generico del Paragrafo 4.8.3 con una tabella LR(1), ottenuta con il metodo seguente.

### Riempire una tabella di parsing LR(1)

Per ogni stato  $s$ :

- se  $x \in T$  e  $s \xrightarrow{x} t$  nell'automa LR(1), inserisci SHIFT  $t$  in  $M[s, x]$ ;
- se  $[A \rightarrow \alpha., x] \in s$  e  $A \neq S'$ , inserisci REDUCE  $A \rightarrow \alpha$  in  $M[s, x]$ ;
- se  $[S' \rightarrow S., \$] \in s$ , inserisci ACCEPT in  $M[s, \$]$ ;
- se  $A \in NT$  e  $s \xrightarrow{A} t$  nell'automa LR(1), inserisci GOTO  $t$  in  $M[s, A]$ .

Ogni casella rimasta vuota è un errore. Una grammatica è LR(1) se ogni casella della tabella ha al più un elemento.

### Esempio 4.36 La seguente grammatica aumentata

$$\begin{array}{lll} 0 & S' \rightarrow S & 1 & S \rightarrow V = E & 2 & S \rightarrow E \\ 3 & E \rightarrow V & 4 & V \rightarrow x & 5 & V \rightarrow *E \end{array}$$

non è SLR(1): lasciando la costruzione completa dell'automa LR(0) per esercizio, basta osservare che dallo stato iniziale, con una transizione etichettata  $V$ , si passa ad uno stato composto da

$$S \rightarrow V. = E \rightarrow V.$$

Nella tabella SLR(1) questo provoca un conflitto SHIFT/REDUCE in corrispondenza di  $=$  (perché  $=$  è in  $\text{FOLLOW}(E)$ ). Si tratta però di una grammatica LR(1). Il suo automa canonico LR(1) è riportato in Figura 4.15 nel quale per concisione si sono abbreviati su una stessa riga più item che si distinguono solo per il simbolo di lookahead (ad esempio, la linea  $[V \rightarrow .x, \$, =]$  sta per i due item  $[V \rightarrow .x, \$]$  e  $[V \rightarrow .x, =]$ ). Dall'automa otteniamo la tabella di parsing LR(1) seguente.

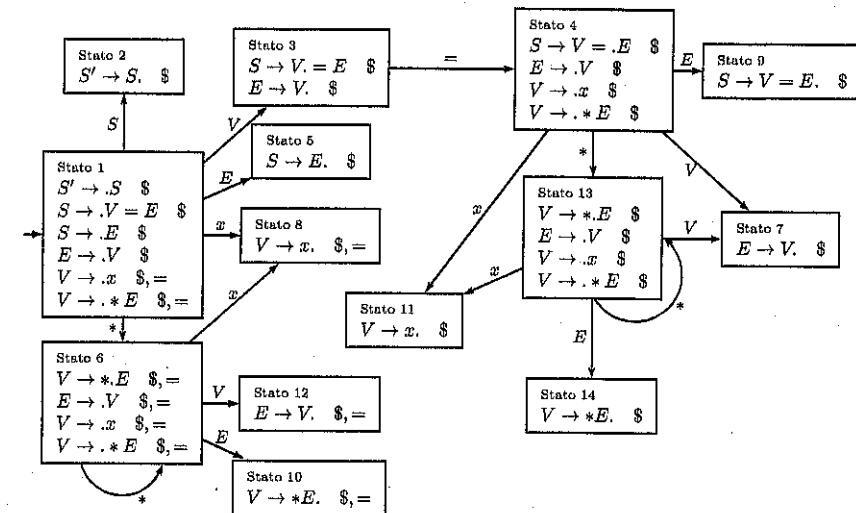


Figura 4.15 Automa LR(1) per la grammatica dell'Esempio 4.36.

	$x$	$*$	$=$	$\$$	$S$	$E$	$V$
1	s8	s6				g2	
2				a		g5	g3
3			s4	r3			
4	s11	s13				g9	g7
5				r2			
6	s8	s6				g10	g12
7				r3			
8				r4	r4		
9					r1		
10				r5	r5		
11					r4		
12				r3	r3		
13	s11	s13				g14	g7
14				r5			

Lo stato dell'automa LR(0) che generava conflitti corrisponde qui allo stato 3: il conflitto non è presente nella tabella LR(1) perché il simbolo di lookahead  $\$$  dell'item è più selettivo sull'azione da compiere di quanto lo siano i simboli in  $\text{FOLLOW}(E)$  (che sono usati per la tabella SLR(1)).

### 4.8.6 Parser LALR(1)

Le grammatiche LR(1) sono sufficientemente potenti da descrivere praticamente tutti i costrutti dei linguaggi di programmazione. Hanno tuttavia il difetto di generare strutture dati molto grandi: per un linguaggio reale di media grandezza (diciamo C, per fissare le idee), l'automa canonico LR(1) ha parecchie migliaia di stati (contro le centinaia di stati dell'automa LR(0) per lo stesso linguaggio),

che danno luogo a una tabella di parsing con centinaia di migliaia di caselle. Sebbene queste tabelle siano sparse e possano essere codificate in modo compatto, è assai utile chiedersi se sia possibile una via intermedia tra la semplicità dell'approccio SLR(1) e la selettività di quello canonico LR(1). Un buon compromesso tra espressività e compattezza è rappresentato dai parser LALR(1) (per *lookahead LR*), che sono alla base della maggior parte dei generatori automatici di analizzatori sintattici.

Il *nucleo* di uno stato dell'automa LR(1) è l'insieme degli item LR(0) che si ottengono dimenticando i simboli di lookahead dagli item LR(1). Non è difficile convincersi che il nucleo di uno stato dell'automa LR(1) è uno stato dell'automa canonico LR(0) (ovviamente in generale più stati LR(1) hanno lo stesso nucleo e danno quindi luogo allo stesso stato LR(0)). Inoltre, controllando come è definita la funzione  $G_{\text{to}}$ , si vede che le transizioni dell'automa LR(1) sono definite tenendo conto solo del nucleo degli stati, cioè non dipendono dai simboli di lookahead. Detto in altro modo, se prendiamo due stati  $s$  e  $t$  dell'automa LR(1) con lo stesso nucleo, se c'è una transizione da  $s$  a  $s'$  etichettata  $X$ , c'è senz'altro anche una transizione da  $t$  a qualche stato  $t'$  con la stessa etichetta  $X$ . Non solo, ma  $s'$  e  $t'$  hanno certamente lo stesso nucleo, ancora una volta perché le transizioni sono definite solo con l'informazione dei nuclei. L'*automa LALR(1)* per una grammatica  $G$  si ottiene prendendo l'automa canonico LR(1) per  $G$ , identificando e facendo l'unione insiemistica degli stati con lo stesso nucleo, e definendo quindi le transizioni nel modo naturale che risulta dalla fusione degli stati. L'automa LALR(1) è sempre isomorfo all'automa LR(0) per la stessa grammatica (ma gli stati sono composti da item LR(1)). Possiamo ora applicare le regole per la costruzione della tabella di parsing LR(1) (a pag. 120) all'automa LALR(1), ottenendo una *tabella di parsing LALR(1)*. Una grammatica è LALR(1) se nella tabella così ottenuta non vi sono conflitti. Si osservi che la dimensione di una tabella LALR(1) coincide con quella tabella SLR(0) per la stessa grammatica (cioè, per linguaggi reali è di un ordine di grandezza più piccola della tabella LR(1)).

Riferendoci alla grammatica dell'Esempio 4.36, il cui automa LR(1) è riportato in Figura 4.15, gli stati che condividono lo stesso nucleo sono: 6 e 13, 7 e 12, 8 e 11, 10 e 14. L'automa LALR(1) risultante dalla loro unione è in Figura 4.16. La tabella LALR(1) si ottiene da quest'automa, o, più direttamente, dalla tabella di parsing LR(1) e fondendo le righe degli stati con lo stesso nucleo.

	$x$	*	=	\$	$S$	$E$	$V$
1	s8	s6			g2	g5	g3
2					a		
3			s4	r3			
4	s8	s6			g9	g7	
5					r2		
6=13	s8	s6			g10	g7	
7=12			r3	r3			
8=11			r4	r4			
9					r1		
10=14			r5	r5			

Abbiamo già osservato che l'automa LALR(1) è isomorfo all'automa LR(0), ma

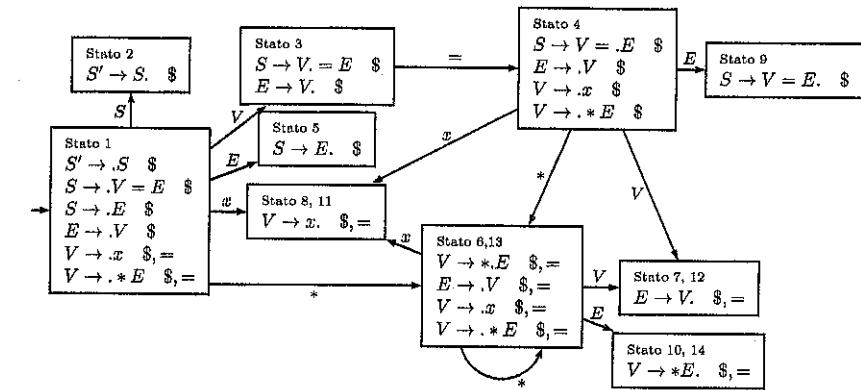


Figura 4.16 Automa LALR(1) per la grammatica dell'Esempio 4.36.

in virtù dei simboli di lookahead in generale cambiano le azioni della tabella: LALR(1) è più selettivo di SLR(1) nell'inserimento di azioni REDUCE. Nel caso di esempio la tabella SLR(1) è la seguente:

	$x$	*	=	\$	$S$	$E$	$V$
1	s8	s6			g2	g5	g3
2					a		
3			s4, r3	r3			
4	s8	s6			g9	g7	
5					r2		
6=13	s8	s6			g10	g7	
7=12			r3	r3			
8=11			r4	r4			
9					r1		
10=14			r5	r5			

dove in posizione [3, =] c'è un conflitto SHIFT/REDUCE. L'azione "r3" è dovuta alla presenza dell'item  $[E \rightarrow V]$  e al fatto che  $\text{Follow}(E) = \{=, \$\}$ . Si tratta dunque di una grammatica LR(1) e LALR(1), ma non SLR(1).

Talvolta il passaggio da una tabella LR(1) ad una LALR(1) mediante fusione di stati produce conflitti che non erano presenti nella tabella LR(1). In tal caso siamo in presenza di una grammatica che non è LALR(1).

**Esempio 4.37** La grammatica seguente è LR(1) ma non LALR(1):

$$\begin{array}{ll} S' \rightarrow S & S \rightarrow aAa \mid bAb \mid aBb \mid bBa \\ A \rightarrow c & B \rightarrow c \end{array}$$

Nell'automa canonico LR(1) per questa grammatica si trovano, tra gli altri, gli stati

$$s = \{[A \rightarrow c \cdot, a], [B \rightarrow c \cdot, b]\} \quad \text{e} \quad t = \{[A \rightarrow c \cdot, b], [B \rightarrow c \cdot, a]\}$$

che nella tabella LR(1) generano azioni REDUCE distinte in virtù del simbolo di lookahead. I due stati  $s$  e  $t$  condividono però lo stesso nucleo: quando vengono fusi in una tabella LALR(1) i simboli di lookahead non permettono più la distinzione e si generano due conflitti REDUCE/REDUCE (in corrispondenza sia di  $a$  sia di  $b$ ).

Nel passaggio da LR(1) a LALR(1), tuttavia, non si possono generare nuovi conflitti SHIFT/REDUCE che non fossero già presenti nella tabella LR(1). Supponiamo infatti, per assurdo, che da una tabella LR(1) senza conflitti, si ottenga una tabella LALR(1) con un conflitto SHIFT/REDUCE. Nel nuovo stato  $s$ , che si ottiene per fusione di  $s_1$  e  $s_2$ , vi è dunque un item  $[A \rightarrow \alpha \cdot, a]$  (che induce una riduzione su input  $a$ ) e un item  $[B \rightarrow \beta \cdot a \gamma, b]$  (che induce uno spostamento su  $a$ ). Siccome i due stati  $s_1$  e  $s_2$  condividono lo stesso nucleo, in uno dei due (poniamo  $s_1$ ) deve essere presente l'item  $[A \rightarrow \alpha \cdot, a]$  insieme ad un altro item  $[B \rightarrow \beta \cdot a \gamma, c]$ . Ma questo significa che già  $s_1$  ha un conflitto SHIFT/REDUCE sull'input  $a$  nella tabella LR(1), contrariamente a quanto avevamo supposto. Gli unici nuovi conflitti che si possono dunque presentare nella tabella LALR(1) sono di tipo REDUCE/REDUCE.

Il parser LALR(1) simula esattamente le stesse mosse del corrispondente parser LR(1) su tutte le stringhe che appartengono al linguaggio generato dalla grammatica. Sulle stringhe che *non* appartengono al linguaggio generato, invece, il parser LALR(1) può eseguire qualche ulteriore passo di REDUCE rispetto al suo corrispondente LR(1). È possibile dimostrare che prima di effettuare un qualsiasi SHIFT anche il parser LALR(1) entrerà in una configurazione di errore. I due parser, quindi, consumano esattamente la stessa porzione di input prima di rilevare che una certa stringa non appartiene al linguaggio generato dalla grammatica.

Nel nostro percorso abbiamo introdotto l'automa e il parser LALR(1) attraverso l'automa canonico (o la tabella) LR(0). D'altra parte, avevamo motivato l'introduzione del parser LALR(1) con la necessità di ridurre la dimensione delle strutture dati necessarie alla costruzione e al funzionamento del parser. Se per costruire un parser LALR(1) dovessimo necessariamente passare per una tabella LR(1), questo vantaggio sarebbe in gran parte vanificato. C'è un altro modo per ottenere un automa LALR(1): partire dall'automa canonico LR(0) e *aggiungere* opportunamente simboli di lookahead agli stati. Questo metodo permette di costruire gli stati LALR(1) in modo efficiente e compatto, senza dover generare esplicitamente tutta la tabella LR(1). I generatori automatici di analizzatori sintattici usano tecniche di questo tipo, e possono dunque essere usati anche su processori di limitate capacità di memorizzazione o di calcolo. Si vedano i testi in bibliografia per una descrizione di queste tecniche.

In conclusione, la Figura 4.17 riassume le modalità di riempimento della tabella di parsing per i diversi metodi che abbiamo trattato. Solo le azioni REDUCE sono inserite in modo diverso, via via più selettivamente passando a parser più potenti. Nel caso della tabella LALR(1), le regole sono le stesse del parser LR(1) se si passa attraverso la costruzione dell'automa LALR(1) mediante unione degli stati LR(1); altrimenti, si prende direttamente la fusione delle righe della tabella LR(1).

	LR(0)	SLR(1)	LR(1)
shift $t$	$\leftarrow$	$M[s, x], \text{ se } s \xrightarrow{\alpha} t, \text{ per } x \in T$	$\longrightarrow$
accept	$\leftarrow$	$M[s, \$], \text{ se } S' \xrightarrow{\cdot} S \cdot \in s$	$\longrightarrow$
reduce $A \rightarrow \alpha$	$M[s, x], \text{ se } A \rightarrow \alpha \cdot \in s$ per ogni $x \in T \cup \{\$\}$	$M[s, x], \text{ se } A \rightarrow \alpha \cdot \in s$ per ogni $x \in \text{FOLLOW}(A)$	$M[s, x],$ se $[A \rightarrow \alpha \cdot, x] \in s$
goto $t$	$\leftarrow$	$M[s, A], \text{ se } s \xrightarrow{A} t, \text{ per } A \in NT$	$\longrightarrow$

Figura 4.17 Riepilogo: come riempire una tabella di parsing  $M$ .

#### 4.8.7 Parsing con grammatiche ambigue

Abbiamo già osservato che le grammatiche ambigue non permettono la costruzione di parser top-down predittivi; lo stesso accade per i parser bottom-up deterministici:

**Teorema 4.38** Una grammatica ambigua non è LR( $k$ ), per nessun  $k$ .

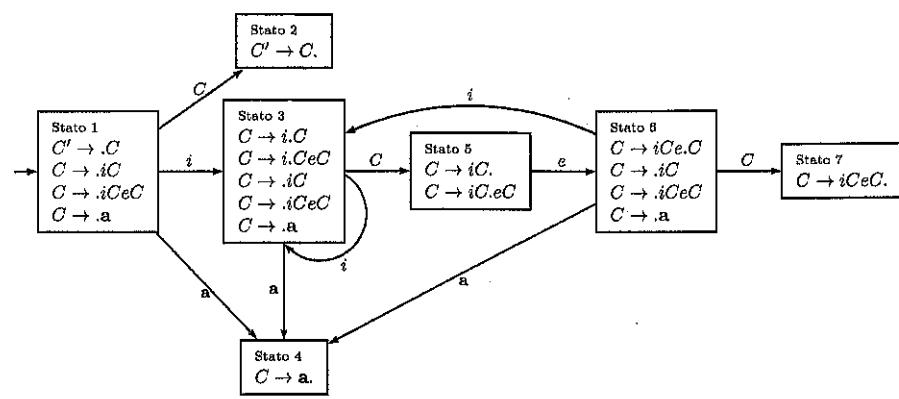
Sebbene esistano linguaggi inerentemente ambigi (cioè che sono generati solo da grammatiche ambigue), si tratta di casi patologici: i costrutti dei linguaggi di programmazione ammettono tutti delle presentazioni non ambigue. Abbiamo però visto nel Paragrafo 2.2.1 che la rimozione dell'ambiguità viene pagata in termini di complessità della grammatica: devono essere aggiunti nuovi non terminali, per "stratificare" le derivazioni e renderle uniche. Talvolta si può preferire una descrizione grammaticale ambigua, insieme ad una nota in linguaggio naturale che specifichi come risolvere l'ambiguità. La seguente grammatica esemplifica il problema collegato con l'*if-then-else* che abbiamo già discusso nel Paragrafo 2.2.1 (i non terminali sono  $i$ ,  $e$  e  $a$ ):

$$C \rightarrow iC \mid iCeC \mid a$$

La stringa  $i \ i \ a \ e \ a$  ha due alberi di derivazione, uno che accoppia la unica  $i$  alla  $i$  esterna, l'altro che l'accoppia alla seconda  $i$ . La prassi canonica dei linguaggi di programmazione è quella di prescrivere che lo "else (e)" è sempre accoppiato con lo *if* ( $i$ ) a lui più vicino" (nel nostro caso, la stringa deve essere letta come  $i$  ( $i \ a \ e \ a$ ), inserendo delle parentesi per comodità di comprensione).

La presentazione ambigua è più agile e intuitiva di quella non ambigua (si confronti con il frammento di grammatica della Figura 2.7). Se costruiamo una tabella di parsing LR a partire dalla grammatica ambigua otteniamo (ovviamente) un conflitto; vediamo il caso SLR(1).<sup>12</sup> La Figura 4.18 mostra l'automa canonico

<sup>12</sup>Il teorema appena enunciato assicura che il problema non è risolto da parser più potenti.



**Figura 4.18** Automa LR(0) per la grammatica ambigua dello if-then-else.

LR(0) per la nostra grammatica. Lo stato 5 provoca un conflitto SHIFT/REDUCE nell'elemento  $M[5, e]$  della tabella di parsing SLR(1): la transizione etichettata  $e$  dallo stato 5 allo stato 6 inserisce SHIFT 6, mentre l'item  $C \rightarrow iC.$  inserisce REDUCE, perché  $\text{FOLLOW}(C) = \{\$\}, e\}$ . Se vogliamo garantire che la  $e$  sia accoppiata con la  $i$  a lei più vicina, possiamo risolvere manualmente il conflitto: se nello stato 5 l'input presenta una  $e$ , dobbiamo *spostare*  $e$ . In questo modo a partire dalla nostra grammatica ambigua abbiamo ottenuto un parser SLR(1) che garantisce il vincolo di accoppiamento espresso in linguaggio naturale.

Un altro esempio istruttivo dello stesso tipo è quello della grammatica delle espressioni aritmetiche

$$E \rightarrow E + E \mid E * E \mid (E) \mid a$$

Si tratta di una grammatica ambigua in due modi diversi: non distingue la precedenza tra  $*$  e  $+$  (per esempio permette due alberi distinti per  $a + a * a$ ) e inoltre non risolve l'associatività dei due operatori (per esempio ci sono due alberi distinti per  $a + a + a$ ). Gli stati dell'automa LR(0) sono riportati in Figura 4.19: si vede immediatamente che l'ambiguità si rivela negli stati 8 e 9. Siccome  $\text{FOLLOW}(E') = \{\$\}$  e  $\text{FOLLOW}(E) = \{+, *, (), \$\}$ , la tabella di parsing SLR(1) è la seguente:<sup>13</sup>

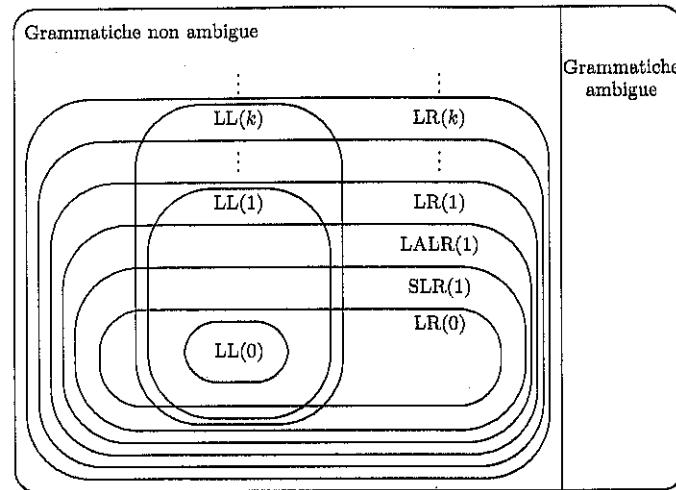
- |    |                        |   |                        |
|----|------------------------|---|------------------------|
| 1  | $E' \rightarrow .E$    | 2 | $E' \rightarrow E.$    |
|    | $E \rightarrow .E + E$ |   | $E \rightarrow E. + E$ |
|    | $E \rightarrow .E * E$ |   | $E \rightarrow E.* E$  |
|    | $E \rightarrow .(E)$   |   |                        |
|    | $E \rightarrow .a$     | 4 | $E \rightarrow a.$     |
| 3  | $E \rightarrow (E)$    | 5 | $E \rightarrow E + .E$ |
|    | $E \rightarrow .E + E$ |   | $E \rightarrow .E + E$ |
|    | $E \rightarrow .E * E$ |   | $E \rightarrow .E * E$ |
|    | $E \rightarrow .(E)$   |   | $E \rightarrow .(E)$   |
|    | $E \rightarrow .a$     | 6 | $E \rightarrow .a$     |
| 7  | $E \rightarrow (E.)$   | 8 | $E \rightarrow E + E.$ |
|    | $E \rightarrow E. + E$ |   | $E \rightarrow E. + E$ |
|    | $E \rightarrow E.* E$  |   | $E \rightarrow E.* E$  |
|    | $E \rightarrow .(E)$   | 9 | $E \rightarrow E * E.$ |
|    | $E \rightarrow .a$     |   | $E \rightarrow E. + E$ |
| 10 | $E \rightarrow (E).$   |   | $E \rightarrow E.* E$  |

**Figura 4.19** Stati LR(0) per la grammatica ambigua delle espressioni.

	+	*	(	)	a	\$		$E$
1				s3		s4		g2
2	s5	s6				a		
3			s3		s4			g7
4	r4	r4		r4		r4		
5			s3		s4			g8
6			s3		s4			g9
7	s5	s6			s10			
8	r1,s5	r1,s6		r1		r1		
9	r2,s5	r2,s6		r2		r2		
10	r3	r3		r3		r3		

I quattro conflitti corrispondono esattamente alle quattro situazioni di ambiguità: le due precedenze e le due associatività. Come dobbiamo risolvere l'ambiguità se assumiamo, al solito, che  $*$  abbia precedenza su  $+$  e che entrambi gli operatori associno a sinistra? Nello stato 8, abbiamo sulla cima della pila  $E + E$ : se il simbolo di lookahead è  $+$ , dobbiamo esplicitare l'associatività a sinistra, ovvero *ridurre*:  $M[8, +] = r1$ ; se invece il simbolo di lookahead è  $*$ , dobbiamo forzare la precedenza di  $*$ , cioè *spostare*:  $M[8, *] = s6$ . Analogamente, se siamo nello stato 9, sulla pila è presente  $E * E$ : sia l'associatività a sinistra di  $*$  sia la minor precedenza di  $+$  su  $*$  si ottiene se si ottengono forzando una riduzione:  $M[9, *] = M[9, +] = r2$ . Il parser così ottenuto è più piccolo (meno stati) di quello che

<sup>13</sup>A prima vista c'è anche un conflitto in posizione  $M[2, \$]$ , perché vi sono due item che richiedono una SHIFT, mentre c'è anche lo item  $E' \rightarrow E.$ , che è completo. Ma siccome si tratta dell'item speciale della grammatica aumentata, l'ambiguità è immediatamente risolta in favore di ACCEPT.



**Figura 4.20** Relazioni tra grammatiche  $LL(k)$  e  $LR(k)$ .

corrisponde alla grammatica non ambigua ed è anche più efficiente: il parser della grammatica non ambigua passa molto del suo tempo a mettere e togliere dalla pila i simboli delle produzioni unitarie (come  $E \rightarrow T$  o  $T \rightarrow A$ ).

## 4.9 Linguaggi e grammatiche deterministici

Ci possiamo chiedere a questo punto quali siano le relazioni tra le diverse classi di grammatiche. Abbiamo già osservato che nessuna grammatica ambigua è mai  $LL(k)$  o  $LR(k)$ . È possibile dimostrare che ogni grammatica  $LL(k)$  è  $LR(k)$  (con inclusione propria) e che le  $LL(1)$  hanno intersezione non vuota con le  $SLR(1)$  e le  $LALR(1)$ , ma senza inclusione in nessuno dei due versi.<sup>14</sup> Inoltre, l'inclusione tra  $LL(k)$  e  $LL(k+1)$  e quella tra  $LR(k)$  e  $LR(k+1)$  è propria. Queste relazioni sono tutte riassunte nella Figura 4.20.

Diverso è il discorso se guardiamo ai *linguaggi*. Ogni grammatica  $LR(k)$  genera un linguaggio deterministico, ma non vi è gerarchia di linguaggi al variare di  $k$ :

**Teorema 4.39** (i) Un linguaggio libero da contesto è deterministico sse è generato da una grammatica  $LR(k)$ , per qualche  $k$ .

<sup>14</sup>Le grammatiche  $LL(1)$  sono "quasi" contenute nelle  $LALR(1)$ : solo alcuni esempi patologici sono  $LL(1)$  e non  $LALR(1)$ ; esistono invece naturali grammatiche  $LALR(1)$  che non sono  $LL(1)$ .

(ii) Un linguaggio è generato da una grammatica  $LR(k)$ , per qualche  $k$ , sse è generato da una grammatica  $SLR(1)$ .

(ii) I linguaggi generati dalle grammatiche  $LL(k)$  sono strettamente contenuti nei linguaggi generati dalle grammatiche  $SLR(1)$ .

Da un certo punto di vista, dunque, tutti i linguaggi di nostro interesse sono descrivibili con una grammatica  $SLR(1)$ . Il punto è che questa grammatica può non essere la grammatica più semplice, o intuitiva, per quel linguaggio. Ecco perché gli strumenti automatici trattano grammatiche  $LALR(1)$ , che cercano di mediare tra grammatiche ragionevolmente semplici e tabelle di parsing relativamente compatte.

## 4.10 Generatori di analizzatori sintattici

Le tecniche per la costruzione di parser LR che abbiamo discusso in questo capitolo sono automatizzate da molti strumenti, di cui il capostipite è YACC (*Yet Another Compiler Compiler*), che discuteremo brevemente in questo paragrafo.<sup>15</sup> Yacc prende in input la descrizione di una grammatica e restituisce un programma C che è un parser  $LALR(1)$  per quella grammatica, interfacciandosi con Lex per la parte lessicale (Paragrafo 3.7) e permettendo la risoluzione manuale di eventuali ambiguità.

Analogamente a Lex, un sorgente Yacc (cioè l'input di Yacc) ha la seguente struttura generale:

```
%{ prologo %}
definizioni
%%
regole
%%
funzioni ausiliarie
```

Sia il prologo (con le sue parentesi `%{` e `%}`) sia le definizioni sono opzionali. Se presente, il prologo contiene la definizione di macro e altre dichiarazioni di variabili o funzioni che saranno usate nella sezione delle regole. Tutto il prologo è copiato da Yacc nel suo output, in modo da precedere la definizione della funzione che effettivamente eseguirà l'analisi sintattica (che si chiama `yyparse`). È comune usare qui delle `#include`.

La parte delle definizioni contiene la dichiarazione dei simboli che saranno usati nella descrizione della grammatica; in particolare sono qui elencati tutti i nomi dei token, che sono resi disponibili anche allo scanner generato da Lex, se questo è usato insieme a Yacc. In questa sezione è possibile anche dichiarare la precedenza e l'associatività di alcuni terminali.

<sup>15</sup>Yacc esiste oggi in tante versioni distinte, tutte basate sullo stesso approccio e tra loro largamente compatibili; una delle più comuni è GNU Bison.

La parte centrale di un sorgente Yacc è data dalle regole, che costituiscono la definizione della grammatica. Una regola è una coppia (*produzione, azione semantica*), nella quale l'azione semantica è il codice che il parser prodotto da Yacc deve eseguire tutte le volte che esegue un'azione REDUCE secondo la produzione associata. Una produzione della forma

$$\text{nonterm} \rightarrow \text{corpo}_1 | \dots | \text{corpo}_k$$

viene espressa in Yacc come

```
nonterm : corpo1 {azione semantica1}
        ...
        | corpok {azione semanticak}
;
```

Una stringa alfanumerica non dichiarata precedentemente come token è considerata un non terminale, mentre un carattere tra apici singoli è un terminale. Il simbolo iniziale della grammatica è il non terminale usato come testa nella prima regola. Un'azione semantica è codice C eseguito al momento in cui il parser riduce mediante quella produzione: in genere questo codice calcola il "valore semantico" della testa, in funzione dei valori semanticici dei simboli che compongono il corpo. Ad esempio, il valore semantico potrebbe essere l'albero di derivazione, nel caso in cui si stia producendo un compilatore con alberi esplicativi; oppure potrebbe essere il codice oggetto connesso alla produzione; oppure, nel caso in cui si stia producendo un interprete, la vera e propria valutazione dell'espressione (faremo un esempio di questo tra breve). In un'azione semantica, il simbolo speciale `$$` si riferisce al valore semantico della testa, mentre `$i` si riferisce al valore semantico dell'*i*-esimo non terminale del corpo della produzione.

La parte delle funzioni ausiliarie contiene tutte le funzioni di supporto per la generazione del parser. Deve essere presente una funzione di nome `yylex()` che fornisca un analizzatore lessicale; la scelta più semplice è chiaramente quella di produrlo con Lex. La funzione `yylex()` restituisce il nome del token, che deve essere stato definito nella prima sezione del file Yacc; il valore del token è condiviso nella variabile intera `yylval`.

La Figura 4.21<sup>16</sup> riporta il classico esempio di un interprete per le espressioni aritmetiche. Il prologo dichiara che `YYSTYPE`, il tipo della pila interna di Yacc usata per i valori semanticici, è `double`. Nelle definizioni, troviamo innanzitutto la dichiarazione del token `NUM` (che sarà restituito da `yylex`). Le linee seguenti introducono altri token (+, -, \*, /, NEG) insieme a specifiche annotazioni per essi: token costituiti da un singolo carattere non devono essere dichiarati, ma qui sono introdotti per specificare le regole di associatività e precedenza. Tutti gli operatori binari associano a sinistra; inoltre viene dichiarata implicitamente anche la loro precedenza: token sulla stessa linea hanno la stessa precedenza tra loro; un token ha precedenza su tutti i token delle linee che precedono (dunque `NEG` ha la massima precedenza, poi seguono \* e /, ecc.).

```
%{ /* PROLOGO */
#define YYSTYPE double
#include <math.h>
#include <stdio.h>
}

/* DEFINIZIONI */
%token NUM

%left '-' '+'
%left '*' '/'
%left NEG      /* meno unario */

%% /* REGOLE E AZIONI SEMANTICHE */
input: /* empty */
      | input line
;

line:   '\n'
      | exp '\n' { printf ("\t%.10g\n", $1); }
;

exp:   NUM          { $$ = $1; }
      | exp '+' exp { $$ = $1 + $3; }
      | exp '-' exp { $$ = $1 - $3; }
      | exp '*' exp { $$ = $1 * $3; }
      | exp '/' exp { $$ = $1 / $3; }
      | '-' exp %prec NEG { $$ = -$2; }
      | '(' exp ')' { $$ = $2; }
;

%% /* FUNZIONI AUSILIARIE */
#include "lex.yy.c"
```

**Figura 4.21** Un programma Yacc per le espressioni aritmetiche.

Le regole introducono il simbolo iniziale `input`: una stringa vuota (la prima alternativa), oppure una sequenza finita di `line` (espressa dalla ricorsione sinistra della seconda alternativa). Una `line` è un carattere di ritorno carrello `\n`, oppure un'espressione terminata dal ritorno carrello: l'azione semantica associata a questa seconda alternativa è quella di stampare `$1`, cioè il valore semantico del primo non terminale del corpo, ovvero di `exp`. In questo modo il parser si comporta come un interprete: riconosce la presenza su una linea di un'espressione fatta di `NUM` e ne stampa il valore.

Le regole per `exp` sono la parte principale: si riconosce subito l'usuale grammatica ambigua per le espressioni. L'ambiguità è risolta forzando opportunamente le azioni, secondo quanto specificato nella parte definizione. Regole ed azioni

<sup>16</sup>Adattata dalla documentazione di Bison; © Free Software Foundation, Inc.

semantiche sono autoesplicative, eccetto l'annotazione `%prec NEG` nella quinta produzione: `%prec` specifica che alla produzione corrente deve essere applicata la stessa precedenza di `NEG`. In altri termini, specifica che `-`, quando è unario, è un'istanza del token `NEG` ed ha dunque la precedenza più alta.

#### 4.10.1 Yacc e l'ambiguità

Se Yacc incontra conflitti nella generazione della tabella LALR(1), forza comunque una risoluzione, comunicando il numero dei conflitti incontrati; invocandolo con l'opzione `-v` si ottiene una lista dei conflitti e come sono stati risolti. Le regole standard utilizzate per la risoluzione dei conflitti sono:

1. ogni conflitto SHIFT/REDUCE è risolto in favore di SHIFT;
2. un conflitto REDUCE/REDUCE viene risolto scegliendo la produzione elencata per prima nelle regole.

La prima regola può essere controllata (imponendo che alcuni conflitti siano risolti a favore di REDUCE), mediante le direttive specifiche introdotte nel file Yacc (come `left`, `right`, `nonassoc`, ecc.). Abbiamo già osservato che l'ordine delle definizioni corrisponde all'ordine (crescente) della precedenza dei token. Yacc assegna poi una precedenza (e un'associatività) anche a ciascuna *regola* (produzione): ad ogni produzione è associata la precedenza del suo terminale più a destra, a meno che tale precedenza non sia modificata con la direttiva `%prec`, che istruisce Yacc ad usare invece una precedenza specifica. Nel caso dell'esempio, la precedenza della sesta produzione (meno unario) sarebbe stata quella di `-` (ultimo terminale a destra), ma la direttiva `%prec NEG` assegna a questa produzione la precedenza e l'associatività di `NEG`. Quando si deve risolvere un conflitto SHIFT  $a$ /REDUCE  $n$ , Yacc confronta la precedenza e l'associatività di  $a$  con quelle della produzione  $n$ : riduce se la precedenza di  $n$  è maggiore di quella di  $a$ , oppure se hanno la stessa precedenza e l'associatività di  $n$  è sinistra; in tutti gli altri casi si attiene alla regola standard, spostando.

### 4.11 Sommario del capitolo

In questo capitolo abbiamo trattato i principali argomenti teorici e implementativi collegati alla costruzione di un analizzatore sintattico.

- Gli *automi a pila* (PDA), discutendone il ruolo come riconoscitori di linguaggi liberi.
- Il *pumping lemma* per i linguaggi libri, un'importante proprietà goduta da ogni linguaggio libero da contesto, che può essere utilizzata per dimostrare che alcuni linguaggi *non* sono libri.
- Abbiamo sinteticamente introdotto la *gerarchia di Chomsky*, una classificazione dei linguaggi che parte dai semplici linguaggi regolari per arrivare ai linguaggi semidecidibili.

- I *linguaggi libri deterministici*, che costituiscono una classe strettamente più piccola di quella di tutti i linguaggi libri (a differenza di quello che accade per i linguaggi regolari).
- Diverse tecniche di *manipolazione delle grammatiche*, per eliminare le produzioni inutili, quelle unitarie, quelle che hanno  $\epsilon$  nella parte destra, per eliminare la ricorsione sinistra.
- I parser top-down, interessandoci in particolare ai parser top-down predittivi, associati alle grammatiche LL( $k$ ).
- I più potenti parser bottom-up deterministici, interessandoci in particolare a quelli associati alle grammatiche LR( $k$ ).
- Abbiamo studiato in dettaglio la costruzione delle tabelle di parsing per le classi di grammatiche LR(0), SLR(0), LR(1) e LALR(1).
- *Yacc*: un generatore di analizzatori sintattici.

### 4.12 Nota bibliografica

La teoria dell'analisi sintattica costituisce una delle più belle applicazioni di un argomento teorico alla realizzazione industriale di programmi. Una trattazione più ampia di quella che abbiamo dato noi si può trovare nel classico [4], che presenta anche le fasi successive della compilazione. Un altro buon testo sulla costruzione di compilatori è [8], che ha versioni per C, Java e ML. Per una trattazione formalmente più completa si può vedere [91] o il più recente [31].

### 4.13 Esercizi

1. Si dimostri il pumping lemma per i linguaggi regolari usando l'equivalenza di questi con le grammatiche regolari (Definizione 3.22) e argomentando come per il PL per i libri.
2. Qual è il linguaggio generato dalla grammatica con terminali  $a, b$ , nonterminali  $A, B, C$ , simbolo iniziale  $A$  e produzioni seguenti:

$$\begin{aligned} A &\rightarrow aAa \mid B \\ B &\rightarrow B \mid \epsilon \\ C &\rightarrow b \end{aligned}$$

3. Si classifichi il linguaggio generato dalla grammatica dell'esercizio 2 (cioè si dica se è regolare, libero non regolare, non libero). Dimostrare quanto asserito.
4. Si verifichi, costruendo il corrispondente automa e tabella, se la seguente grammatica è SLR(1):

- (1)  $A \rightarrow Ba$
- (2)  $A \rightarrow ba$
- (3)  $A \rightarrow bBc$
- (4)  $B \rightarrow \epsilon$

5. Si consideri la seguente grammatica:

$$\begin{aligned} S &\rightarrow AaAb \mid BbBa \\ A &\rightarrow \epsilon \\ B &\rightarrow \epsilon \end{aligned}$$

- Si dica se si tratta o meno di una grammatica LL(1), motivando brevemente.
6. Dare le tabelle SLR(1) e LR(1) per la grammatica dell'esercizio 5. Vi sono conflitti in qualcuna delle due tabelle?
7. Si elimini la ricorsione sinistra dalla seguente grammatica

$$A ::= Aa \mid AbA \mid c \mid d$$

8. Si consideri la grammatica

- (1)  $S \rightarrow aAb$
- (2)  $S \rightarrow BbBa$
- (3)  $A \rightarrow \epsilon$
- (4)  $B \rightarrow \epsilon$

Si dica se si tratta di una grammatica LR(0), SLR(1), LR(1), LALR(1). Motivare opportunamente.

9. La grammatica dell'esercizio 8 è LL(1)? Motivare.
10. Data la seguente grammatica aumentata

$$\begin{array}{ll} (0) & S' \rightarrow S \\ (1) & S \rightarrow V = E \\ (2) & S \rightarrow E \end{array} \quad \begin{array}{ll} (3) & E \rightarrow V \\ (4) & V \rightarrow x \\ (5) & V \rightarrow *E \end{array}$$

- l'insieme di item LR(0)  $\{S \rightarrow \cdot V = E, V \rightarrow \cdot x\}$  può essere uno stato dell'automa canonico? Giustificare brevemente.
11. Si costruisca la tabella di parsing LALR(1) per la grammatica dell'esercizio 10; considerando poi l'elaborazione della stringa  $x = **x\$$ , si dia: (i) lo stato della pila del parser e (ii) il simbolo corrente, nel momento immediatamente successivo alla seconda  $r5$ . Si dia poi l'elenco completo delle riduzioni operate dal parser nell'elaborazione completa della stringa.

## Fondamenti

In questo capitolo tratteremo non tanto di linguaggi di programmazione, quanto delle limitazioni dei programmi che con essi si possono scrivere, chiedendoci se esistano dei problemi che non possono essere risolti da alcun programma. Una motivazione per questa indagine è la domanda che ci siamo posti al termine del Paragrafo 2.3, se cioè sia possibile costruire degli analizzatori di semantica statica che verifichino determinati vincoli imposti dalla definizione di un linguaggio. Scopriremo presto, tuttavia, che la risposta a questa domanda è assai più generale, tanto da riguardare una sorta di limite *assoluto* riguardo a cosa si possa (e non si possa) fare con un calcolatore. Sebbene la materia possa sembrare astratta, la trattazione che ne faremo sarà del tutto elementare.

### 5.1 Il problema della fermata

Nel Paragrafo 2.3 ci chiedevamo se esista un analizzatore di semantica statica in grado di rilevare se un programma può generare una divisione per zero durante una sua qualsiasi esecuzione. Invece di affrontare questo problema, considereremo un problema più basilare, nonché assai più interessante: ci chiederemo se esista un analizzatore statico capace di scoprire se un programma, su un certo dato di ingresso, possa andare in ciclo. Non c'è bisogno di sottolineare l'utilità di un controllo del genere: se sappiamo *prima dell'esecuzione* che un certo programma, su un fissato dato di ingresso, andrà in ciclo, abbiamo un modo per rilevare automaticamente un chiaro errore logico al suo interno.

Un analizzatore statico non è altro che un programma, usato come sottoprogramma all'interno di un compilatore. Fissiamo dunque un linguaggio di programmazione  $\mathcal{L}$  in cui scrivere i nostri programmi; dato un programma  $P$  ed un input  $x$ , indichiamo con  $P(x)$  il risultato della computazione di  $P$  su  $x$ . Osserviamo che, in generale, il programma  $P$  potrebbe non terminare su  $x$ , per effetto di un ciclo o di una ricorsione infinita; scrivendo  $P(x)$ , dunque, non indichiamo necessariamente una computazione terminata. Senza perdita di generalità, possiamo ora riformulare la nostra domanda come:

Esiste un programma  $H$  tale che, ricevuti in ingresso un programma  $P$  nel linguaggio  $\mathcal{L}$  ed un input  $x$ , termina stampando SI se  $P(x)$  termina, e termina invece stampando NO se  $P(x)$  va in ciclo?

Sottolineiamo come  $H$  ha due input, deve terminare sempre (non vogliamo un compilatore in ciclo!) e deve funzionare per ogni programma  $P$  in  $\mathcal{L}$  e ogni input  $x$ . Possiamo poi supporre, ancora senza perdere di generalità, che anche  $H$  sia scritto nel linguaggio  $\mathcal{L}$ , visto che  $\mathcal{L}$ , per essere interessante, deve essere un linguaggio nel quale si possano scrivere tutti i programmi possibili<sup>1</sup>.

Scopriamo ora le carte: vogliamo dimostrare che non esiste alcun programma  $H$  col comportamento appena discusso. Ragioneremo per assurdo, supponendo di avere a disposizione  $H$  e derivando da questo una contraddizione. L'argomentazione può sembrare a prima vista contorta, ma non richiede conoscenze avanzate. Il ragionamento, però, è sottile ed elegante: il lettore lo legga più volte e sia sicuro di comprendere il ruolo cruciale dell'autoapplicazione.

1. Supponiamo per assurdo di avere a disposizione un programma  $H$  con le proprietà su esposte.
2. Sfruttando  $H$ , possiamo scrivere un altro programma  $K$ , con un solo input (costituito da un programma), così fatto:

Il programma  $K$ , sull'input  $P$ , termina stampando SI, se  $H(P, P)$  stampa NO; va invece in ciclo se  $H(P, P)$  stampa SI.

Scrivere il programma  $K$ , avendo a disposizione  $H$ , è elementare: leggiamo l'input  $P$ ; chiamiamo  $H$  come sottoprogramma, passando  $P$  come primo e secondo parametro; aspettiamo che  $H$  termini (e ciò avverrà senz'altro, viste le specifiche di  $H$ ); se  $H(P, P)$  termina stampando NO (possiamo supporre di intercettare questa stampa e non farla comparire sull'output),  $K$  stampa SI; se invece  $H(P, P)$  stampa SI, allora  $K$  entra in un ciclo infinito programmato allo scopo.

Se ricordiamo le specifiche di  $H$ , possiamo riassumere la semantica di  $K$  come

$$K(P) = \begin{cases} \text{SI} & \text{se } H(P, P) \text{ non termina} \\ \text{non termina} & \text{se } H(P, P) \text{ termina.} \end{cases} \quad (5.1)$$

A prima vista può sembrare strana questa applicazione di  $P$  a se stesso. Ma ciò non ci deve meravigliare:  $P$  riceverà in input una stringa costituita dal testo di  $P^2$ .

3. Eseguiamo ora  $K$  sul suo stesso testo, cioè interessiamoci a  $K(K)$ . Qual è il suo comportamento? Se sostituiamo  $K$  al posto di  $P$  nella (5.1), otteniamo

$$K(K) = \begin{cases} \text{SI} & \text{se } K(K) \text{ non termina} \\ \text{non termina} & \text{se } K(K) \text{ termina.} \end{cases} \quad (5.2)$$

<sup>1</sup>Chiarire il senso di "tutti i programmi possibili" è proprio uno degli scopi di questo capitolo.

<sup>2</sup>Se ci eravamo immaginati che l'input di  $P$  dovesse essere un numero, basterà leggere come un numero la stringa del testo di  $P$ : un numero i cui bit sono costituiti dalla codifica dei singoli caratteri del testo di  $P$ .

4. Ma ora osserviamo come la (5.2) sia assurda! Dice che  $K(K)$  termina (stampando SI) quando  $K(K)$  non termina; e che non termina, quando  $K(K)$  termina.
5. Da cosa discende l'assurdo? Non da  $K$  in sé, che, come abbiamo visto, è un semplice programma che sfrutta  $H$ . L'assurdo discende dall'aver supposto che esistesse un programma con le caratteristiche di  $H$ . Dunque  $H$  non esiste.

Abbiamo dunque dimostrato che non esiste alcuna *procedura di decisione* (nel linguaggio  $\mathcal{L}$ ) capace di verificare se un altro, generico programma in  $\mathcal{L}$  termina su un generico input. Usiamo qui procedura di decisione nel senso tecnico di un programma che (i) funzioni per argomenti arbitrari; (ii) termini sempre e (iii) discrими (rispondendo si/no) gli argomenti che sono soluzione del problema da quelli che non lo sono.

Questo risultato, di fondamentale importanza per l'informatica, va sotto il nome di *indecidibilità del problema della fermata*. Molti altri interessanti problemi sono indecidibili allo stesso modo: ne discuteremo alcuni nel prossimo paragrafo, dopo aver prima discusso le caratteristiche di  $\mathcal{L}$  da cui discende il risultato; potremo così affrontare il problema della potenza espressiva dei linguaggi di programmazione.

## 5.2 Espressività dei linguaggi di programmazione

A prima vista, il risultato che abbiamo ottenuto nel paragrafo precedente può apparire abbastanza limitato: se prendiamo un linguaggio diverso da  $\mathcal{L}$ , forse il programma  $H$  esiste senza generare contraddizioni.

A ben vedere, tuttavia, non abbiamo assunto granché su  $\mathcal{L}$ . Abbiamo usato  $\mathcal{L}$  in modo implicito nella definizione del programma  $K$  a partire da  $H$  (cioè al punto 2 della dimostrazione). Per poter davvero scrivere  $K$  sono necessarie le seguenti condizioni:

1. in  $\mathcal{L}$  deve essere disponibile una forma di condizionale, per distinguere i casi nella definizione di  $K$ ;
2. in  $\mathcal{L}$  si devono poter definire funzioni che non terminano (dovremo dunque avere a disposizione qualche forma di iterazione o ricorsione).

A questo livello di dettaglio, in  $\mathcal{L}$  non serve molto di più. Quale linguaggio di programmazione non mette a disposizione questi costrutti? E se li mette a disposizione, lo si può usare al posto di  $\mathcal{L}$  nella dimostrazione, ottenendo che un programma come  $H$  non esiste in *nessun* linguaggio di programmazione degno di questo nome.

L'indecidibilità del problema della fermata, dunque, non è un fatto contingente, legato ad un particolare linguaggio di programmazione; né è l'espressione di una nostra incapacità come programmati. Si tratta, al contrario, di una limitazione in qualche modo assoluta, insindibilmente legata al concetto intuitivo di programma (algoritmo) e di linguaggio di programmazione. È un principio

di natura, simile in questo alla conservazione dell'energia o ai principi della termodinamica. “In natura” esistono più problemi e funzioni di quanti programmi possiamo scrivere, e tra i problemi ai quali non corrisponde nessun programma ve ne sono alcuni tutt'altro che insignificanti, come appunto il problema della fermata.

Anzi, come argomentato nel Paragrafo 5.3, i problemi per i quali esiste un programma che li risolve sono una “piccolissima parte” dell’insieme di tutti i possibili problemi.

Come abbiamo fatto per la fermata, affermare che un certo problema è indecidibile, significa che non esiste alcun programma che (i) funzioni per argomenti arbitrari; (ii) termini sempre e (iii) discrimin gli argomenti che sono soluzione del problema da quelli che non lo sono. Divagheremmo troppo se iniziassimo qui a dimostrare l’indecidibilità di altri problemi significativi; il lettore interessato può consultare ogni buon testo di teoria della calcolabilità. Possiamo però elenare alcuni problemi indecidibili, senza pretesa di spiegare tutti i termini in gioco o dimostrare alcunché. Sono indecidibili i seguenti problemi:

- verificare se un programma calcola una funzione costante;
- verificare se due programmi calcolano la stessa funzione;
- verificare se un programma termina per ogni input;
- verificare se un programma diverge per ogni input;
- verificare se un programma, dato un input, genererà un errore durante l’esecuzione;
- verificare se un programma causerà un errore di tipo (si veda il riquadro di pag. 286).

Come si vede, sono indecidibili problemi di grandissima importanza, anche applicativa. I risultati di indecidibilità ci dicono che non esistono strumenti software generali che stabiliscano in modo automatico proprietà significative di programmi.

### 5.2.1 Formalismi per la calcolabilità

Per rendere più preciso il discorso fatto su  $\mathcal{L}$ , dovremmo fissare un linguaggio di programmazione e mostrare che ad esso si applica il ragionamento del paragrafo precedente. Storicamente, il primo linguaggio nel quale è stata dimostrata l’impossibilità di scrivere il programma  $H$  è quello delle macchine di Turing, un formalismo a prima vista rudimentale introdotto negli anni ’30 dal matematico britannico Alan M. Turing e descritto sinteticamente nel riquadro con questo titolo. È a prima vista sorprendente che un formalismo così rudimentale (non c’è aritmetica predefinita, tutto si risolve nello spostamento di un numero finito di simboli su un nastro) sia in grado di esprimere computazioni abbastanza sofisticate come quelle necessarie a scrivere  $K$  a partire da  $H$ . Più sorprendente ancora è che esista una macchina di Turing che funge da interprete di tutte le altre, cioè una macchina che, presa in ingresso sul proprio nastro la descrizione (con un’opportuna codifica) di una generica macchina e un input per essa, esegue la computazione che quella macchina farebbe sull’input. Questo interprete è a tutti gli effetti un

### La macchina di Turing

Una macchina di Turing è composta da un nastro infinito, diviso in celle in ciascuna delle quali può essere memorizzato un unico simbolo appartenente ad un alfabeto finito. Sul nastro legge e scrive una testina mobile, che in ogni momento è posizionata su una cella. La macchina è gestita da un controllo con un numero finito di stati. Ad ogni passo di computazione, la macchina legge un simbolo dal nastro; a seconda dello stato della macchina e del simbolo letto, il controllo decide con quale simbolo sostituirlo sul nastro e se la testina si deve muovere a sinistra o a destra; il controllo passa quindi in uno degli altri stati (che sono, lo ricordiamo, in numero finito). Il controllo di una macchina di Turing può essere visto come il programma ad essa associato.

(semplicissimo) calcolatore come quelli che conosciamo oggi (programma e dati in memoria, ciclo fondamentale che interpreta le istruzioni del programma).

Una funzione è *calcolabile da un linguaggio*  $\mathcal{L}$  se esiste un programma in  $\mathcal{L}$  che la calcola. Più precisamente, la funzione (parziale, vedi il riquadro di pag. 13)  $f : A \rightarrow B$  è calcolabile se esiste un programma  $P$  con le seguenti caratteristiche: per ogni elemento  $a \in A$ , qualora  $P$  venga eseguito fornendo in input (una codifica di)  $a$ , si ha che  $P(a)$  termina fornendo in output (una codifica di)  $f(a)$ , se  $f(a)$  è definita; la computazione  $P(a)$  invece non termina se  $f$  non è definita su  $a$ .

Ci aspetteremmo forse che le funzioni calcolate dalle macchine di Turing siano meno di quelle calcolate da un sofisticato linguaggio di programmazione dei nostri giorni. Per indagare questa questione, sin dagli anni ’30 sono stati proposti e studiati innumerevoli formalismi per esprimere algoritmi (programmi), tra i quali possiamo ricordare il formalismo delle funzioni ricorsive generali di Church-Gödel-Kleene (che non fa riferimento ad alcun linguaggio di programmazione), il lambda-calcolo (che vedremo sinteticamente nel Capitolo 13), e poi tutti i linguaggi di programmazione esistenti. Ebbene, tutti questi formalismi e linguaggi possono essere simulati l’uno nell’altro, nel senso che in ciascuno di essi si può scrivere un interprete per un altro formalismo qualsiasi. Ne segue che essi sono tutti equivalenti quanto alle funzioni calcolate: tutti calcolano esattamente le funzioni calcolate dalle macchine di Turing. In principio, dunque, ogni algoritmo è esprimibile in un qualsiasi linguaggio di programmazione. Talvolta ci si riferisce a questa situazione dicendo che *tutti i linguaggi di programmazione sono Turing completi* (o Turing equivalenti). I risultati di indecidibilità si possono esprimere anche dicendo che esistono delle funzioni non calcolabili (con una macchina di Turing, ovvero, per quanto appena detto, in nessun linguaggio di programmazione).

Se tutti i linguaggi sono equivalenti quanto a funzioni calcolabili, è evidente che essi non sono equivalenti quanto a flessibilità d’uso, a pragmatica, a principi d’astrazione, e così via. Spesso ci si riferisce al complesso di queste caratteristiche come all’*espressività* di un certo linguaggio. Mentre dunque i formalismi per la calcolabilità forniscono un risultato definitivo di equivalenza relativamente

### La tesi di Church

Le dimostrazioni di equivalenza tra i vari linguaggi di programmazione (e tra i vari formalismi per la calcolabilità) sono veri e propri teoremi: dati i linguaggi  $\mathcal{L}$  e  $\mathcal{L}'$ , si scrive in  $\mathcal{L}$  l'interprete per  $\mathcal{L}'$ , e poi in  $\mathcal{L}'$  l'interprete per  $\mathcal{L}$ . A questo punto  $\mathcal{L}$  e  $\mathcal{L}'$  sono equivalenti. Una dimostrazione di questo tipo è stata effettivamente fatta per tutti i linguaggi esistenti, che sono dunque dimostrabilmente equivalenti. Questo argomento lascerebbe in realtà aperta la porta alla possibilità che prima o poi qualcuno sia in grado di trovare una funzione “intuitivamente calcolabile” la quale non abbia programmi in nessun linguaggio di programmazione esistente. Tutti i risultati di equivalenza dimostrati in più di settant'anni, tuttavia, costituiscono imponente evidenza che questo sia impossibile. Alla metà degli anni '30, Alonzo Church propose un principio, che da allora viene detto tesi di Church (o di Church-Turing), che afferma proprio questa impossibilità. Possiamo formulare la tesi di Church come: ogni funzione intuitivamente calcolabile è calcolata da una macchina di Turing.

A differenza dei risultati di equivalenza, la tesi di Church non è un teorema, perché fa riferimento ad un concetto (quello di intuitivamente calcolabile) che non è suscettibile di ragionamento formale. Si tratta di un principio di filosofia naturale, che la comunità degli informatici ritiene vero in modo assai forte e che non è stato scalfito neppure da nuovi paradigmi di computazione, come per esempio la computazione quantistica.

alle funzioni esprimibili, questi stessi formalismi non sono di nessun aiuto per discutere dell'espressività dei linguaggi. Anzi, a tutt'oggi non c'è accordo su come confrontare in modo formale questi aspetti.

### 5.3 Esistono più funzioni che algoritmi

Nel Paragrafo 5.1 abbiamo dato uno specifico esempio di funzione non calcolabile. Possiamo dare un'altra dimostrazione che esistono delle funzioni non calcolabili (pur senza riuscire ad “esibire” un esempio specifico come quello del Paragrafo 5.1) usando un semplice argomento di cardinalità, cioè mostrando che “in natura” esistono più funzioni che algoritmi.

Consideriamo innanzitutto un qualsiasi sistema formale che ci permetta di esprimere gli algoritmi e che, per semplicità, assumiamo essere un linguaggio di programmazione  $\mathcal{L}$  (il discorso potrebbe comunque essere generalizzato a definizioni più ampie). È abbastanza semplice vedere che l'insieme di tutti i possibili programmi (finiti) che possiamo scrivere in  $\mathcal{L}$  è numerabile, ossia può essere messo in corrispondenza biunivoca con i numeri naturali (che indichiamo con  $\mathbb{N}$ ): possiamo infatti considerare prima l'insieme  $P_1$  contenente tutti i programmi di lunghezza 1 (ossia che contengono un solo carattere), poi l'insieme  $P_2$  contenente tutti i programmi di lunghezza due e così via. Ogni insieme  $P_i$  è finito e può essere ordinato, ad esempio lessicograficamente (prima tutti i programmi che iniziano

per  $a$ , poi quelli che iniziano per  $b$  ecc., considerando anche i caratteri successivi). È evidente che così facendo, considerando l'ordine dato dal pedice negli insiemi  $P_1, P_2, \dots$  e quello interno ad ogni insieme, possiamo “contare” (o enumerare) tutti i possibili programmi e quindi metterli in corrispondenza biunivoca con i numeri naturali. Questo fatto, in termini più formali, si esprime dicendo che la cardinalità dell'insieme di tutti i programmi scrivibili in  $\mathcal{L}$  è eguale alla cardinalità dei naturali. Consideriamo adesso l'insieme  $\mathcal{F}$  contenente tutte le funzioni  $\mathbb{N} \rightarrow \{0, 1\}$ . Un importante teorema dovuto a Cantor ci dice che tale insieme non è numerabile, ossia ha una cardinalità strettamente maggiore di quella di  $\mathbb{N}$  per cui, dato che ogni programma può esprimere un'unica funzione, non riusciremo mai ad esprimere tutte le funzioni in  $\mathcal{F}$  con programmi di  $\mathcal{L}$ .

Vediamo una dimostrazione diretta del fatto che  $\mathcal{F}$  non è numerabile. Supponiamo, per assurdo, che  $\mathcal{F}$  sia numerabile, cioè che si possa scrivere  $\mathcal{F} = \{f_j\}_{j \in \mathbb{N}}$ . Osserviamo innanzitutto che possiamo mettere in corrispondenza biunivoca  $\mathcal{F}$  con l'insieme  $\mathcal{B}$  di tutte le successioni infinite di cifre binarie: ad  $f_j \in \mathcal{F}$  corrisponde la successione  $b_{j,1}, b_{j,2}, b_{j,3}, \dots$ , dove  $b_{j,i} = f_j(i)$ , per  $i, j \in \mathbb{N}$ . Dunque se  $\mathcal{F}$  è numerabile, lo è anche l'insieme  $\mathcal{B}$ . Essendo  $\mathcal{B}$  numerabile, possiamo enumerare i suoi elementi uno dopo l'altro, elencando per ogni elemento (per ogni sequenza) le cifre binarie che lo compongono. Possiamo disporre tale enumerazione in una matrice quadrata infinita

$$\begin{matrix} b_{1,1}, b_{1,2}, b_{1,3}, \dots \\ b_{2,1}, b_{2,2}, b_{2,3}, \dots \\ b_{3,1}, b_{3,2}, b_{3,3}, \dots \\ \vdots \end{matrix}$$

dove la riga  $j$  contiene la sequenza relativa alla funzione  $j$ -esima. Indicando con  $\bar{b}$  il complemento della cifra binaria  $b$ , consideriamo ora la sequenza di cifre binarie  $\bar{b}_{1,1}, \bar{b}_{2,2}, \bar{b}_{3,3}, \dots$ . Questa successione (essendo una successione infinita di cifre binarie), è certo un elemento di  $\mathcal{B}$ , tuttavia *non compare* nella nostra matrice (e quindi nella nostra enumerazione), visto che si differenzia in almeno un punto da ogni riga: sulla diagonale, la successione presente nella matrice ha l'elemento  $b_{j,j}$ , mentre, per costruzione, la nostra nuova successione nella posizione  $j$  ha l'elemento  $\bar{b}_{j,j}$ .

Abbiamo dunque un assurdo: avevamo supposto che  $\mathcal{F} = \{f_j\}_{j \in \mathbb{N}}$  fosse un'enumerazione di *tutte* le funzioni (cioè di tutte le successioni), mentre abbiamo costruito una funzione (una successione) che non appartiene all'enumerazione. Dunque la cardinalità di  $\mathcal{F}$  è strettamente maggiore di quella di  $\mathbb{N}$ .

Si può dimostrare che  $\mathcal{F}$  ha la cardinalità dei numeri reali, il che indica che l'insieme dei programmi (che è numerabile) è “molto più piccolo” di quello delle possibili funzioni, e quindi dei possibili problemi.

## 5.4 Sommario del capitolo

I fenomeni di calcolo su cui si basa l'informatica hanno le loro radici nella teoria della calcolabilità, che studia i formalismi nei quali esprimere algoritmi e le loro limitazioni. Il capitolo ha fornito solo il risultato principale di questa teoria, ma si tratta di un fatto di grandissima importanza, che ogni informatico dovrebbe conoscere. I concetti principali introdotti sono:

- *Indecidibilità*: esistono molte proprietà importanti dei programmi che non possono essere determinate in modo meccanico da nessun algoritmo; una tra queste è il problema della fermata.
- *Calcolabilità*: una funzione è calcolabile quando esiste un programma che la calcola. L'indecidibilità del problema della fermata afferma che esistono funzioni non calcolabili.
- *Parzialità*: le funzioni espresse da un programma possono essere indefinite su alcuni argomenti, in corrispondenza di quei dati di input per i quali il programma non termina.
- *Turing completezza*: ogni linguaggio di programmazione *general purpose* calcola lo stesso insieme di funzioni, quelle calcolate dalle macchine di Turing.

## 5.5 Nota bibliografica

Il risultato originale di indecidibilità è nell'articolo di A. Turing [98], che dovrebbe essere una lettura obbligata per ogni informatico con qualche sensibilità teorica. Gli argomenti di questo capitolo possono essere approfonditi in un qualsiasi buon testo di teoria della calcolabilità. Tra questi ricordiamo [45], che abbiamo già citato nel Capitolo 2, il classico [85], che dopo quasi quarant'anni continua ad essere uno dei riferimenti più autorevoli, e [5], il primo manuale di calcolabilità (e non solo) scritto per studenti italiani di informatica, che costituisce un'introduzione piana e intuitiva, ma rigorosa, ai concetti cui abbiamo sommariamente accennato.

## 5.6 Esercizi

1. Si dimostri che è indecidibile il problema ristretto della fermata, cioè verificare se un programma termina quando applicato a sè stesso. (Suggerimento: se il problema fosse decidibile, il programma che lo decide dovrebbe avere le stesse proprietà del programma  $K$ ; da cui si deriva l'assurdo nel solito modo).
2. Si dimostri che è indecidibile il problema di verificare se un programma calcola una funzione costante. Suggerimento: dato un generico programma  $P$ , si consideri il programma  $Q_P$ , con un solo input, specificato come segue:

$$Q_P(y) = \begin{cases} 1 & \text{se } P(P) \text{ termina} \\ \text{non termina} & \text{altrimenti.} \end{cases}$$

## I nomi e l'ambiente

L'evoluzione dei linguaggi di programmazione può essere vista in larga misura come un processo che ha portato alla definizione di formalismi sempre più distanti dalla macchina fisica, mediante l'uso di opportuni meccanismi d'astrazione. In questo contesto i *nomi* giocano un ruolo fondamentale: un nome infatti non è altro che una sequenza (possibilmente significativa) di caratteri usata per rappresentare qualche altra cosa e permette di astrarre sia aspetti relativi ai dati, ad esempio denotando con un nome una locazione di memoria, sia aspetti relativi al controllo, ad esempio rappresentando un insieme di comandi con un nome. La gestione corretta dei nomi richiede sia regole semantiche precise che meccanismi implementativi adeguati.

In questo capitolo analizzeremo tali regole soffermandoci in particolare sulla nozione d'ambiente, sui costrutti usati per la sua strutturazione e sulle regole di visibilità (o di *scope*). Rimandiamo invece al prossimo capitolo la trattazione degli aspetti implementativi relativi a queste questioni. Osserviamo subito come nei linguaggi con procedure la nozione di ambiente necessita per essere definita con precisione anche di altre nozioni, relative al passaggio dei parametri: vedremo queste regole nel Capitolo 9. Nel caso dei linguaggi orientati ad oggetti, infine, si hanno ulteriori specifiche regole di visibilità che esamineremo nel Capitolo 12.

### 6.1 Nomi e oggetti denotabili

Quando in un programma dichiariamo una nuova variabile

```
int pippo;
```

oppure definiamo una nuova funzione

```
int pluto() {
    pippo = 1;
}
```

introduciamo dei nuovi nomi, quali `pippo` e `pluto`, per rappresentare un oggetto (una variabile ed una funzione nel nostro esempio). La sequenza di caratteri `pippo` potrà essere usata tutte le volte che ci si vorrà riferire alla nuova variabile,

così come la sequenza di caratteri `pluto` ci permetterà di chiamare la funzione che assegna a `pippo` il valore 1.

Un nome dunque non è altro che una sequenza di caratteri usata per rappresentare, o denotare, un altro oggetto<sup>1</sup>.

Nella maggior parte dei linguaggi i nomi sono costituiti da identificatori, ossia da token alfanumerici, tuttavia possono essere nomi anche altri simboli: ad esempio `+` e `*` sono nomi che denotano in genere operazioni primitive.

Per quanto ovvio, è importante sottolineare che il nome e l'oggetto da questo denotato *non* sono la stessa cosa: il nome infatti è solo una sequenza di caratteri, mentre la sua denotazione può essere un oggetto complesso, quale una variabile, una funzione, un tipo ecc. E difatti uno stesso oggetto può avere più nomi (si parla in questo caso di *aliasing*), mentre uno stesso nome può denotare oggetti diversi in momenti diversi. Quando dunque si usa la terminologia “la variabile `pippo`” o “la funzione `pluto`”, come è comune ed ammissibile fare, si ricordi che queste sono abbreviazioni per “la variabile di nome `pippo`” e “la funzione di nome `pluto`”. Più in generale, nella pratica programmatica, quando si usa un nome quasi sempre si intende riferirsi all'oggetto da esso denotato.

L'uso dei nomi realizza un primo meccanismo elementare di *astrazione sui dati*. Quando definiamo un nome per una variabile di un linguaggio imperativo, ad esempio, introduciamo un identificatore simbolico per una locazione di memoria, astraendo così dai dettagli di basso livello relativi agli indirizzi di memoria. Se poi usiamo il comando di assegnamento

```
pippo = 2;
```

il valore 2 sarà memorizzato nella locazione che è stata riservata per la variabile di nome `pippo` e l'uso del nome evita di doversi preoccupare, a livello di programma, di quale sia questa locazione. La corrispondenza fra nome e locazione di memoria dovrà essere garantita dall'implementazione e chiameremo *ambiente* quella parte dell'implementazione che è responsabile delle associazioni tra i nomi e gli oggetti che questi denotano. Vedremo meglio nel Paragrafo 8.2.1 cosa sia esattamente una variabile e come questa sia associabile a dei valori.

I nomi sono fondamentali anche per realizzare una forma di *astrazione sul controllo*: una procedura<sup>2</sup> non è altro che un nome associato ad un insieme di comandi, insieme a determinate regole di visibilità che rendono disponibile al programmatore la sola interfaccia, costituita dal nome della procedura e dagli eventuali parametri. Vedremo gli aspetti specifici di astrazione del controllo nel Capitolo 9.

<sup>1</sup>Qui e nel seguito di questo capitolo oggetto è inteso in senso lato, senza alcun riferimento al termine tecnico usato nell'ambito dei linguaggi orientati ad oggetti

<sup>2</sup>Qui e altrove, useremo il generico termine “procedura” per indicare anche le funzioni, i metodi e i sottoprogrammi; si veda anche il Paragrafo 9.1.

### 6.1.1 Oggetti denotabili

Gli oggetti ai quali può essere dato un nome si dicono *oggetti denotabili*. Anche se nei vari linguaggi di programmazione si possono osservare ampie differenze, la seguente è una lista non esaustiva di possibili oggetti denotabili:

- oggetti i cui nomi sono definiti dall'utente: variabili, parametri formali, procedure (in senso lato), tipi definiti dall'utente, etichette, moduli, costanti definite dall'utente, eccezioni;
- oggetti i cui nomi sono definiti dal linguaggio di programmazione: tipi primitivi, operazioni primitive, costanti predefinite.

Il legame (o associazione, o *binding* secondo la terminologia inglese) fra un nome e l'oggetto da esso denotato può dunque avvenire in momenti diversi: alcuni nomi sono associati a degli oggetti al momento della progettazione di un linguaggio, mentre altre associazioni sono introdotte solo al momento dell'esecuzione di un programma. Più in dettaglio, considerando tutto il processo che va dalla definizione di un linguaggio di programmazione all'esecuzione di uno specifico programma, possiamo identificare le fasi descritte qui sotto riguardo alla creazione delle associazioni fra nomi e oggetti.

**Progettazione del linguaggio:** in questa fase sono definite le associazioni fra nomi e costanti primitive, tipi primitivi e operazioni primitive del linguaggio (ad esempio, `+` per indicare la somma, `int` per indicare il tipo degli interi ecc.).

**Scrittura del programma:** dato che il programmatore sceglie i nomi quando scrive il programma, possiamo considerare questa fase come quella in cui inizia la definizione di alcune associazioni che poi saranno completate successivamente. L'associazione fra un identificatore e una variabile, ad esempio, è definita dal programma ma viene effettivamente realizzata solo al momento in cui lo spazio per la variabile sarà allocato in memoria.

**Compilazione (compile-time):** il compilatore, traducendo i costrutti del linguaggio di alto livello in codice macchina, alloca anche spazio di memoria per alcune strutture dati che possono essere gestite staticamente. Ad esempio, questo è il caso delle variabili globali di un programma, per le quali al momento della compilazione viene creato il legame fra l'identificatore della variabile e la corrispondente locazione di memoria.

**Esecuzione (run-time):** con questo termine indichiamo tutto l'arco temporale che va dall'inizio alla fine dell'esecuzione di un programma. Tutte le associazioni che non siano state precedentemente definite devono essere realizzate a tempo d'esecuzione. Questo è il caso, ad esempio, delle associazioni fra identificatori di variabili e locazioni di memoria per le variabili locali di una procedura ricorsiva, oppure delle variabili di tipo puntatore per le quali si alloca dinamicamente memoria.

Nella precedente descrizione abbiamo ignorato altre fasi importanti, quali quella di collegamento (*linking*) e quella di caricamento (*loading*) nelle quali possono essere realizzate altre associazioni (ad esempio, per dei nomi esterni che si

riferiscono ad oggetti di altri moduli). Nella pratica comunque, si distinguono sostanzialmente due fasi principali usando i termini “statico” e “dinamico”: con “statico” ci si riferisce a tutto quello che avviene prima dell’esecuzione, mentre con “dinamico” si indica tutto quello che avviene al momento dell’esecuzione. Così, ad esempio, la gestione statica della memoria è quella operata dal compilatore, mentre quella dinamica è quella realizzata da opportune operazioni eseguite dalla macchina astratta a tempo d’esecuzione.

## 6.2 Ambiente e blocchi

Non tutte le associazioni fra nomi e oggetti denotabili sono fissate una volta per tutte all’inizio dell’esecuzione del programma: molte possono variare durante l’esecuzione. Per poter comprendere correttamente come si comportano tali associazioni dobbiamo introdurre il concetto di ambiente.

**Definizione 6.1 (Ambiente)** *L’insieme delle associazioni fra nomi e oggetti denotabili esistenti a run-time in uno specifico punto del programma ed in uno specifico momento dell’esecuzione è detto ambiente (referencing environment).*

Usualmente quando si parla di ambiente ci si riferisce solo alle associazioni che non sono stabilite dalla definizione del linguaggio. L’ambiente è dunque quella componente della macchina astratta che, per ogni nome introdotto dal programmatore e in ogni punto del programma, permette di determinare quale sia l’associazione corretta. Si noti che l’ambiente non esiste a livello della macchina fisica: la presenza dell’ambiente costituisce una delle principali caratteristiche dei linguaggi di alto livello che devono essere opportunamente simulate nell’implementazione del linguaggio.

Una *dichiarazione* è un costrutto che permette di introdurre un’associazione nell’ambiente. I linguaggi di alto livello hanno spesso dichiarazioni esplicite, quali

```
int x;
int f () {
    return 0;
}
type T = int;
```

(la prima è la dichiarazione di una variabile, la seconda di una funzione di nome *f*, la terza la dichiarazione di un nuovo tipo *T*, che coincide col tipo *int*). Alcuni linguaggi permettono dichiarazioni implicite, che introducono un’associazione in ambiente per un nome al momento del primo uso di tale nome. Il tipo dell’oggetto denotato viene dedotto dal contesto nel quale il nome è usato la prima volta (o talvolta addirittura dalla forma sintattica del nome).

Come vedremo in dettaglio nel seguito, si hanno vari gradi di libertà nelle associazioni fra nomi e oggetti denotabili. Innanzitutto, uno stesso nome può denotare oggetti diversi in regioni diverse di un programma. Si consideri, ad esempio, il codice della Figura 6.1: il nome *pippo* più esterno denota una variabile intera,

```
{int pippo;
pippo = 2;
{char pippo;
pippo = a;
}
}
```

Figura 6.1 Un nome che denota oggetti diversi.

mentre quello più interno denota una variabile di tipo carattere.

È anche possibile che uno stesso oggetto sia denotato da più nomi in ambienti diversi. Ad esempio, se passiamo una variabile per riferimento ad una procedura, la variabile è accessibile attraverso il suo nome nel programma chiamante e attraverso il nome del parametro formale nel corpo della procedura (si veda il Paragrafo 9.1.2). Oppure possiamo usare i puntatori per creare delle strutture dati nelle quali lo stesso oggetto sia poi raggiungibile attraverso nomi diversi.

Fino a che nomi diversi per lo stesso oggetto vengono usati in ambienti diversi non sorgono particolari problemi. Più complicata è invece la situazione nella quale uno stesso oggetto è visibile mediante nomi diversi nello stesso ambiente. Questa situazione è detta *aliasing* ed i nomi diversi per lo stesso oggetto sono detti “alias”<sup>3</sup>. Il passaggio per riferimento (che discuteremo nel Capitolo 9) è una possibile (e frequente) causa di aliasing: se il nome di variabile che passiamo per riferimento ad una procedura è visibile anche all’interno della procedura stessa, allora il parametro formale e la variabile passata per riferimento si riferiscono alla stessa locazione, dunque una chiara situazione di aliasing. Un altro caso di aliasing si presenta con i campi delle “unioni” (in C), o dei record varianti (in Pascal), che presenteremo nel Capitolo 10.

Situazioni simili, in cui due *espressioni* (e non due *nomi*) denotano lo stesso oggetto, si possono ottenere facilmente usando i puntatori, dato che se *X* e *Y* sono variabili di tipo puntatore l’assegnamento *X=Y* permette di accedere alla stessa locazione usando sia *X* che *Y*. Consideriamo, ad esempio, il seguente frammento di programma C dove, così come faremo in seguito, supponiamo che *write(Z)* sia una procedura che permette di stampare il valore della variabile intera *Z*:

```
int *X, *Y;           // X, Y puntatori a interi
X = (int *) malloc (sizeof (int));
                    // allocata la memoria puntata
*X = 5;              // * dereferenzia
Y=X;                 // Y punta alla stessa oggetto di X
*Y=10;
write(*X);
```

<sup>3</sup>La terminologia è dovuta all’avverbio latino “alias”, di uso corrente sia in italiano che in inglese per indicare uno pseudonimo.

I nomi *x* e *y* denotano due diverse variabili che però, dopo l'esecuzione del comando di assegnamento *y=x*, permettono di accedere alla stessa locazione di memoria (e quindi, il successivo comando di stampa produrrà il valore 10).

È infine possibile che uno stesso nome, in una stessa regione testuale del programma, denoti oggetti diversi a seconda del flusso di esecuzione del programma, cioè del momento in cui il controllo raggiunge quel nome. Questa situazione è più comune di quello che a prima vista potrebbe sembrare. È il caso, ad esempio, di una procedura ricorsiva che dichiara un nome locale. Altri casi del genere, più delicati, saranno discussi più avanti in questo capitolo quando parleremo di scope dinamico (Paragrafo 6.3.2).

## 6.2.1 I blocchi

Quasi tutti i principali linguaggi di programmazione attuali permettono l'uso dei blocchi, un meccanismo di strutturazione dei programmi introdotto in ALGOL 60 che è fondamentale per l'organizzazione dell'ambiente.

**Definizione 6.2 (Blocco)** *Un blocco è una regione testuale del programma, identificata da un segnale di inizio ed uno di fine, che può contenere dichiarazioni locali a quella regione (cioè che compaiono nella regione).*

I costrutti di inizio e fine blocco variano, al solito, da linguaggio a linguaggio: *begin ... end* per i linguaggi della famiglia ALGOL, le parentesi graffe *{...}* per C e Java, le parentesi tonde *(...)* per LISP e i suoi dialetti, *let ... in ... end* per ML ecc. Inoltre, la definizione esatta di blocco in uno specifico linguaggio di programmazione può differire leggermente da quella generale data sopra. In alcuni casi, ad esempio, si parla di blocco solo quando sono effettivamente presenti dichiarazioni locali. Spesso, inoltre, i blocchi hanno anche un'altra funzione importante, quella di "raggruppare" una serie di comandi in un'entità sintattica che possa essere considerata come un unico comando (composto). Queste distinzioni, tuttavia, non sono rilevanti per i nostri scopi. Ci atterremo pertanto alla definizione appena data, distinguendo due casi:

**blocco associato ad una procedura:** è il blocco associato alla dichiarazione di una procedura e che testualmente corrisponde al corpo della procedura stessa, esteso con le dichiarazioni relative ai parametri formali;

**blocco in-line (o anonimo):** è il blocco che non corrisponde ad una dichiarazione di procedura, e che può pertanto comparire (in genere) in una qualsiasi posizione dove sia richiesto un comando.

## 6.2.2 Tipi di ambiente

L'ambiente cambia durante l'esecuzione di un programma, tuttavia i cambiamenti avvengono generalmente in due momenti ben precisi: all'entrata e all'uscita di un

blocco. Il blocco quindi può essere considerato come il costrutto di granularità più piccola al quale può essere associato un ambiente costante<sup>4</sup>.

L'ambiente di un blocco, intendendo con questa terminologia l'ambiente esistente nel momento in cui il blocco è eseguito, è costituito in primo luogo dalle associazioni relative ai nomi dichiarati localmente al blocco stesso. Nella maggior parte dei linguaggi che permettono i blocchi, questi possono essere "annidati", ossia la definizione di un blocco può essere interamente inclusa in quella di un altro. Un esempio di blocchi annidati anonimi è quello contenuto nell'esempio in Figura 6.1. Non sono invece mai permesse sovrapposizioni di blocchi nelle quali l'ultimo blocco aperto non sia il primo blocco ad essere chiuso. Ossia, una sequenza di comandi del tipo

```
apri blocco A;
    apri blocco B;
    chiudi blocco A;
        chiudi blocco B;
```

non è permessa in alcun linguaggio.

I vari linguaggi differiscono poi nel tipo di annidamento permesso: in C, ad esempio, blocchi associati a procedure non possono essere annidati tra loro (cioè non vi possono essere dichiarazioni di procedure all'interno di altre procedure), mentre in Pascal e Ada tale restrizione non è presente<sup>5</sup>.

L'annidamento dei blocchi costituisce un importante strumento di strutturazione dell'ambiente, con meccanismi che permettono di "vedere" le dichiarazioni locali di un blocco anche da parte dei blocchi in questo annidati.

In modo per il momento informale, diciamo che una dichiarazione locale ad un blocco è *visibile* in un altro blocco, se l'associazione creata per effetto di quella dichiarazione è presente nell'ambiente del secondo blocco. Sono chiamate *regole di visibilità* quei meccanismi di un linguaggio che regolano come e quando una dichiarazione è visibile. La regola di visibilità canonica per i linguaggi con blocchi è ben nota:

Una dichiarazione locale ad un blocco è visibile in quel blocco e in tutti i blocchi in esso annidati, a meno che non intervenga in tali blocchi una nuova dichiarazione dello stesso nome. In tal caso, nel blocco in cui compare la ridefinizione la nuova dichiarazione *nasconde* (o maschera) la precedente.

Nel caso in cui vi sia ridefinizione, la regola di visibilità stabilisce che nel blocco interno sarà visibile solo l'ultimo nome dichiarato, mentre per quello esterno si avrà un "buco di visibilità": l'associazione per il nome dichiarato nel blocco esterno infatti sarà disattivata per tutto il blocco interno (contenente la nuova dichiarazione) e sarà riattivata all'uscita da tale blocco. Si osservi come non vi sia

<sup>4</sup>Qui assumiamo per semplicità che tutte le dichiarazioni locali del blocco siano valutate quando si entra nel blocco e siano visibili in tutto il blocco; esistono innumerevoli eccezioni a questa situazione, alcune delle quali saranno discusse nel seguito.

<sup>5</sup>Le motivazioni delle restrizioni di C saranno chiare solo nel prossimo capitolo, quando avremo discusso delle tecniche di implementazione delle regole di scope.

visibilità dall'esterno verso l'interno: ogni associazione introdotta nell'ambiente locale di un blocco interno non è attiva (ossia il nome che essa definisce non è visibile) in un blocco esterno che contenga il blocco interno. Analogamente, se abbiamo due blocchi allo stesso livello di annidamento, ovvero se nessuno dei due contiene l'altro, un nome introdotto localmente in un blocco non è visibile nell'altro.

La definizione appena discussa, pur apparentemente precisa, in alcuni casi non è in grado di stabilire con precisione quale sia l'ambiente in un generico punto di un programma. Assumeremo questa regola per il resto di questo paragrafo, mentre il prossimo si incaricherà proprio di enunciare correttamente le regole di visibilità.

In generale, possiamo identificare tre componenti dell'ambiente, come descritto dalla seguente definizione.

**Definizione 6.3 (Tipi d'ambiente)** *L'ambiente associato ad un blocco è costituito dalle tre parti seguenti:*

**ambiente locale:** *quello costituito dall'insieme delle associazioni per nomi dichiarati localmente al blocco; nel caso in cui il blocco sia relativo ad una procedura l'ambiente locale contiene anche le associazioni relative ai parametri formali, dato che questi possono essere visti, ai fini dell'ambiente, come variabili dichiarate localmente;*

**ambiente non locale:** *questo è l'ambiente costituito dalle associazioni relative ai nomi che sono visibili all'interno di un blocco ma che non sono stati dichiarati localmente;*

**ambiente globale:** *infine questo è l'ambiente costituito dalle associazioni create all'inizio dell'esecuzione del programma principale. Contiene dunque le associazioni per i nomi che sono usabili in tutti i blocchi che compongono il programma.*

L'ambiente locale di un blocco può essere determinato considerando solo le dichiarazioni presenti nel blocco, mentre per definire l'ambiente non locale si deve guardare anche all'esterno del blocco. L'ambiente globale fa parte dell'ambiente non locale. I nomi che sono introdotti nell'ambiente locale possono essere gli stessi presenti nell'ambiente non locale: in tale caso la dichiarazione più interna (locale) nasconde quella più esterna.

Le regole di visibilità specificano come i nomi dichiarati in blocchi esterni sono visibili nei blocchi più interni. In alcuni casi inoltre è possibile importare nomi da altri moduli, definiti separatamente. Le associazioni per tali nomi fanno parte dell'ambiente globale.

Vediamo adesso l'esempio in Figura 6.2 dove, per comodità di riferimento, assumiamo che i blocchi possano essere etichettati (come prima, assumiamo anche che `write(x)` permetta di stampare un valore intero); l'etichetta si comporta come un commento ai fini dell'esecuzione.

Supponendo che il blocco A sia quello più esterno, corrispondente al programma principale, la dichiarazione della variabile `a` introduce un'associazione nell'ambiente globale.

```
A:{int a =1;

B:{int b = 2;
    int c = 2;

C:{int c =3;
    int d;
    d = a+b+c;
    write(d)
}

D:{int e;
    e = a+b+c;
    write(e)
}
}
```

**Figura 6.2** Blocchi annidati con ambienti diversi.

All'interno del blocco B vengono dichiarate due variabili locali (`b` e `c`). L'ambiente di B è dunque costituito dall'ambiente locale, contenente l'associazione per i due nomi (`b` e `c`) e dall'ambiente globale, contenente l'associazione per `a`.

All'interno del blocco C vengono dichiarate due variabili locali (`c` e `d`). L'ambiente di C è dunque costituito dall'ambiente locale, contenente l'associazione per i due nomi (`c` e `d`) e dall'ambiente non locale, contenente lo stesso ambiente globale di prima ed anche l'associazione per il nome `b` che è "ereditata" dall'ambiente del blocco B. Si noti che la dichiarazione locale di `c` nel blocco C nasconde la dichiarazione di `c` presente nel blocco B: il comando di stampa presente nel blocco C stamperà quindi il valore 6.

Nel blocco D, infine, abbiamo un ambiente locale, contenente l'associazione per il nome locale `e`, il solito ambiente globale e l'ambiente non locale che, oltre all'associazione per `a`, contiene le associazioni per i nomi `b` e `c` introdotti nel blocco B. Dato che la variabile `c` non è stata ridichiarata internamente, in questo caso dunque rimane visibile la variabile dichiarata nel blocco B ed il valore stampato sarà 5. Si noti anche che l'associazione per il nome `d` non compare nell'ambiente non locale di D, dato che tale nome è introdotto in un blocco esterno che non contiene D. Le regole di visibilità infatti permettono solo di ereditare nomi dichiarati nei blocchi esterni da parte dei blocchi interni, e non viceversa.

### 6.2.3 Operazioni sull'ambiente

Come abbiamo visto, i cambiamenti nell'ambiente sono prodotti all'entrata e all'uscita di un blocco. Più in dettaglio, nel momento in cui, durante l'esecuzione

di un programma, si entra in un nuovo blocco si hanno le seguenti modifiche dell'ambiente:

1. sono create le associazioni fra i nomi dichiarati localmente al blocco e i relativi oggetti denotati;
2. sono disattivate le associazioni per quei nomi già esistenti all'esterno del blocco che siano ridefiniti al suo interno.

Anche all'uscita da un blocco l'ambiente viene modificato in quanto:

1. sono distrutte le associazioni fra i nomi dichiarati localmente al blocco e i relativi oggetti denotati;
2. sono riattivate le associazioni per i nomi, già esistenti all'esterno del blocco, che erano stati ridefiniti al suo interno.

Più in generale, possiamo identificare le seguenti operazioni sui nomi e sull'ambiente:

**Creazione di un'associazione fra nome ed oggetto denotato (naming):** corrisponde all'elaborazione di una dichiarazione (o di un legame fra parametro formale e parametro attuale) al momento in cui si entra in un nuovo blocco che contiene dichiarazioni locali e/o parametri.

**Riferimento di oggetto denotato mediante il suo nome (referencing):** corrisponde all'uso di un nome (in un'espressione, in un comando, o in un qualsiasi altro contesto); il nome serve per accedere all'oggetto denotato.

**Disattivazione di un'associazione fra il nome e l'oggetto denotato:** corrisponde all'ingresso in un blocco dove viene creata localmente una nuova associazione per quel nome; la vecchia associazione non è distrutta ma rimane (inattiva) nell'ambiente. Essa sarà di nuovo utilizzabile quando si uscirà dal blocco contenente la nuova associazione.

**Riattivazione di un'associazione fra il nome e l'oggetto denotato:** avviene quando si esce da un blocco dove era stata creata localmente una nuova associazione per quel nome. La vecchia associazione, che era stata disattivata all'entrata nel blocco, è adesso nuovamente utilizzabile.

**Distruzione di un'associazione fra il nome e l'oggetto denotato (unnaming):** viene effettuata, relativamente alle associazioni locali, quando si esce dal blocco dove tali associazioni erano state create. L'associazione viene rimossa dall'ambiente e non sarà più utilizzabile.

Osserviamo esplicitamente, dunque, che un certo ambiente contiene sia associazioni attive che associazioni disattivate (corrispondenti a dichiarazioni mascherate per effetto delle regole di visibilità). Per quanto riguarda gli oggetti denotabili, sono possibili le seguenti operazioni:

**Creazione di un oggetto denotabile:** questa operazione avviene allocando la memoria necessaria a contenere l'oggetto; talvolta la creazione include anche l'inizializzazione dell'oggetto.

**Accesso ad un oggetto denotabile:** tramite il nome, e quindi l'ambiente, possiamo accedere all'oggetto denotabile e quindi ottenerne il valore (ad esempio, per leggere il contenuto di una variabile); osserviamo come l'insieme delle regole che definiscono l'ambiente ha come scopo quello di rendere univoca (in un fissato punto del programma e in una fissata esecuzione) l'associazione tra un nome e l'oggetto cui si riferisce.

**Modifica di un oggetto denotabile:** sempre tramite il nome possiamo accedere all'oggetto denotabile e quindi modificarne il valore (ad esempio, assegnando un valore ad una variabile).

**Distruzione di un oggetto denotabile:** un oggetto può essere distrutto deallocando la memoria che era stata riservata per esso.

In molti linguaggi le operazioni di creazione di un'associazione fra nome e oggetto denotato e quella di creazione dell'oggetto denotato avvengono allo stesso tempo. Questo è il caso, ad esempio, di una dichiarazione della forma

```
int x;
```

Tale dichiarazione introduce nell'ambiente una nuova associazione fra il nome x e una variabile intera e allo stesso tempo alloca la memoria per la variabile.

Tuttavia, questo non vale sempre e, in generale, non è detto che il *tempo di vita* di un oggetto denotabile, ossia il tempo che intercorre fra la creazione dell'oggetto e la sua distruzione, coincida con il *tempo di vita* dell'associazione fra nome e oggetto. Infatti, un oggetto denotabile può avere un tempo di vita maggiore di quello dell'associazione fra un nome e l'oggetto stesso, come nel caso in cui si passi per riferimento una variabile ad una procedura: l'associazione fra il parametro formale e la variabile ha un tempo di vita inferiore a quello della variabile stessa. Più in generale, una situazione di questo tipo si ha quando viene introdotto un nome temporaneo (ad esempio locale in un blocco) per un oggetto che ha già un nome.

Si noti che la situazione della quale stiamo parlando *non* è quella descritta nell'esempio di Figura 6.2: in tale esempio infatti la dichiarazione interna della variabile c *non* introduce un nuovo nome per un oggetto esistente, ma introduce un nuovo oggetto (una nuova variabile).

Anche se a prima vista può apparire strano, può accadere anche che il tempo di vita di un'associazione fra nome e oggetto denotato sia superiore a quello dell'oggetto stesso; più precisamente, può avvenire che un nome permetta di accedere ad un oggetto che non esiste più. Una tale situazione anomala si può avere, ad esempio, se si passa per riferimento un oggetto creato e quindi si dealloca la memoria per tale oggetto prima che la procedura termini: il parametro formale della procedura in tale caso si troverà a denotare un oggetto che non esiste più. Una situazione di questo tipo, nella quale si può accedere ad un oggetto la cui memoria è stata deallocata, è detta "dangling reference" (letteralmente riferimento pendente) ed è sintomo di un qualche errore. Riprenderemo il problema dei dangling reference nel Capitolo 10, dove presenteremo anche alcune tecniche per la loro gestione.

## 6.3 Regole di scope

Abbiamo visto come, all'ingresso e all'uscita di un blocco, l'ambiente possa cambiare in seguito a operazioni di creazione, distruzione, attivazione e disattivazione di associazioni. Tali cambiamenti sono abbastanza chiari per l'ambiente locale, ma possono essere meno evidenti per quanto riguarda l'ambiente non locale: le regole di visibilità enunciate nel paragrafo precedente si prestano infatti ad almeno due interpretazioni distinte. Si consideri ad esempio il seguente frammento di programma:

```
A:{int x = 0;

void pippo() {
    x = 1;
}

B:{int x;
    pippo();
}

write(x);
}
```

Quale valore viene stampato? Per rispondere a questa domanda il problema fondamentale è sapere a quale dichiarazione del nome *x* fa riferimento l'occorrenza non locale di questo nome che compare nell'assegnamento nel corpo della procedura *pippo*. Da un lato potremmo ragionevolmente pensare che venga stampato il valore 1, dato che la procedura *pippo* è definita nel blocco A e quindi la *x* che compare nel corpo della procedura potrebbe essere quella definita nella prima riga di A. D'altro canto, però, potremmo anche ragionare come segue: quando chiamiamo la procedura *pippo* siamo nel blocco B, quindi la *x* che usiamo nel comando di assegnamento presente nel corpo di *pippo* è la variabile dichiarata localmente al blocco B. Tale variabile locale non è più visibile quando usciamo dal blocco B, quindi il comando *write(x)* si riferisce alla variabile *x* dichiarata ed inizializzata a 0 nel blocco A e mai più modificata. *Ergo* la procedura stampa il valore 0.

Prima che il lettore si affanni a cercare l'eventuale trucco nei due precedenti ragionamenti, diciamo che essi sono entrambi legittimi: il risultato del frammento di programma infatti dipende da qual è la *regola di scope* adottata, come sarà chiaro alla fine di questo paragrafo. La regola di visibilità che abbiamo enunciato poc'anzi stabilisce che "una dichiarazione locale ad un blocco è visibile in quel blocco e in tutti i blocchi in esso annidati", ma non specifica se tale nozione di annidamento debba essere considerata in modo statico (cioè basata sul testo del programma) o in modo dinamico (cioè basata sul flusso d'esecuzione). Quando le regole di visibilità, dette anche regole di *scope*, dipendono solo dalla struttura sintattica del programma, parliamo di un linguaggio con *scope statico*, o lessicale. Quando esse sono influenzate anche dal flusso del controllo a run-time, siamo in presenza di un linguaggio con *scope dinamico*. Nei due seguenti paragrafi analizzeremo in dettaglio i due casi.

### 6.3.1 Scope statico

In un linguaggio con scope statico (o lessicale), l'ambiente esistente in un qualsiasi punto del programma ed in un qualsiasi momento dell'esecuzione dipende unicamente dalla struttura sintattica del programma stesso. Un tale ambiente può essere dunque determinato completamente dal compilatore, da cui il termine "statico".

Ovviamente vi possono essere varie regole di scope statico. Una delle più semplici, ad esempio, è quella delle prime versioni del linguaggio Basic che permetteva un solo ambiente globale nel quale era possibile usare solo pochi nomi (qualche centinaio) e dove non venivano usate dichiarazioni.

Molto più interessante è la regola di scope statico usata nei linguaggi a blocchi con possibilità di annidamento, introdotta in ALGOL 60 e presente con poche modifiche in molti linguaggi moderni, inclusi Ada, Pascal e Java. Tale regola può essere definita come segue.

**Definizione 6.4 (Scope statico)** *La regola dello scope statico, o regola dello scope annidato più vicino, è definita dai seguenti tre punti:*

- (i) *Le dichiarazioni locali di un blocco definiscono l'ambiente locale di quel blocco. Le dichiarazioni locali di un blocco includono solo quelle presenti nel blocco (usualmente all'inizio del blocco stesso) e non quelle eventualmente presenti in blocchi annidati all'interno del blocco in questione.*
- (ii) *Se si usa un nome all'interno di un blocco l'associazione valida per tale nome è quella presente nell'ambiente locale del blocco, se esiste. Se non esiste alcuna associazione per il nome nell'ambiente locale del blocco si considerano le associazioni esistenti nell'ambiente locale del blocco immediatamente esterno che contiene il blocco di partenza. Se in questo blocco si trova un'associazione essa è quella valida, altrimenti si prosegue con i blocchi esterni che contengono il blocco di partenza, dal più vicino al più lontano. Se procedendo in questo modo si raggiunge il blocco più esterno del programma e questo non contiene alcuna associazione per il nome, allora tale associazione deve essere cercata nell'ambiente predefinito del linguaggio. Se anche qui l'associazione non esiste, allora si ha un errore.*
- (iii) *Un blocco può avere un nome, nel qual caso tale nome fa parte dell'ambiente locale del blocco immediatamente esterno che contiene il blocco a cui abbiamo dato un nome. Questo è il caso anche dei blocchi associati alle procedure.*

È immediato osservare che questa definizione corrisponde alla regola di visibilità informale che abbiamo già discusso, opportunamente completata con un'interpretazione statica della nozione di annidamento.

Tra i vari dettagli della regola, non sfugga il fatto che la dichiarazione di una procedura introduce un'associazione per il nome della procedura nell'ambiente locale del blocco che contiene la dichiarazione (l'associazione dunque, a causa dell'annidamento, è visibile anche nel blocco che costituisce il corpo della procedura, il che permette procedure ricorsive). I parametri formali della procedura

```

int x = 0;
void pippo(int n){
    x = n+1;
}
pippo(3);
write(x);
{int x = 0;
    pippo(3);
    write(x);
}
write(x);
}

```

**Figura 6.3** Un esempio per lo scope statico.

tuttavia sono presenti solo nell'ambiente locale della procedura e non sono visibili nell'ambiente che contiene la dichiarazione della procedura.

In un linguaggio con scope statico, chiamiamo *scope di una dichiarazione*<sup>6</sup> quella porzione del programma nella quale la dichiarazione è visibile secondo la Definizione 6.4.

Concludiamo la nostra analisi dello scope statico discutendo l'esempio della Figura 6.3: la prima e la terza occorrenza della `write` stampano il valore 4, mentre la seconda occorrenza di `write` stampa il valore 0. Si noti che il parametro formale `n` non è visibile al di fuori del corpo della procedura.

Lo scope statico permette di determinare tutti gli ambienti presenti nel programma semplicemente leggendone il testo. Questo ha due importanti conseguenze positive. Innanzitutto, il programmatore ha una migliore comprensione del programma, in quanto può collegare ogni occorrenza di un nome alla sua dichiarazione corretta osservando la struttura testuale del programma, senza dover simulare la sua esecuzione. Inoltre, tale collegamento può essere fatto anche dal compilatore, che quindi può individuare quali siano le dichiarazioni corrette per ogni uso di un nome. Questo fa sì che siano possibili a tempo di compilazione un maggior numero di controlli di correttezza, usando le informazioni contenute nei tipi, e un maggior numero di ottimizzazioni del codice. Ad esempio, se il compilatore "sa" (dalla dichiarazione) che la variabile `x` che occorre in un blocco è una variabile intera, allora segnalerà un errore nel caso in cui si assegni a tale variabile un carattere. Analogamente, se il compilatore sa che la costante `pippo` è associata al valore 10, esso potrà sostituire ogni riferimento a `pippo` con il valore 10, evitando quindi che una tale operazione debba essere fatta a run-time e quindi ottimizzando il codice. Se invece la dichiarazione corretta per `x` e per `pippo` può

<sup>6</sup>La terminologia italiana è lungi dall'esser concorde. Vengono usati *portata*, *campo d'azione*, *ambito*, *estensione*.

```

const x = 0;
void pippo(){
    write(x);
}
void pluto(){
    const x = 1;
    pippo();
}
pluto();
}

```

**Figura 6.4** Un esempio per lo scope dinamico.

essere determinata solo al momento dell'esecuzione, è chiaro che questi controlli e ottimizzazioni non sono possibili al momento della compilazione.

Si noti che, anche con la regola di scope statico, il compilatore non può sapere in generale quale sia la locazione di memoria per la variabile di nome `x`, né tanto meno quale sia il suo valore, dato che queste informazioni dipendono dall'esecuzione del programma. In regime di scope statico, tuttavia, il compilatore conosce comunque alcune informazioni importanti relative alla memorizzazione delle variabili (in particolare conosce l'offset relativamente ad una posizione fissata, come vedremo in dettaglio nel prossimo capitolo), che usa per compilare efficientemente gli accessi alle variabili. Come vedremo, queste informazioni non sono disponibili in scope dinamico, che, quindi, risulta meno efficiente in esecuzione. Per questi motivi, la maggior parte dei linguaggi attuali (ad esempio ALGOL, Pascal, C, C++, Ada, Scheme e Java) usano una qualche forma di scope statico.

### 6.3.2 Scope dinamico

Lo scope dinamico è stato introdotto in alcuni linguaggi, quali ad esempio APL, LISP (alcune versioni), SNOBOL e PERL, principalmente per semplificare la gestione a run-time dell'ambiente. A fronte dei vantaggi visti nel paragrafo precedente, infatti, lo scope statico richiede una gestione a run-time relativamente complicata, perché i vari ambienti non locali evolvono in modo diverso dal normale flusso di attivazione e disattivazione dei blocchi. Cerchiamo di capire il problema considerando il frammento di codice della Figura 6.4 e seguendone l'esecuzione: innanzitutto si entra nel blocco esterno e si creano le associazioni fra il nome `x` e la costante 0, e fra i nomi `pippo` e `pluto` e le relative procedure (come detto in precedenza, queste associazioni in realtà possono essere realizzate dal compilatore). Quindi si esegue la chiamata di procedura `pluto` e si entra in un secondo blocco relativo a tale procedura. In tale blocco viene creato il legame fra il nome `x` e la costante 1 e viene poi eseguita la chiamata di procedura `pippo` che causa l'entrata in un nuovo blocco (relativo a quest'ultima procedura). È a questo

punto che si esegue il comando `write(x)` e dato che la `x` non è un nome locale del blocco introdotto dalla procedura `pippo`, l'associazione per il nome `x` deve essere cercata nei blocchi precedenti. Tuttavia, secondo le regole di scope statico viste nel precedente paragrafo, il primo blocco esterno nel quale cercare l'associazione per `x` non è l'ultimo che è stato attivato (quello della procedura `pluto` nel nostro esempio) ma dipende dalla struttura del programma. Nel nostro caso quindi, l'associazione corretta per il nome `x` usato da `pippo` è quella presente nel primo blocco e di conseguenza viene stampato il valore 0; il blocco relativo alla procedura `pluto`, anche se contiene una dichiarazione per `x` ed è ancora attivo, non viene considerato.

Generalizzando dal precedente esempio possiamo dire che con lo scope statico la sequenza di blocchi che si devono considerare per risolvere i riferimenti a nomi non locali è diversa dalla sequenza di blocchi aperti e chiusi nel normale flusso di esecuzione del programma. Quest'ultima infatti può essere gestita in modo naturale secondo la politica LIFO (Last In First Out), ossia usando una pila: durante l'esecuzione, il primo blocco dal quale si esce è l'ultimo nel quale siamo entrati. La sequenza dei blocchi da considerarsi per realizzare lo scope statico invece dipende dalla struttura sintattica del programma e per poterla gestire correttamente al momento dell'esecuzione serviranno delle strutture dati aggiuntive, come vedremo meglio nel prossimo capitolo.

Per semplificare la gestione a run-time dell'ambiente alcuni linguaggi usano la regola dello *scope dinamico* e determinano le associazioni fra nomi e oggetti denotati seguendo a ritroso l'esecuzione del programma. Tali linguaggi cioè, per risolvere i nomi non locali, usano la sola struttura a pila impiegata per la gestione a run-time dei blocchi. Nel nostro esempio questo significa che, quando si esegue il comando `write(x)`, invece che nel primo blocco, si va a cercare l'associazione per il nome `x` nel secondo blocco (quello relativo alla procedura `pluto`), perché questo è l'ultimo blocco, diverso da quello attuale, nel quale siamo entrati e dal quale non siamo ancora usciti. Dato che nel secondo blocco troviamo la dichiarazione `const x = 1`, nel caso di scope dinamico il precedente programma stampa il valore 1.

Con più precisione la regola dello scope dinamico, detta anche regola dell'associazione più recente, può essere definita come segue.

**Definizione 6.5 (Scope dinamico)** *Secondo la regola dello scope dinamico, l'associazione valida per un nome X, in un qualsiasi punto P di un programma, è la più recente (in senso temporale) associazione creata per X che sia ancora attiva quando il flusso di esecuzione arriva a P.*

È opportuno osservare che questa regola non è contraddittoria con la regola di visibilità informale che abbiamo enunciato nel Paragrafo 6.2.2. Un momento di riflessione mostra, infatti, come la regola di scope dinamico esprima nient'altro che la stessa regola di visibilità, nella quale la nozione di annidamento tra blocchi è intesa in senso dinamico.

```
(const x = 0;
void pippo(){
    write(x);
}
void pluto(){
    const x = 1;
    {const x = 2;
    }
    pippo();
}
pluto();
```

Figura 6.5 Un altro esempio di scope dinamico.

Osserviamo ancora come la differenza tra scope statico e dinamico interviene solo per la determinazione dell'ambiente che è contemporaneamente *non locale e non globale*: per l'ambiente locale e quello globale le due regole coincidono.

Concludiamo discutendo l'esempio della Figura 6.5: in un linguaggio con scope dinamico, il codice stampa il valore 1 perché quando viene eseguito il comando `write(x)` l'ultima associazione creata per `x`, *che sia ancora attiva*, associa `x` a 1. L'associazione che associa `x` a 2, anche se è la più recente ad essere stata creata, non è più attiva quando viene eseguita la procedura `pippo` e dunque non è considerata.

Si noti che lo scope dinamico permette di modificare il comportamento di procedure o di sottoprogrammi senza usare parametri esplicativi ma solo ridefinendo delle variabili non locali usate dalla procedura. Per esemplificare questo punto, si supponga di avere una procedura `visualizza(testo)` che può visualizzare un testo in vari colori, a seconda del valore della variabile non locale `colore`. Se assumiamo che nella maggior parte dei casi la procedura sia usata per visualizzare testi in nero, è naturale supporre che non si voglia introdurre un altro parametro alla procedura per determinare il colore. Se il linguaggio usa lo scope dinamico, nel caso in cui la procedura debba visualizzare un testo in rosso basterà introdurre una nuova dichiarazione per la variabile `colore` prima della *chiamata* della procedura. Possiamo quindi scrivere

```
...
var colore = rosso;
visualizza(testo);
}
```

e la chiamata di procedura `visualizza` adesso userà il colore rosso, per effetto dello scope dinamico.

Questa flessibilità dello scope dinamico, se da un lato è vantaggiosa, dall'altro spesso rende più difficile la comprensione di un programma, dato che la stessa chiamata di procedura, nelle stesse identiche condizioni salvo una variabile non

locale, può fornire risultati diversi. E se la variabile (*colore* nel nostro esempio) venisse modificata in una zona del programma molto lontana dalla chiamata di procedura la comprensione di quello che accade potrebbe risultare ardua.

Per questo motivo, oltre che per una scarsa efficienza della gestione a runtime, lo scope dinamico rimane poco usato dai linguaggi moderni *general purpose*, che preferiscono invece regole di scope statico.

#### Approfondimento

6.1

## 6.4 Sommario del capitolo

In questo capitolo abbiamo visto i principali aspetti riguardanti la gestione dei nomi in un linguaggio di alto livello. La presenza dell'ambiente, ossia di un insieme di associazioni fra nomi e oggetti che essi rappresentano, costituisce una delle principali caratteristiche che differenziano i linguaggi di alto livello da quelli di basso livello. Data la mancanza dell'ambiente nei linguaggi di basso livello, la gestione dei nomi e dell'ambiente è una componente importante nell'implementazione di un linguaggio di alto livello. Vedremo gli aspetti implementativi relativi alla gestione dei nomi nel prossimo capitolo. Qui ci siamo invece soffermati su quegli aspetti che dovrebbero essere noti ad un qualsiasi utente (programmatore) di un linguaggio di alto livello per una comprensione accurata del significato dei nomi e, quindi, del comportamento dei programmi. In particolare, abbiamo analizzato gli aspetti elencati qui sotto.

- *La nozione di oggetto denotabile*: sono questi gli oggetti ai quali si può dare un nome. Gli oggetti denotabili variano a seconda del linguaggio considerato anche se alcune categorie di oggetti (ad esempio le variabili) sono abbastanza generali.
- *L'ambiente*: l'insieme delle associazioni esistenti a tempo di esecuzione fra nomi e oggetti denotabili.
- *I blocchi*: in-line o associati alle procedure, sono i costrutti fondamentali per la strutturazione dell'ambiente e per la definizione delle regole di visibilità.
- *Tipi di ambiente*: ossia le tre componenti che in ogni momento caratterizzano l'ambiente complessivo: ambiente locale, ambiente globale e ambiente non locale.
- *Operazioni sull'ambiente*: le associazioni presenti nell'ambiente oltre che create e distrutte possono anche essere disattivate, riattivate e, ovviamente, usate.
- *Regole di scope*: le regole che, in ogni linguaggio, determinano la visibilità dei nomi.
- *Scope statico*: la regola di scope più usata nei linguaggi moderni.
- *Scope dinamico*: la regola di scope più semplice da implementare, usata oggi in pochi linguaggi.

In modo informale, possiamo dire che le regole che intervengono a definire l'ambiente sono costituite dalla regola di visibilità tra blocchi e dalla regola di

scope, che chiarisce come debba essere determinato l'ambiente non locale. In presenza di procedure, tali regole non sono ancora sufficienti a definire la nozione d'ambiente. Riprenderemo la questione nel Capitolo 9 (in particolare al termine del Paragrafo 9.2.1).

## 6.5 Nota bibliografica

I testi generali di linguaggi di programmazione, quali ad esempio [81], [88] e [89], trattano le problematiche viste in questo capitolo, anche se quasi sempre queste sono viste congiuntamente agli aspetti implementativi. Per ragioni di chiarezza espositiva, abbiamo scelto di considerare qui solo le regole semantiche relative alla gestione dei nomi e dell'ambiente, mentre considereremo nel prossimo capitolo la loro implementazione.

Per quanto quanto riguarda le regole dei singoli linguaggi si può poi fare riferimento agli specifici manuali, alcuni dei quali sono menzionati nella nota bibliografica del Capitolo 16, anche se a volte, come abbiamo discusso nell'Approfondimento 6.1, non tutti i dettagli vengono adeguatamente chiariti. La discussione dell'Approfondimento 6.1 trae materiale da [18].

## 6.6 Esercizi

I seguenti Esercizi 6-13, pur essendo incentrati su aspetti di scope, presuppongono la conoscenza dei meccanismi di passaggio dei parametri, che discuteremo nel Capitolo 9.

1. Si consideri il seguente frammento di programma scritto in uno pseudolinguaggio che usa scope statico e dove la primitiva `read(Y)` permette di leggere nella variabile `Y` un intero dall'input standard.

```
...
int X = 0;
int Y;
void pippo() {
    X++;
}
void pluto() {
    X++;
    pippo();
}
read(Y);
if Y > 0{int X = 5;
    pluto();}
else pluto();
write(X);
```

Si dica quali sono i valori stampati.

2. Si consideri il seguente frammento di programma scritto in uno pseudolinguaggio che usa scope dinamico.

```

...
int x;
x = 1;
int y;
void fie() {
    foo();
    x = 0;
}
void foo() {
    int x;
    x = 5;
}
read(y);
if y > 0{int x;
    x = 4;
    fie();}
else
    fie();
write(x);

```

Si dica quali sono (o qual è) i valori stampati.

3. Si consideri lo schema di codice seguente, nel quale vi sono due "buchi" indicati rispettivamente con (\*) e (\*\*). Si dia del codice da inserire al posto di (\*) e (\*\*) in modo tale che:
  - (a) se il linguaggio usato adotta scope statico, le due chiamate alla procedura `foo` assegnino a `x` lo stesso valore;
  - (b) se il linguaggio usato adotta scope dinamico, le due chiamate alla procedura `foo` assegnino a `x` valori diversi.

La funzione `foo` deve essere opportunamente dichiarata in (\*).

```

int i;
(*)
for (i=0; i<=1; i++) {
    int x;
    (**)
    x= foo();
}

```

4. Si fornisca un esempio di oggetto denotabile la cui vita sia più lunga di quella dei legami (nomi, puntatori ecc.) che vi si riferiscono.
5. Si fornisca un esempio di un legame fra un nome ed un oggetto denotabile la cui vita sia più lunga di quella dell'oggetto stesso.
6. Si dica cosa viene stampato dal seguente frammento di codice scritto in uno pseudolinguaggio che usa scope statico e dove i parametri sono passati per valore.

```

int x = 2;
int pippo(int y){
    x = x + y;
}
int x = 5;

```

```

    pippo(x);
    write(x);
}
write(x);
}

```

7. Si dica cosa stampa il frammento dell'esercizio precedente in regime di scope dinamico e passaggio dei parametri per riferimento.
8. Si dica cosa viene stampato dal seguente frammento di codice scritto in uno pseudolinguaggio che usa scope statico e passaggio di parametri per riferimento.

```

(int x = 2;
void pippo(reference int y){
    x = x + y;
    y = y + 1;
}
int x = 5;
int y = 5;
pippo(x);
write(x);
}
write(x);
}

```

9. Si dica cosa viene stampato dal seguente frammento di codice scritto in uno pseudolinguaggio che usa scope statico e passaggio di parametri per valore (un comando della forma `foo (w++)`; passa a `foo` il valore corrente di `w` e poi incrementa `w` di uno).

```

(int x = 2;
void pippo(value int y){
    x = x + y;
}
int x = 5;
pippo(x++);
write(x);
}
write(x);
}

```

10. Si dica cosa viene stampato dal seguente frammento di codice scritto in uno pseudolinguaggio che usa scope statico e passaggio per nome.

```

(int x = 2;
void pippo(name int y){
    x = x + y;
}
int x = 5;
    {int x = 7
    }
pippo(x++);
write(x);
}
write(x);
}

```

11. Si dica cosa viene stampato dal seguente frammento di codice scritto in uno pseudolinguaggio che usa scope dinamico e passaggio per riferimento.

```

int x = 1;
int y = 1;
void pippo(reference int z) {
    z = x + y + z;
}
int y = 3;
{int x = 3
}
pippo(y);
write(y);
}
write(y);
}

```

12. Si dica cosa viene stampato dal seguente frammento di codice scritto in uno pseudolinguaggio che usa scope statico e passaggio per riferimento.

```

int x = 0;
int A(reference int y) {
    int x = 2;
    y=y+1;
    return B(y)+x;
}
int B(reference int y){
    int C(reference int y) {
        int x = 3;
        return A(y)+x+y;
    }
    if (y==1) return C(x)+y;
    else return x+y;
}
write (A(x));
}

```

13. Si consideri il seguente frammento di codice in un linguaggio con scope statico e passaggio dei parametri sia per valore che per nome.

```

int z= 0;
int Omega() {
    return Omega();
}
int foo(int x, int y) {
    if (x==0) return x;
    else return x+y;
}
write(foo(z, Omega()+z));
}

```

- (i) Si dica qual è il risultato dell'esecuzione di tale frammento nel caso in cui i parametri di `foo` siano passati *per nome*.  
(ii) Si dica qual è il risultato dell'esecuzione di tale frammento nel caso in cui i parametri di `foo` siano passati *per valore*.

## La gestione della memoria

Una componente importante dell'interprete di una macchina astratta è quella che si occupa della gestione della memoria. Se in una macchina hardware tale componente può essere estremamente semplice, nel caso della macchina astratta di un linguaggio di alto livello la gestione della memoria è invece abbastanza complessa e usa varie tecniche, sia statiche che dinamiche. In questo capitolo analizzeremo nel dettaglio tali tecniche. Vedremo sia la gestione statica che quella dinamica, esaminando i record di attivazione, la pila di sistema, lo heap. Un paragrafo a parte è dedicato alle strutture dati ed ai meccanismi usati per l'implementazione delle regole di scope.

Fanno concettualmente parte delle tecniche di gestione della memoria anche le tecniche di garbage collection, cioè di recupero automatico della memoria allocata sullo heap. Per coerenza d'esposizione, tuttavia, questi argomenti saranno esposti nel Paragrafo 10.11, dopo aver trattato di tipi di dato e puntatori.

### 7.1 Tecniche di gestione della memoria

Come abbiamo visto nel Capitolo 1, la gestione della memoria costituisce una delle funzionalità dell'interprete associato ad una macchina astratta. Tale funzionalità gestisce l'allocazione di memoria per i programmi e per i dati, ossia stabilisce come questi debbano essere disposti in memoria, quanto tempo vi debbano rimanere e quali strutture dati ausiliarie siano necessarie per reperire le informazioni dalla memoria.

Nel caso di una macchina astratta di basso livello, quale ad esempio la macchina hardware, la gestione della memoria è molto semplice e può essere integralmente *statica*: prima dell'inizio dell'esecuzione il programma in linguaggio macchina ed i relativi dati vengono disposti in opportune zone di memoria, dove rimangono fino alla fine dell'esecuzione.

Nel caso di un linguaggio di alto livello le cose si fanno più complicate per vari motivi. Innanzitutto, se il linguaggio permette la ricorsione, l'allocazione sta-

tica non è più sufficiente<sup>1</sup>: difatti, mentre nel caso di linguaggi senza ricorsione possiamo stabilire staticamente il numero massimo di procedure attive in un qualsiasi momento dell'esecuzione (sicuramente tale numero è limitato dal numero di procedure dichiarate nel programma), nel caso di procedure ricorsive questo non è più vero, perché il numero di chiamate di procedura attive contemporaneamente può dipendere dai parametri delle procedure o comunque da informazioni disponibili solo a run-time.

**Esempio 7.1** Consideriamo la seguente funzione

```
int fib (int n) {
    if (n == 0) return 1;
    else if (n == 1) return 1;
    else return fib(n-1) + fib(n-2);
}
```

che, se chiamata con argomento  $n$ , calcola (in modo molto inefficiente) il valore dell' $n$ -simo numero di Fibonacci. Ricordiamo che i numeri di Fibonacci sono i termini della successione definita induttivamente come segue:  $\text{Fib}(0) = \text{Fib}(1) = 1$ ;  $\text{Fib}(n) = \text{Fib}(n - 1) + \text{Fib}(n - 2)$ , per  $n > 1$ <sup>2</sup>.

È evidente che il numero di chiamate di `fib` attive contemporaneamente dipende, oltre che dal punto dell'esecuzione, dal valore dell'argomento  $n$ . Usando una semplice relazione di ricorrenza si può verificare che il numero  $C(n)$  di chiamate di `fib` necessarie per calcolare il valore del termine  $\text{Fib}(n)$  (e, dunque, contemporaneamente attive) è esattamente eguale a tale valore: da una semplice ispezione del codice infatti si ha che  $C(n) = 1$  per  $n = 0$  e  $n = 1$ , mentre  $C(n) = C(n - 1) + C(n - 2)$  per  $n > 1$ . È noto che i numeri di Fibonacci crescono esponenzialmente, quindi il numero di chiamate di `fib` è dell'ordine di  $O(2^n)$ .

Dato che ogni chiamata di procedura richiede un proprio spazio di memoria per memorizzare parametri, risultati intermedi, indirizzi di ritorno ecc., quando si hanno procedure ricorsive l'allocazione statica di memoria non è più sufficiente e dobbiamo permettere operazioni di allocazione e deallocazione di memoria *dinamica*, effettuate cioè durante l'esecuzione del programma. Una tale gestione dinamica della memoria può essere realizzata in modo naturale usando una *pila*, in quanto le attivazioni di procedure (o di blocchi in-line) seguono una politica LIFO (Last In First Out): l'ultima procedura chiamata (o l'ultimo blocco nel quale siamo entrati) sarà la prima dalla quale usciremo.

Vi sono tuttavia altri casi che richiedono una gestione dinamica della memoria per i quali l'uso di una pila non è sufficiente. Sono i casi in cui il linguaggio

<sup>1</sup>Vedremo più avanti un'eccezione a questo principio generale nel caso della cosiddetta ricorsione in coda.

<sup>2</sup>La successione prende il nome dal matematico pisano omonimo, noto anche come Leonardo da Pisa (circa 1175-1250), che pare abbia avuto a che fare con questa successione studiando l'incremento di una popolazione di conigli. Per quanto riguarda le definizioni induttive si veda il riquadro a pagina 233.

permette operazioni di allocazione e deallocazione di memoria esplicite, come avviene ad esempio in C con i comandi `malloc` e `free`. In questi casi, dato che le operazioni di allocazione (`malloc`) e deallocazione (`free`) possono essere alternate in un ordine qualsiasi, non è possibile usare una pila per gestire la memoria e, come vedremo meglio nel seguito del capitolo, si usa una particolare struttura di memoria detta *heap*.

## 7.2 Gestione statica della memoria

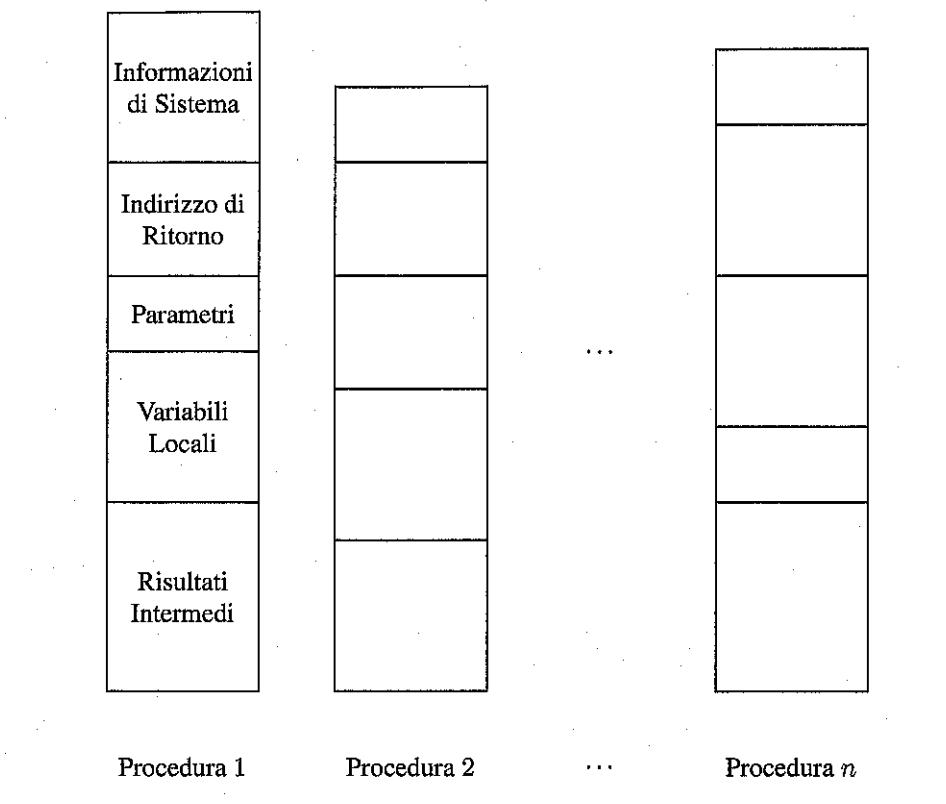
La memoria gestita *staticamente* è quella allocata dal compilatore, prima dell'esecuzione. Gli oggetti per i quali la memoria è allocata staticamente risiedono in una zona fissa di memoria (stabilita dal compilatore) per tutta la durata dell'esecuzione del programma. Tipici elementi per i quali è possibile allocare staticamente la memoria sono le *variabili globali*: queste infatti possono essere memorizzate in una zona di memoria fissata prima dell'esecuzione perché sono visibili in tutto il programma. Le *istruzioni del codice oggetto* prodotto dal compilatore possono essere considerate altri oggetti statici, dato che normalmente non cambiano durante l'esecuzione del programma; anche per esse la memoria sarà allocata dal compilatore. Le *costanti* sono altri elementi che possono essere gestiti staticamente (nel caso in cui i valori di queste non dipendano da altri valori non noti a tempo di compilazione). Infine varie *tabelle prodotte dal compilatore*, necessarie per il supporto a run-time del linguaggio (ad esempio per la gestione dei nomi, per il type checking, per il garbage collection) sono memorizzate in zone riservate allocate dal compilatore.

Nel caso in cui il linguaggio non supporti la ricorsione, come abbiamo anticipato, è possibile gestire staticamente anche la memoria per le rimanenti componenti del linguaggio: sostanzialmente si tratta di associare, staticamente, ad ogni procedura (o subroutine<sup>3</sup>) una zona di memoria nella quale memorizzare le informazioni locali della procedura stessa. Tali informazioni sono costituite dalle variabili locali, gli eventuali parametri della procedura (contenenti sia argomenti che risultati), l'indirizzo di ritorno (ossia l'indirizzo al quale deve tornare il controllo quando la procedura termina), eventuali valori temporanei usati in calcoli complessi e varie informazioni di "bookkeeping" (valori di registri salvati, informazioni per il debugging ed altro ancora).

La situazione di un linguaggio con sola allocazione statica della memoria è illustrata dalla Figura 7.1.

Si noti che chiamate successive della stessa procedura condividono la stessa zona di memoria, il che è corretto perché, in assenza di ricorsione, non possono esservi due diverse chiamate della stessa procedura attive contemporaneamente.

<sup>3</sup>Sarebbe più corretto parlare di subroutine perché questo era il termine usato nei linguaggi che usavano l'allocazione statica della memoria, quali, ad esempio, le prime versioni di FORTRAN degli anni 1960 e 1970.



**Figura 7.1** Gestione della memoria statica.

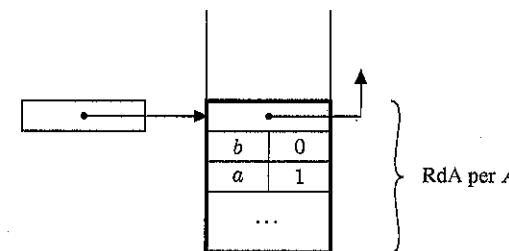
### 7.3 Gestione dinamica mediante pila

La maggior parte dei linguaggi di programmazione moderni permette una strutturazione a blocchi dei programmi<sup>4</sup>.

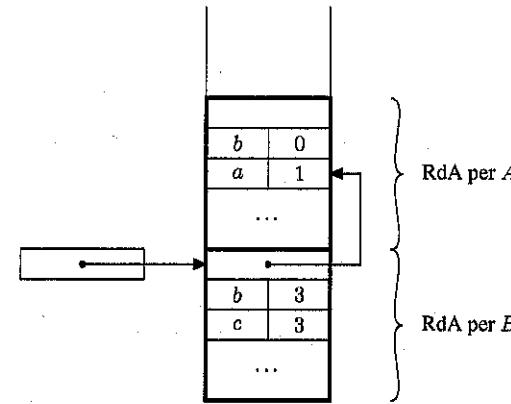
I blocchi, siano essi in-line oppure associati alle procedure, vengono aperti e chiusi usando la politica LIFO: quando si entra in un blocco A e poi in un blocco B, prima di uscire da A si deve uscire da B. È dunque naturale gestire lo spazio di memoria necessario per memorizzare le informazioni locali di ogni blocco usando una pila. Vediamo un esempio.

**Esempio 7.2** Consideriamo il seguente programma:

<sup>4</sup>Vedremo più avanti che linguaggi importanti (ad esempio C) tuttavia non offrono tutte le potenzialità di questo meccanismo, in quanto non permettono la dichiarazione di funzioni e procedure locali in blocchi annidati.



**Figura 7.2** Allocazione del record di attivazione per il blocco A nell'Esempio 7.2.

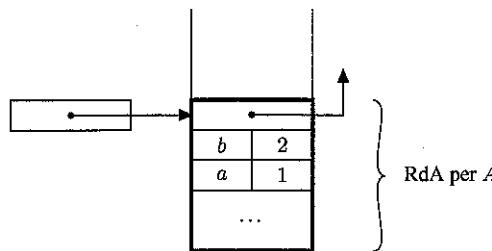


**Figura 7.3** Allocazione del record di attivazione per i blocchi A e B nell'Esempio 7.2.

```
A:{int a = 1;
  int b = 0;

B:{int c = 3;
  int b = 3;
  }
  b=a+1;
  }
```

Quando entriamo nel blocco A, a tempo di esecuzione, dobbiamo allocare sulla pila, con un'operazione di “push”, uno spazio opportuno per memorizzare le variabili a e b, come mostrato in Figura 7.2. All’entrata nel blocco B, dovremo allocare un nuovo spazio sulla pila per le variabili c e b (si ricordi che la variabile b interna è diversa da quella esterna) e dunque la situazione, dopo questa seconda allocazione, sarà quella descritta in Figura 7.3. All’uscita dal blocco B invece si dovrà effettuare un’operazione di “pop” per deallocare dalla pila lo spazio che



**Figura 7.4** Situazione dopo l'esecuzione dell'assegnamento nell'Esempio 7.2.

era stato riservato per il blocco. La situazione dopo tale deallocazione e dopo l'assegnamento è mostrata in Figura 7.4. Analogamente, all'uscita dal blocco A dovrà essere effettuata un'altra "pop" per deallocare anche lo spazio per A.

Il caso delle procedure è analogo e lo tratteremo nel Paragrafo 7.3.2.

Lo spazio di memoria, allocato sulla pila, dedicato ad un blocco in-line o ad un'attivazione di una procedura è detto *record di attivazione* (RdA) o anche *frame*. Si noti che il record di attivazione è associato ad una specifica attivazione di procedura (conseguente ad una chiamata della stessa) e non ad una dichiarazione di procedura: i valori che devono essere memorizzati nel RdA (variabili locali, valori temporanei ecc.) sono infatti diversi per le diverse chiamate della stessa procedura.

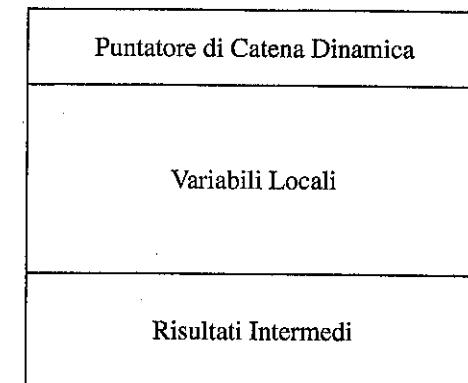
La pila sulla quale sono memorizzati i record di attivazione è detta pila a run-time (o pila di sistema).

Va infine notato che, per migliorare l'utilizzo della memoria a run-time, la gestione dinamica della memoria a volte è usata anche in implementazioni di linguaggi che non supportano la ricorsione. Se infatti il numero medio di chiamate di procedura attive contemporaneamente è minore del numero di procedure dichiarate nel programma, usando una pila si risparmierà spazio, in quanto non ci sarà bisogno di allocare una zona di memoria per ogni procedura dichiarata, come dovremmo invece fare con una gestione interamente statica.

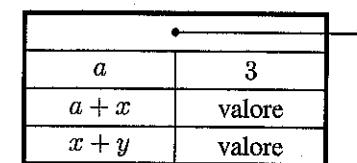
### 7.3.1 Record di attivazione per i blocchi in-line

La struttura di un generico record di attivazione per un blocco in-line è mostrata in Figura 7.5. I vari settori del record di attivazione contengono le seguenti informazioni:

**Risultati intermedi:** nel caso in cui si debbano effettuare dei calcoli può essere necessario memorizzare alcuni risultati intermedi, anche se ad essi il programma non assegna un nome esplicito. Ad esempio, il record di attivazione per il blocco



**Figura 7.5** Struttura di un record di attivazione per un blocco in-line.



**Figura 7.6** Un record di attivazione con lo spazio per i risultati intermedi.

```
int a = 3;
b= (a+x) / (x+y);
```

potrebbe avere la forma mostrata in Figura 7.6, dove i risultati intermedi ( $a+x$ ) e ( $x+y$ ) sono memorizzati esplicitamente prima che sia fatta la divisione. La necessità di memorizzare i risultati intermedi sulla pila dipende dal compilatore usato e, soprattutto, dall'architettura su cui si compila: su molte architetture essi saranno memorizzati nei registri.

**Variabili locali:** le variabili locali, dichiarate all'interno di un blocco, devono avere a disposizione uno spazio di memoria la cui dimensione dipenderà dal numero e dal tipo delle variabili. Queste informazioni in generale sono note al compilatore, che quindi potrà determinare la dimensione di questa parte del record di attivazione. In alcuni casi, tuttavia, vi possono essere dichiarazioni che dipendono da valori noti solo al momento dell'esecuzione (è questo ad esempio il caso degli array dinamici, presenti in alcuni linguaggi, le cui dimensioni dipendono da variabili che verranno istanziate solo al momento dell'esecuzione). In questi casi il record di attivazione prevede anche una parte di dimensione variabile che sarà definita al momento dell'esecuzione. Vedremo in dettaglio questo aspetto nel Capitolo 10 quando parleremo degli array.

**Puntatore di catena dinamica:** questo campo serve per memorizzare il puntatore al precedente record di attivazione sulla pila (ossia all'ultimo RdA creato in precedenza). Questa informazione è necessaria perché i RdA in generale hanno dimensioni diverse. Alcuni autori chiamano questo puntatore *link dinamico* o anche *link di controllo*. L'insieme dei collegamenti realizzati da questi puntatori è detto *catena dinamica*.

### 7.3.2 Record di attivazione per le procedure

Il caso delle procedure e delle funzioni<sup>5</sup> è analogo a quello dei blocchi in-line, con qualche ulteriore complicazione dovuta al fatto che, quando si attiva una procedura, occorre memorizzare una maggiore quantità di informazioni per gestire correttamente il controllo. La struttura di un generico record di attivazione per una procedura è quella mostrata in Figura 7.7. Si ricordi che una funzione, a differenza di una procedura, quando termina l'esecuzione restituisce un valore al chiamante. I record di attivazione per i due casi sono dunque eguali ad eccezione del fatto che, nel caso di una funzione, occorre tener conto anche della locazione di memoria nella quale la funzione deve memorizzare il valore restituito. Vediamo in dettaglio i vari campi.

**Risultati intermedi, variabili locali, puntatore di catena dinamica:** si veda quanto detto nel caso dei blocchi in-line.

**Puntatore di catena statica:** serve per gestire le informazioni necessarie a realizzare le regole di scope statico e sarà descritto nel Paragrafo 7.5.1.

**Indirizzo di ritorno:** contiene l'indirizzo della prima istruzione da eseguire dopo che la chiamata di procedura/funzione attuale ha terminato l'esecuzione.

**Indirizzo del risultato:** presente solo nel caso delle funzioni, contiene l'indirizzo della locazione di memoria nella quale il sottoprogramma deposita il valore restituito dalla funzione, quando questa termina. Si tratta di una locazione di memoria all'interno del RdA del chiamante.

**Parametri:** in questo spazio sono memorizzati i valori dei parametri attuali usati nella chiamata della procedura o funzione.

La disposizione dei vari campi nel record di attivazione varia a seconda delle diverse implementazioni. Il puntatore di catena dinamica e, in generale, ogni puntatore ad un RdA, punta ad una zona fissa (usualmente centrale) del record di attivazione. Gli indirizzi dei vari campi si ottengono quindi aggiungendo un offset negativo o positivo al valore del puntatore.

<sup>5</sup>Qui e nel seguito useremo quasi sempre i termini "funzione" e "procedura" come sinonimi. Sebbene non ci sia affatto accordo tra i vari autori, con procedura si dovrebbe indicare un sottoprogramma che non restituisce direttamente un valore; una funzione è un sottoprogramma che restituisce un valore.

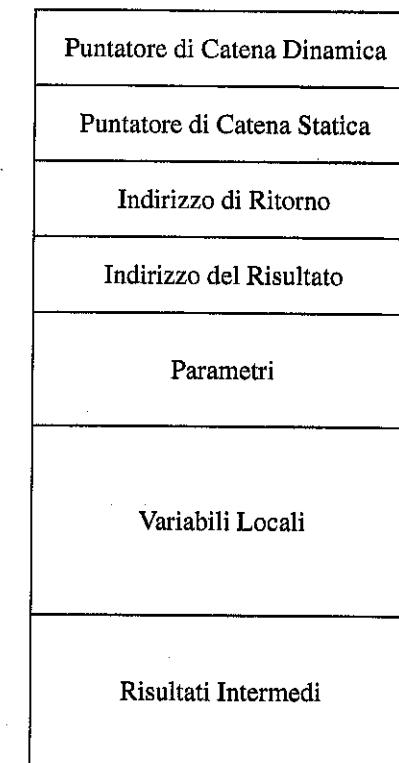
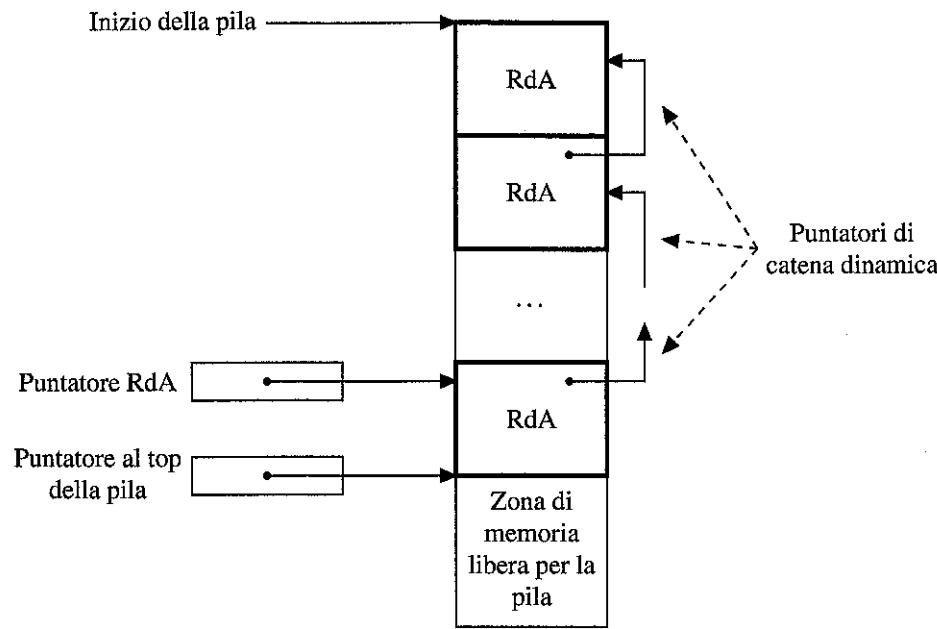


Figura 7.7 Struttura del record di attivazione per una procedura.

I nomi delle variabili normalmente non vengono memorizzati nel RdA e i riferimenti alle variabili locali sono sostituiti dal compilatore con un indirizzo relativo (o offset) rispetto ad una posizione fissa del RdA del blocco nel quale le variabili sono dichiarate. Questo è possibile perché la posizione di una dichiarazione di variabile all'interno di un blocco è fissata staticamente, e il compilatore può quindi associare ad ogni variabile locale una posizione precisa all'interno del record di attivazione.

Anche nel caso di riferimenti a variabili non locali, come vedremo quando parleremo delle regole di scope, è possibile usare opportuni accorgimenti per evitare la memorizzazione dei nomi e quindi per evitare di dover effettuare a runtime ricerche basate sul nome nella pila dei record di attivazione per risolvere un riferimento.

Infine, spesso i compilatori moderni ottimizzano il codice prodotto e salvano alcune informazioni nei registri invece che nel record di attivazione. Per semplicità in questo testo non considereremo tali ottimizzazioni e comunque, per maggiore chiarezza, negli esempi manterremo sempre il nome delle variabili nei



**Figura 7.8** La pila dei record di attivazione.

record di attivazione.

Notiamo, per concludere, che tutte le osservazioni che abbiamo fatto circa i nomi delle variabili, la loro accessibilità e memorizzazione nel RdA possono essere estese ad altre tipologie di oggetti denotabili.

### 7.3.3 Gestione della pila

La Figura 7.8 illustra la struttura di una pila di sistema, che assumiamo cresca verso il basso (la direzione di crescita della pila varia a seconda delle diverse implementazioni). Come si nota nella figura, un puntatore esterno alla pila indica l'ultimo RdA inserito nella pila stessa (puntando ad una zona prefissata del RdA, rispetto alla quale sono calcolati gli offset usati per gli accessi ai nomi locali). Questo puntatore, che noi chiamiamo puntatore al record di attivazione, è anche detto frame pointer o anche puntatore all'ambiente corrente (perché l'ambiente è realizzato dai RdA). Nella figura abbiamo indicato anche un altro puntatore, chiamato puntatore alla pila (o "stack pointer") che indica la prima posizione di memoria libera nella pila. Questo secondo puntatore, presente in alcune implementazioni, in linea di principio può anche essere omesso se il puntatore al RdA

punta sempre ad una posizione di distanza prefissata dall'inizio della parte libera della pila.

I record di attivazione vengono inseriti e rimossi dalla pila a tempo di esecuzione: quando si entra in un blocco, o si chiama una procedura, il relativo RdA verrà inserito nella pila, per poi essere eliminato dalla stessa quando si esce dal blocco o quando termina l'esecuzione della procedura.

La gestione a run-time della pila di sistema è realizzata da alcuni frammenti di codice che il compilatore (o l'interprete) inserisce immediatamente prima e dopo la chiamata di una procedura oppure prima dell'inizio e dopo la fine di un blocco.

Vediamo nel dettaglio cosa accade nel caso delle procedure, dato che il caso dei blocchi in-line è solo una semplificazione di questo.

Innanzitutto chiariamo la terminologia che usiamo: "chiamante" e "chiamato" indicano, rispettivamente, il programma o la procedura che effettua una chiamata (di procedura) e la procedura che è stata chiamata.

La gestione della pila è fatta sia dal chiamante che dal chiamato. Per questo scopo, e anche per gestire altre informazioni di controllo, nel chiamante viene aggiunta una parte di codice detta *sequenza di chiamata* che è eseguita in parte immediatamente prima della chiamata di procedura, e in parte immediatamente dopo la terminazione della procedura che è stata chiamata. Nel chiamato invece viene aggiunto un *prologo*, da eseguirsi subito dopo la chiamata, ed un *epilogo*, da eseguirsi al termine dell'esecuzione della procedura. Questi tre frammenti di codice si dividono le varie operazioni necessarie a gestire i record di attivazione e ad implementare correttamente una chiamata di procedura. La divisione esatta tra che cosa fa il chiamante e cosa il chiamato dipende, al solito, dal compilatore e dalla specifica implementazione considerata. Tuttavia, per ottimizzare la dimensione del codice prodotto è preferibile che la maggior parte delle attività sia delegata al chiamato, visto che in questo caso il codice viene aggiunto una sola volta (al codice relativo alla dichiarazione del chiamato) invece che molte (al codice relativo alle varie chiamate). Senza dunque specificare ulteriormente la divisione dei compiti, al momento della *chiamata di procedura* la sequenza di chiamata ed il prologo si devono occupare delle seguenti attività.

**Modifica del valore del contatore programma:** questo evidentemente è necessario per passare il controllo alla procedura chiamata. Il vecchio valore (incrementato) dovrà essere salvato per mantenere l'indirizzo di ritorno.

**Allocazione dello spazio sulla pila:** ossia deve essere predisposto lo spazio per il nuovo RdA e quindi modificare di conseguenza il puntatore alla prima posizione libera della pila.

**Modifica del puntatore al RdA:** il puntatore dovrà indicare il nuovo RdA, relativo alla procedura chiamata, che è stato inserito sulla pila.

**Passaggio dei parametri:** questa attività usualmente è compito del chiamante, visto che chiamate diverse della stessa procedura possono avere parametri attuali diversi.

**Salvataggio dei registri:** i valori per la gestione del controllo, tipicamente memorizzati nei registri, devono essere salvati. Questo è il caso ad esempio del vecchio puntatore al RdA, che viene salvato come puntatore di catena dinamica.

**Esecuzione del codice per l'inizializzazione:** alcuni linguaggi prevedono costrutti esplicativi per inizializzare alcuni elementi memorizzati nel nuovo record di attivazione.

Al momento del *ritorno del controllo al programma chiamante*, quando la procedura chiamata termina la sua esecuzione, *l'epilogo* (nel chiamato) e la *sequenza di chiamata* (nel chiamante) devono invece gestire le seguenti operazioni.

**Ripristino del valore del contatore programma:** questo è necessario per restituire il controllo al chiamante.

**Restituzione dei valori:** i valori dei parametri che permettono di passare informazioni dal chiamato al chiamante, oppure il valore calcolato dalla funzione, devono essere memorizzati in opportune locazioni, di solito presenti nel RdA del chiamante, reperibili a partire dal RdA del chiamato.

**Ripristino dei registri:** i valori dei registri precedentemente salvati devono essere ripristinati. In particolare deve essere ripristinato il vecchio valore del puntatore al RdA.

**Esecuzione del codice per la finalizzazione:** alcuni linguaggi prevedono l'esecuzione di un opportuno codice di finalizzazione prima che alcuni oggetti locali siano distrutti.

**Deallocazione dello spazio sulla pila:** il RdA della procedura che è terminata deve essere rimosso dalla pila. Il puntatore alla (prima posizione libera della) pila deve essere modificato di conseguenza.

Si noti che nella descrizione precedente abbiamo omesso la gestione delle strutture dati necessarie alla realizzazione delle regole di scope. Questa verrà esaminata in dettaglio nel Paragrafo 7.5 di questo capitolo.

## 7.4 Gestione dinamica mediante heap

Nel caso in cui il linguaggio includa comandi esplicativi di allocazione della memoria, come ad esempio avviene in C e in Pascal, la sola gestione mediante pila non è sufficiente. Consideriamo ad esempio il seguente frammento di codice C:

```
int *p, *q; /* p, q puntatori NULL a interi */
p = malloc (sizeof (int));
/* alloca la memoria puntata da p */
q = malloc (sizeof (int));
/* alloca la memoria puntata da q */
*p = 0; /* dereferenzia e assegna */
*q = 1; /* dereferenzia e assegna */
free(p); /* dealloca la memoria puntata da p */
free(q); /* dealloca la memoria puntata da q */
```

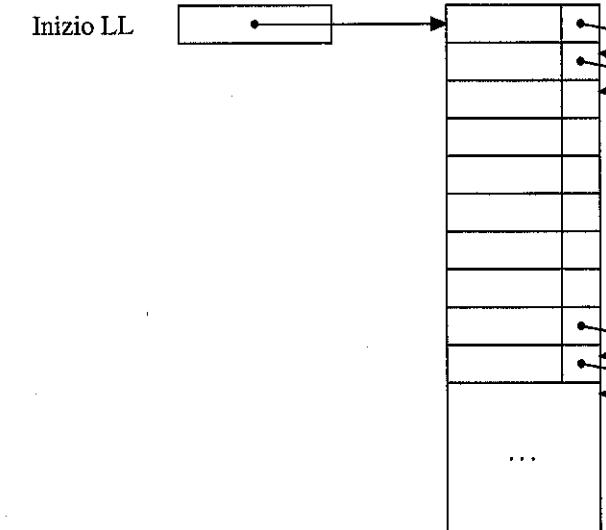


Figura 7.9 Lista libera per heap con blocchi di dimensione fissa.

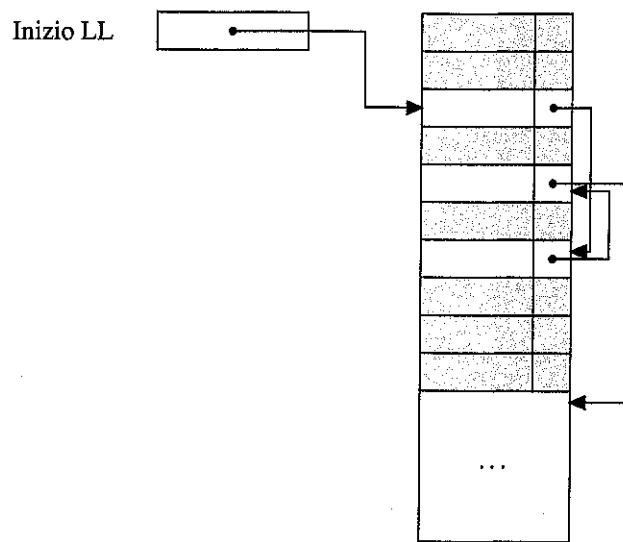
Dato che le operazioni di deallocazione della memoria sono fatte nello stesso ordine delle operazioni di allocazione (prima *p* e poi *q*) la memoria da allocare non può essere gestita in modo LIFO.

Per gestire allocazioni esplicative di memoria, che possono avvenire in momenti di tempo arbitrari, si usa una particolare zona di memoria, detta *heap*. Si noti che questo termine è usato in informatica anche per indicare un particolare tipo di struttura dati, rappresentabile mediante un albero binario oppure mediante un vettore, usata per implementare efficientemente code con priorità (e usato anche nell'algoritmo di ordinamento "heapsort", dove fu originariamente introdotto il termine "heap"). La definizione di *heap* che usiamo qui non ha niente a che vedere con questa struttura dati: nel gergo dei linguaggi di programmazione un *heap* è semplicemente una zona di memoria nella quale blocchi di memoria possono essere allocati e deallocati in modo relativamente libero.

I metodi di gestione dello *heap* si dividono in due categorie principali, a seconda che i blocchi di memoria sia considerati di *dimensione fissa* oppure di *dimensione variabile*.

### 7.4.1 Blocchi di dimensione fissa

In questo caso lo *heap* è diviso in un certo numero di elementi, o blocchi, di dimensione fissa abbastanza limitata, collegati in una struttura a lista, detta *lista libera* e mostrata in Figura 7.9. Quando, a run-time, un'operazione richiede l'allocazione di un blocco di memoria sullo *heap* (ad esempio, con il comando *malloc*),



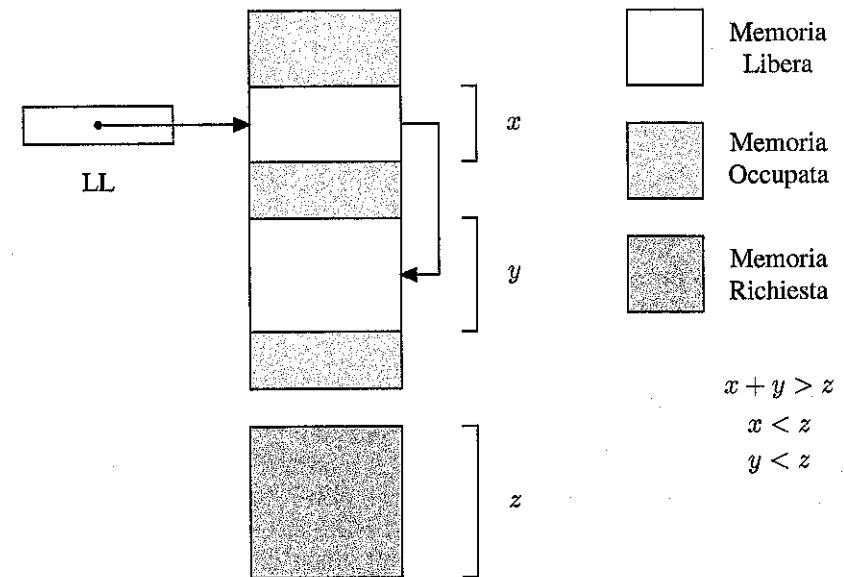
**Figura 7.10** Lista libera per heap con blocchi di dimensione fissa dopo alcune allocazioni di memoria. I blocchi grigi sono allocati (in uso).

il primo elemento della lista libera viene rimosso dalla lista, il puntatore a tale elemento è restituito all'operazione che aveva richiesto la memoria ed il puntatore alla lista libera è aggiornato in modo tale da puntare all'elemento successivo. Quando invece viene liberata, o deallocated, della memoria (ad esempio con una `free`), il blocco liberato viene collegato nuovamente alla testa della lista libera. La situazione dopo alcune allocazioni di memoria è mostrata in Figura 7.10. Concettualmente, dunque, la gestione dello heap con blocchi di dimensione fissa è semplice, a patto che si sappia identificare e recuperare facilmente la memoria che deve essere restituita alla lista libera. Queste operazioni di identificazione e recupero non sono ovvie, come vedremo più avanti.

#### 7.4.2 Blocchi di dimensione variabile

Nel caso in cui il linguaggio permetta l'allocazione a run-time di spazi di memoria di dimensione variabile, ad esempio per memorizzare un *array* di dimensione variabile, i blocchi di dimensione fissa non sono più adeguati dato che la memoria che dobbiamo allocare può avere dimensione maggiore di quella, prefissata, del blocco, e non possiamo usare più blocchi diversi per memorizzare una struttura dati come l'*array*, che richiede una porzione sequenziale di memoria. In questi casi si usa una gestione dello heap con blocchi di *dimensione variabile*.

Questo secondo tipo di gestione usa tecniche diverse, definite principalmente con lo scopo di migliorare l'occupazione di memoria e la velocità di esecuzio-



**Figura 7.11** Frammentazione esterna.

ne per le operazioni di gestione dello heap (si ricordi che queste sono effettuate a run-time e quindi incidono sui tempi d'esecuzione dei programmi). Al solito, queste due caratteristiche sono difficilmente conciliabili e le buone implementazioni tendono ad un ragionevole compromesso.

In particolare, riguardo all'occupazione di memoria, si cerca di evitare i fenomeni di *frammentazione* della memoria. La cosiddetta *frammentazione interna* si verifica quando si aloca un blocco di dimensione strettamente maggiore di quella richiesta dal programma: la porzione di memoria non utilizzata, interna al blocco, evidentemente andrà sprecata fino a che il blocco non sarà restituito alla lista libera. Ma non è questo il problema più serio. Infatti è ben peggiore la cosiddetta *frammentazione esterna*, che si verifica quando la lista libera è composta di blocchi di dimensione (relativamente) piccola per cui, anche se la somma della memoria libera totale presente è sufficiente, non si riesce ad usare effettivamente la memoria libera. La Figura 7.11 mostra un esempio di questa situazione: se abbiamo nella lista libera due blocchi di dimensione  $x$  e  $y$  (parole o altro, non ha importanza in questo contesto) e richiediamo l'allocazione di un blocco di dimensione superiore, la nostra richiesta non potrà essere soddisfatta, nonostante la somma totale della memoria libera sia superiore alla memoria richiesta. Le tecniche di allocazione di memoria tendono quindi a "ricompattare" la memoria libera, unendo blocchi liberi contigui, in modo tale da evitare la frammentazione esterna. Per ottenere questo obiettivo possono essere richieste operazioni aggiuntive che appesantiscono la gestione e quindi riducono l'efficienza.

**Unica lista libera** La prima tecnica che esaminiamo considera un'unica lista libera, costituita inizialmente da un unico blocco di memoria, contenente l'intero heap. È infatti conveniente cercare di mantenere i blocchi della massima dimensione possibile: non ha dunque senso dividere inizialmente lo heap in tanti piccoli blocchi, come invece facevamo nel caso dei blocchi di dimensione fissa. Quando viene richiesta l'allocazione di un blocco di  $n$  parole di memoria, le prime  $n$  parole sono allocate e il puntatore all'inizio dello heap avanza di  $n$ . Analogamente per le richieste successive, mentre i blocchi di memoria deallocated vengono collegati in una lista libera. Quando si raggiunge la fine dello spazio di memoria dedicato allo heap si dovrà passare ad utilizzare lo spazio di memoria deallocated, e questo può essere fatto nei due modi seguenti.

(i) **Utilizzo diretto della lista libera:** in questo caso viene mantenuta una lista libera di blocchi di dimensione variabile. Quando si richiede l'allocazione di memoria per un blocco di  $n$  parole viene cercato nella lista libera un blocco di dimensione  $k$  maggiore o eguale a  $n$  e all'interno di questo viene allocata la memoria richiesta. La porzione del blocco non usata (di dimensione  $k - n$ ), se superiore ad una soglia prefissata va a costituire un nuovo blocco da inserire nella lista libera (sotto tale soglia, è ammisible la frammentazione interna). La ricerca del blocco di dimensione adeguata può avvenire secondo due politiche: con la *first fit* si cerca nella lista il primo blocco di dimensione sufficiente, mentre con la *best fit* si cerca quello di dimensione minima fra tutti quelli di dimensione sufficiente. La prima tecnica privilegia il tempo di gestione, mentre la seconda l'occupazione di memoria. Per entrambe comunque il costo di allocazione è lineare rispetto al numero di blocchi presenti nella lista libera. Se si mantengono ordinati i blocchi per dimensione crescente, le due politiche coincidono, dato che in entrambi i casi si tratta di scorrere la lista fino a quando si trova un blocco di dimensione sufficiente. Tuttavia in questo caso il costo dell'inserimento di un blocco nella lista libera aumenta (da costante a lineare), dato che si deve trovare la posizione giusta per inserire un blocco. Infine quando un blocco deallocated viene restituito alla lista libera, per ridurre la frammentazione esterna si controlla se i blocchi di memoria fisicamente adiacenti sono liberi, nel qual caso si ricompattano in un unico blocco. Questo tipo di compattazione è detto *parziale* in quanto si compattano solo i blocchi liberi adiacenti.

(ii) **Compattazione della memoria libera:** secondo questa tecnica, quando si raggiunge la fine dello spazio inizialmente dedicato allo heap si spostano tutti i blocchi ancora attivi, ossia tutti quelli che non sono stati restituiti alla lista libera, ad un'estremità dello heap, lasciando così tutta la memoria libera in un unico blocco contiguo. A questo punto si aggiorna il puntatore allo heap con l'inizio dell'unico blocco di memoria libera e si riprende l'allocazione come prima. Ovviamente perché questa tecnica sia possibile i blocchi di memoria allocata devono essere spostabili, cosa non sempre garantita (si pensi a blocchi i cui indirizzi sono stati memorizzati in puntatori allocati sulla pila); alcune tecniche di compattamento saranno discusse nel Paragrafo 10.11, quando tratteremo del garbage collection.

**Liste libere multiple** Per ridurre il costo di allocazione di un blocco alcuni metodi di gestione dello heap usano più liste libere per blocchi di dimensione diversa. Quando viene richiesto un blocco di dimensione  $n$  viene selezionata la lista che contiene blocchi di dimensione maggiore o eguale a  $n$  e quindi viene allocato un blocco da tale lista (con una qualche frammentazione interna se il blocco ha dimensione maggiore di  $n$ ). Le dimensioni dei blocchi anche in questo caso possono essere statiche o dinamiche e, nel caso di dimensioni dinamiche, due metodi di gestione comunemente usati sono il *buddy system* e gli *heap di Fibonacci*. Nel primo le dimensioni dei blocchi delle varie liste libere sono potenze di due. Se è richiesto un blocco di dimensione  $n$  e  $k$  è il più piccolo intero tale che  $2^k \geq n$ , allora si cerca un blocco di dimensione  $2^k$  (nella opportuna lista libera). Se un tale blocco libero viene trovato questo viene allocato, altrimenti si cerca nella successiva lista libera un blocco (libero) di dimensione  $2^{k+1}$  e si divide in due. Uno dei due blocchi (che quindi hanno dimensione  $2^k$ ) viene allocato, mentre l'altro viene inserito nella lista libera dei blocchi di dimensione  $2^k$ . Quando un blocco risultante da una divisione viene restituito alla lista libera si cerca il suo "compagno" (*buddy* in inglese), ossia l'altra metà risultante dalla divisione, e se questo è libero i due blocchi vengono riuniti per ricostituire il blocco iniziale di dimensione  $2^{k+1}$ . Il metodo del *Fibonacci heap* è simile ma usa i numeri di Fibonacci, invece che le potenze di 2, come dimensioni dei vari blocchi. Dato che la successione di Fibonacci cresce più lentamente della successione  $2^n$ , con questo secondo metodo si ha una minore frammentazione interna.

## 7.5 Implementazione delle regole di scope

La possibilità di denotare oggetti, anche complessi, mediante nomi con opportune regole di visibilità costituisce uno degli aspetti più importanti che differenziano un linguaggio di alto livello da uno di basso livello. La realizzazione dell'ambiente e l'implementazione delle regole di scope viste nel Capitolo 6 richiedono dunque opportune strutture dati aggiuntive. In questo paragrafo analizziamo tali strutture e la loro gestione.

Dato che il record di attivazione contiene lo spazio di memoria per i nomi locali, quando si incontra un riferimento ad un nome non locale si dovranno esaminare i vari record di attivazione ancora attivi (ossia presenti sulla pila) per trovare quello corrispondente al blocco dove il nome in questione è stato dichiarato: sarà tale blocco infatti a contenere l'associazione per il nostro nome. L'ordine con cui esaminare i record di attivazione varia a seconda del tipo di scope considerato.

### 7.5.1 Scope statico: la catena statica

Se si usa la regola dello scope statico, come abbiamo anticipato nel Capitolo 6, l'ordine con cui esaminare i record di attivazione per risolvere un riferimento non locale non è quello definito dalla posizione degli stessi nella pila. In altri termini, il RdA direttamente collegato dal puntatore di catena dinamica non è necessariamente il primo RdA nel quale cercare di risolvere un riferimento non

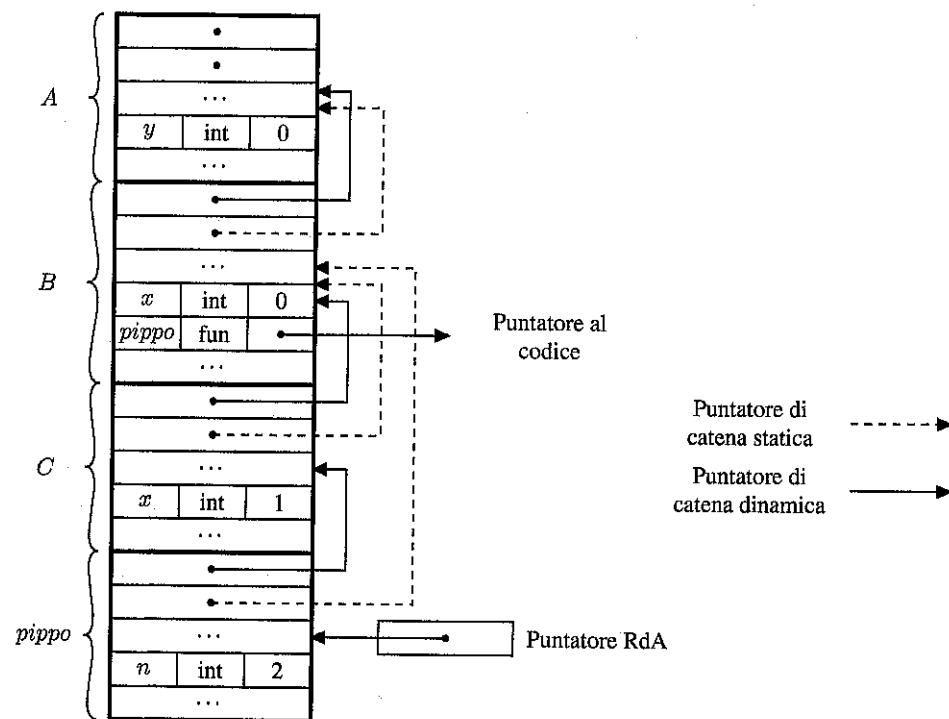


Figura 7.12 Pila dei RdA con catena statica, per l'Esempio 7.3.

locale, ma tale primo RdA sarà definito dalla struttura testuale del programma. Vediamo un esempio.

**Esempio 7.3** Consideriamo il seguente codice dove, al solito, per comodità di riferimento etichettiamo i blocchi:

```
A:{int y=0;
B:{int x = 0;
void pippo(int n) {
    x = n+1;
    y = n+2;
}
C:{int x = 1;
    pippo(2);
    write(x);
}
write(y);
}
```

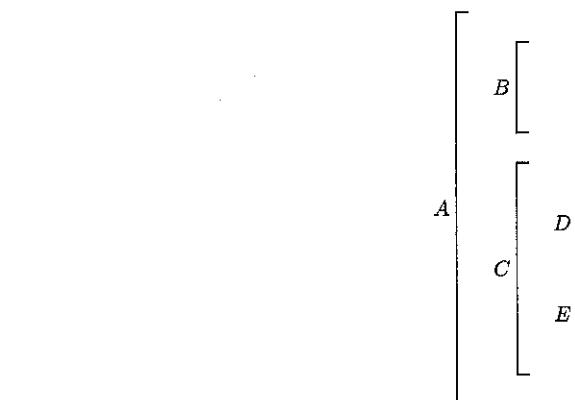
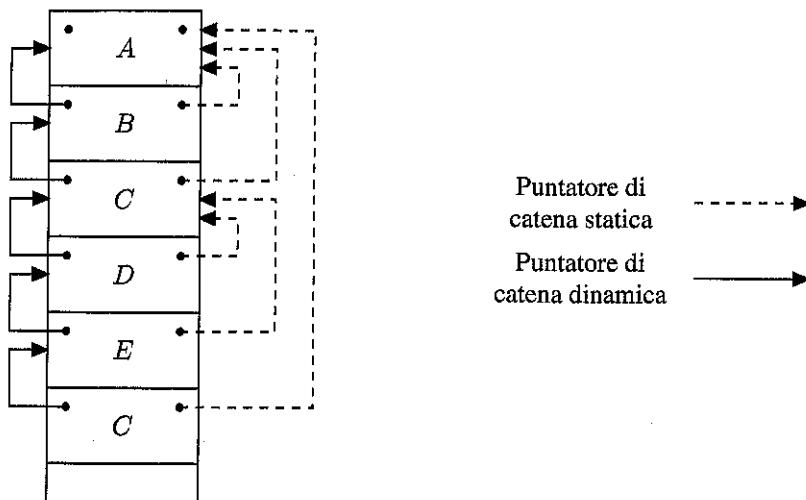


Figura 7.13 Una struttura a blocchi.

Dopo aver eseguito la chiamata `pippo(2)` la situazione della pila dei record di attivazione è quella descritta nella Figura 7.12. Il primo record di attivazione sulla pila (quello più in alto) è relativo al blocco esterno; il secondo è quello per il blocco B, il terzo è per C ed infine il quarto è il record di attivazione per la chiamata di procedura. La variabile non locale `x` usata nella procedura `pippo`, come sappiamo dalla regola di scope statico, non è quella dichiarata nel blocco C ma è quella dichiarata nel blocco B. Per poter reperire correttamente questa informazione a tempo di esecuzione, il record di attivazione della chiamata di procedura è collegato da un opportuno puntatore, detto puntatore di *catena statica* e disegnato tratteggiato in Figura 7.12, al record del blocco contenente la dichiarazione della variabile. Tale record è collegato, a sua volta, da un puntatore di catena statica al record del blocco A in quanto questo blocco, essendo il primo immediatamente esterno a B, è il primo blocco da considerare per risolvere i riferimenti non locali di B. Quando all'interno della chiamata di procedura `pippo` si usano le variabili `x` e `y`, per reperire la zona di memoria dove queste sono memorizzate si seguono i puntatori di catena statica a partire dal record di attivazione di `pippo` arrivando così al RdA di B, per la `x`, e a quello di A, per la `y`.

Generalizzando dal precedente esempio possiamo dire che, per gestire a runtime la regola di scope statico, il record di attivazione del generico blocco B è collegato dal *puntatore di catena statica* al record del blocco immediatamente esterno a B (ossia al più vicino blocco esterno che contiene B). Si noti che nel caso in cui B sia il blocco di una chiamata procedura, il blocco immediatamente esterno a B è quello che contiene la dichiarazione della procedura stessa. Inoltre se B è attivo, ossia se il suo RdA è sulla pila, allora anche i blocchi esterni a B che lo contengono devono essere attivi, e quindi si trovano sulla pila.

Oltre alla *catena dinamica*, costituita dai vari record presenti sulla pila di sistema (collegati nell'ordine dato dalla pila stessa), esiste dunque anche una *catena*



**Figura 7.14** Catena statica per la struttura precedente e la sequenza di chiamate A, B, C, D, E, C.

statica, costituita dai vari puntatori di catena statica e usata per rappresentare la struttura statica di annidamento dei blocchi nel programma.

Come esempio si consideri la Figura 7.13 che mostra una generica struttura a blocchi risultante da procedure annidate; si consideri ora la sequenza di chiamate A, B, C, D, E, C, dove si intende che ogni chiamata rimane attiva quando viene invocata la chiamata successiva. La situazione della pila dei RdA, con i relativi puntatori di catena statica, dopo una tale sequenza di chiamate è quella mostrata in Figura 7.14.

La gestione della catena statica a run-time rientra fra le funzioni svolte dalla sequenza di chiamata, dal prologo e dall'epilogo visti in precedenza. Anche per la gestione della catena statica chiamante e chiamato possono dividere in vario modo le operazioni da svolgere. Secondo l'approccio più comune al momento in cui si entra in un nuovo blocco il chiamante calcola il puntatore di catena statica del chiamato e quindi passa tale puntatore al chiamato. Tale calcolo è abbastanza semplice e può essere facilmente compreso dividendo due casi:

**Il chiamato si trova all'esterno del chiamante.** In questo caso, in base alle regole di visibilità definite dallo scope statico, perché il chiamato sia visibile si deve trovare in un blocco esterno, che includa il blocco del chiamante. Dunque il RdA di tale blocco esterno si deve trovare già sulla pila. Supponiamo che fra il chiamato ed il chiamante vi siano  $k$  livelli di annidamento nella struttura a blocchi del programma: se il chiamante si trova a livello di annidamento  $n$  ed il chiamato si trova a livello  $m$  supponiamo dunque che si abbia  $k = n - m$ . Questo valore di  $k$  è determinabile dal compilatore, dato che dipende unica-

mente dalla struttura statica del programma e dunque può essere associato alla chiamata in questione. Il chiamante può quindi calcolare il puntatore di catena statica del chiamato semplicemente dereferenziando  $k$  volte il proprio puntatore di catena statica (ossia, seguendo per  $k$  passi la propria catena statica).

**Il chiamato si trova all'interno del chiamante.** In questo caso le regole di visibilità assicurano che il chiamato è dichiarato immediatamente nel blocco in cui avviene la chiamata e quindi il primo blocco esterno al chiamato è proprio quello del chiamante: il puntatore di catena statica del chiamato dovrà "puntare" al RdA del chiamante. Il chiamante può semplicemente passare al chiamato, come puntatore di catena statica, il puntatore al proprio record di attivazione.

Una volta che il chiamato ha ricevuto il puntatore di catena statica deve soltanto memorizzarlo in un'apposita area del proprio record di attivazione, operazione che sarà fatta dal prologo. Al momento in cui si esce da un blocco la catena statica non richiede particolari operazioni di gestione.

Abbiamo accennato al fatto che il compilatore, per permettere la gestione della catena statica a run-time, tiene traccia del livello di annidamento delle varie chiamate di procedura. Questo è fatto usando la *tabella dei simboli*, una sorta di dizionario dove, più in generale, il compilatore memorizza i nomi usati nel programma e tutte le informazioni necessarie per gestire gli oggetti denotati da tali nomi (ad esempio per conoscerne il tipo) e per realizzare le regole di visibilità. In particolare vengono identificati e numerati i vari scope in base al livello di annidamento e quindi ad ogni riferimento ad un nome viene associato anche un numero che indica lo scope che contiene la dichiarazione per tale nome. In base a tale numero possiamo calcolare, a tempo di compilazione, la distanza fra lo scope della chiamata e quello della dichiarazione, necessaria a run-time per gestire la catena statica.

Si noti che questa distanza calcolata staticamente permette anche di risolvere a run-time riferimenti non locali senza dover effettuare alcuna ricerca (in base al nome) nei RdA che si trovano sulla pila. Infatti, se usiamo un riferimento al nome non locale  $x$ , per trovare il RdA che contiene lo spazio di memoria per  $x$  basterà partire dal RdA corrispondente al blocco che contiene il riferimento e scorrere la catena statica di un numero di link pari al valore di tale distanza. All'interno del RdA così trovato la posizione di memoria per  $x$  è anch'essa fissata dal compilatore e quindi a run-time non serve alcuna ricerca, ma solo l'offset statico di  $x$  rispetto al puntatore al RdA.

È evidente che, comunque, il compilatore non può risolvere completamente in modo statico un riferimento ad un nome non locale ed occorre sempre seguire, a run-time, i link di catena statica. Questo perché in generale non è possibile conoscere staticamente il numero di RdA presenti sulla pila.

Come esempio concreto di quanto appena detto, si consideri il codice dell'Esempio 7.3. Il compilatore "sa" che dall'uso della variabile  $y$  nella procedura  $pippo$  occorre passare due blocchi esterni ( $B$  e  $A$ ) per arrivare a quello contenente la dichiarazione della variabile. Basta dunque memorizzare tale valore a tempo di compilazione per poi sapere, a run-time, che per risolvere il nome  $y$  si devono percorrere due puntatori di catena statica. Non occorre memorizzare il nome

y esplicitamente perché la sua posizione all'interno del record di attivazione del blocco A è fissata dal compilatore. Analogamente, le informazioni di tipo che noi, per chiarezza, abbiamo riportato nella Figura 7.12, fanno parte della tabella dei simboli e, fatti gli opportuni controlli a tempo di compilazione, possono in larga misura scomparire a run-time.

### 7.5.2 Scope statico: il display

La realizzazione dello scope statico mediante catena statica ha un inconveniente: se dobbiamo usare un nome non locale, dichiarato in un blocco esterno di  $k$  livelli rispetto al punto in cui ci troviamo, a tempo di esecuzione dobbiamo effettuare  $k$  accessi in memoria, necessari per scorrere la catena statica, per determinare il RdA che contiene la locazione di memoria per il nome che ci interessa. Questo problema non è poi così drammatico, dato che raramente nei programmi reali si superano i tre livelli di annidamento di blocchi e procedure. La tecnica del *display* permette comunque di ridurre il numero degli accessi ad una costante (due).

Questa tecnica usa un vettore, detto *display*, contenente tanti elementi quanti sono i livelli di annidamento dei blocchi presenti nel programma, dove l'elemento  $k$ -esimo del vettore contiene il puntatore al RdA di livello di annidamento  $k$  correntemente attivo. Quando ci si riferisce ad un oggetto non locale, dichiarato in un blocco esterno di livello  $k$ , il RdA contenente tale oggetto può essere reperito semplicemente accedendo alla posizione  $k$  del vettore e seguendo il puntatore presente in tale posizione.

La gestione del *display* è molto semplice, anche se è leggermente più costosa di quella della catena statica in quanto, quando si entra o esce da un ambiente, oltre ad aggiornare il puntatore presente nel vettore si deve anche salvare il vecchio valore. Più precisamente, quando viene chiamata una procedura (o si entra in un blocco in-line) di livello  $k$ , la posizione  $k$  del *display* dovrà essere aggiornata con il valore del puntatore al RdA del chiamato, in quanto questo è diventato il nuovo blocco attivo di livello  $k$ . Prima di questo aggiornamento, però, deve essere salvato il precedente contenuto della posizione  $k$  del *display*, normalmente memorizzandolo nel RdA del chiamato.

La necessità di salvare il vecchio valore del *display* può essere meglio compresa esaminando i due casi possibili:

**Il chiamato si trova all'esterno del chiamante.** Supponiamo che il chiamante si trovi a livello di annidamento  $n$  ed il chiamato si trovi a livello  $m$ , con  $m < n$ . Il chiamato ed il chiamante condividono dunque la struttura statica fino al livello  $m - 1$ , e quindi il *display* fino alla posizione  $m - 1$ . L'elemento  $m$  del *display* è aggiornato con il puntatore al RdA del chiamato e fino a quando il chiamato non termina la propria esecuzione il *display* attivo è quello costituito dai primi  $m$  elementi. Il vecchio valore contenuto nella posizione  $m$  deve essere salvato, perché questo punta al RdA del blocco che tornerà ad essere attivo quando il chiamato terminerà la propria esecuzione e quindi il *display* tornerà ad essere quello esistente prima della chiamata.

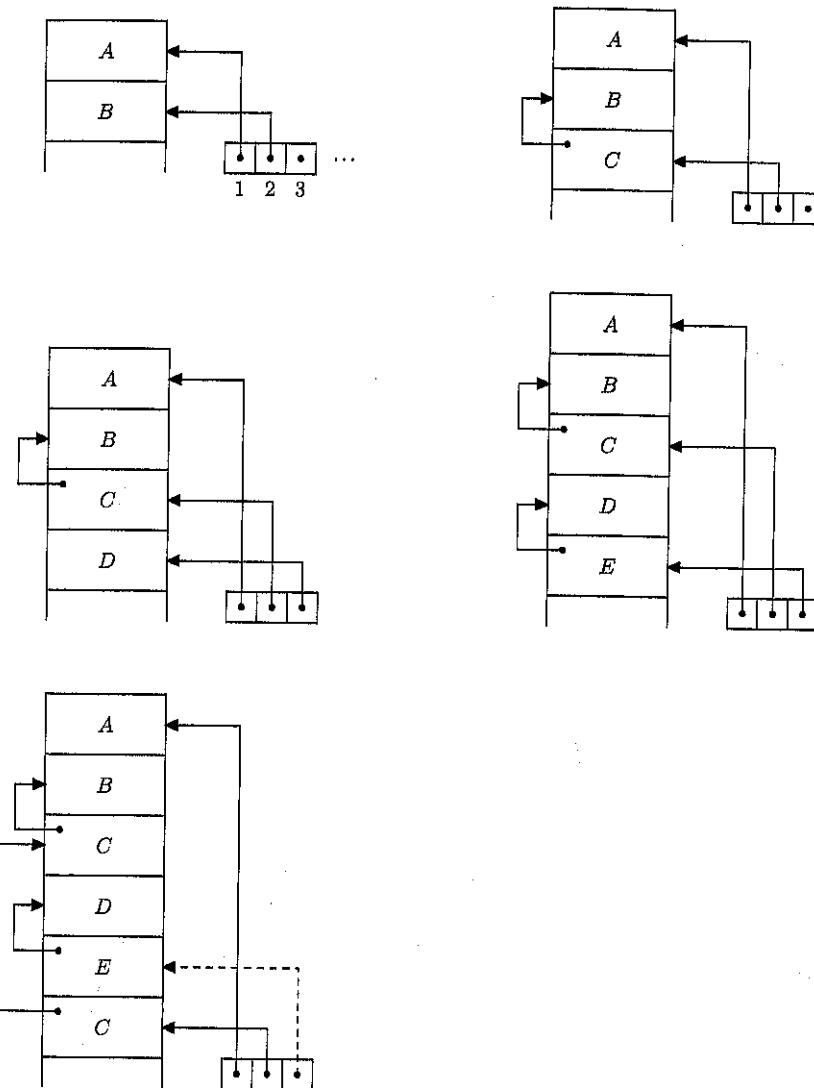
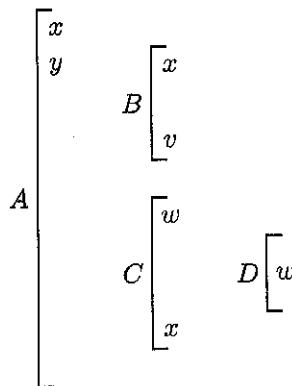


Figura 7.15 Display per la struttura di Figura 7.13 e la sequenza di chiamate A, B, C, D, E, C.



**Figura 7.16** Una struttura a blocchi con dichiarazioni locali.

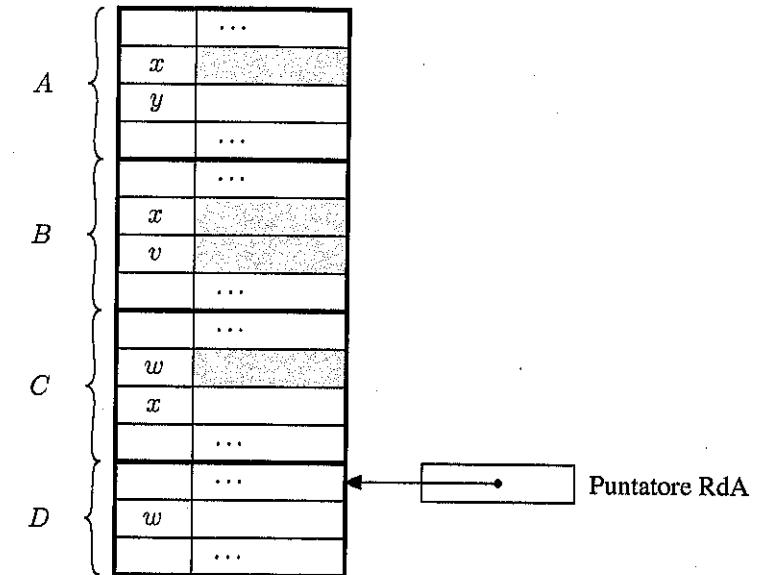
**Il chiamato si trova all'interno del chiamante.** In questo caso si incrementa il livello di profondità di annidamento raggiunto. Se il chiamante si trova a livello  $n$ , chiamato e chiamante condividono tutto il display corrente, fino alla posizione  $n$ , e si deve aggiungere un nuovo valore, in posizione  $n + 1$ , per contenere il puntatore al RdA del chiamato.

Nel caso in cui si tratti della prima attivazione di un blocco di livello  $n + 1$  il vecchio valore presente sul display non interessa. Tuttavia, in generale, non possiamo sapere se questo è il caso: infatti, si potrebbe essere giunti all'attuale chiamata da una serie di chiamate precedenti che utilizzavano anche il livello  $n + 1$ . Anche in questo caso dunque sarà necessario memorizzare il vecchio valore presente sul display in posizione  $n + 1$ .

Sia l'aggiornamento del display che il salvataggio del vecchio valore possono essere fatti dal chiamato. La Figura 7.15 mostra la gestione del display per la sequenza di chiamate A, B, C, D, E, C, facendo riferimento alla struttura a blocchi descritta nella Figura 7.13. I puntatori a sinistra della pila indicano la memorizzazione del vecchio valore del display nel RdA del chiamato, mentre il puntatore tratteggiato indica un puntatore del display che non è correntemente attivo.

### 7.5.3 Scope dinamico: lista di associazioni e CRT

L'implementazione della regola di scope dinamico concettualmente è molto più semplice di quella dello scope statico. Infatti, visto che gli ambienti non locali si considerano nell'ordine con cui sono attivati a run-time, per risolvere un riferimento non locale al nome  $x$  basterà, almeno in linea di principio, percorrere a ritroso la pila, partendo dal record di attivazione corrente, fino a trovare il RdA corrispondente al blocco nel quale il nome  $x$  è stato dichiarato.

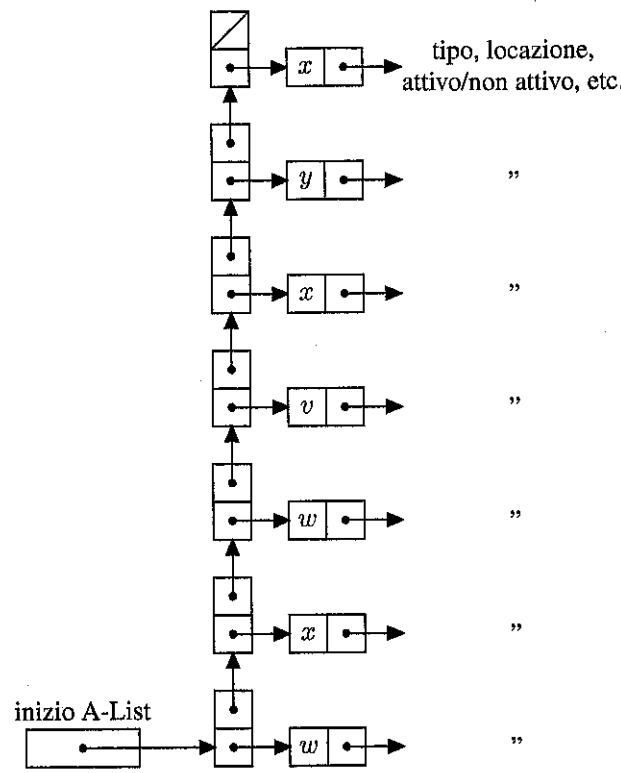


**Figura 7.17** Ambiente per il blocco D di Figura 7.16 dopo la sequenza di chiamate A, B, C, D, con scope dinamico realizzato mediante la memorizzazione delle associazioni nei RdA. In grigio i riferimenti non attivi.

Le varie associazioni fra nomi e oggetti denotati, che costituiscono i vari ambienti locali, possono essere memorizzate direttamente nei record di attivazione. Consideriamo ad esempio la struttura a blocchi mostrata in Figura 7.16, dove i nomi indicano delle dichiarazioni di variabili locali (con le usuali regole di visibilità). Se effettuiamo la sequenza di chiamate A, B, C, D (dove, al solito, tutte le chiamate rimangono attive) quando il controllo raggiunge il blocco D otteniamo la pila mostrata in Figura 7.17, dove il campo a destra di ogni nome contiene le informazioni associate all'oggetto denotato dal nome. L'ambiente (locale e non locale) di D è costituito da tutte le associazioni nome-oggetto denotato in cui il campo per le informazioni è in bianco nella figura. In grigio, invece, sono evidenziati i campi delle associazioni che non sono più attive. Un'associazione non è attiva o perché il corrispondente nome non è più visibile (questo è il caso di v) o perché il nome è stato ridefinito in un blocco interno (questo è il caso di w e delle occorrenze di x presenti in A ed in B).

Alternativamente alla memorizzazione diretta nei RdA, le associazioni nome-oggetto possono essere memorizzate separatamente in una lista di associazioni, detta *A-list*, e gestita come una pila. Questa soluzione è scelta normalmente dalle implementazioni di LISP.

Quando l'esecuzione di un programma entra in un nuovo ambiente le nuove associazioni locali vengono inserite nella A-list, mentre quando si esce da un am-

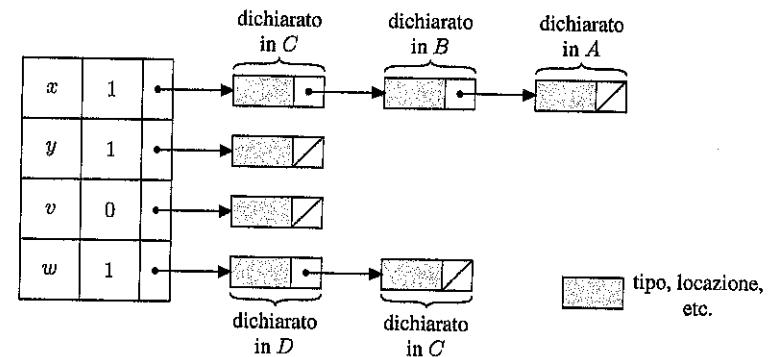


**Figura 7.18** Ambiente per il blocco D di Figura 7.16, dopo la sequenza di chiamate A,B,C,D, con scope dinamico realizzato mediante A-list.

biente le associazioni locali vengono rimosse dalla A-list. L'informazione relativa all'oggetto denotato conterrà la locazione di memoria dove l'oggetto è effettivamente memorizzato, il suo tipo, un flag che indica se l'associazione per quell'oggetto è attiva ed eventualmente altre informazioni necessarie per effettuare a run-time controlli semanticci. La Figura 7.18 mostra come viene realizzato lo scope dinamico mediante A-list per l'esempio di Figura 7.16 (i campi in grigio sono realizzati mediante i flag di cui sopra, non presenti in figura).

Sia che si memorizzino le associazioni direttamente nei RdA, sia che si usi una A-list separata, questo tipo di gestione dello scope dinamico, per quanto semplice, presenta due inconvenienti.

Innanzitutto i nomi, contrariamente a quanto abbiamo visto nel caso dell'implementazione dello scope statico, adesso devono essere memorizzati in strutture presenti a tempo di esecuzione. Con la A-list questo è evidente (dipende dalla sua stessa definizione). Nel caso invece in cui si usino i RdA per realizzare gli ambienti locali, la necessità di memorizzare i nomi dipende dal fatto che uno stesso



**Figura 7.19** Ambiente per il blocco D di Figura 7.16, dopo la sequenza di chiamate A,B,C,D, con scope dinamico realizzato mediante CRT.

nome, dichiarato in blocchi diversi, può essere memorizzato in posizioni differenti nei diversi RdA. Dato che se usiamo la regola di scope dinamico non possiamo determinare staticamente qual è il blocco (e quindi il RdA) da usarsi per risolvere un riferimento non locale, non possiamo sapere in quale posizione del RdA andare a cercare l'associazione per il nome che ci interessa. L'unica possibilità è dunque memorizzare esplicitamente il nome e fare una ricerca (sulla base del nome stesso) a tempo di esecuzione.

Il secondo inconveniente è dovuto alla inefficienza di questa ricerca a run-time: può essere frequente il caso di dover scandire quasi tutta la lista (sia essa A-list o pila dei RdA) nel caso si faccia riferimento ad un nome dichiarato in uno dei primi blocchi attivati (i nomi "globali").

**Tabella Centrale dell'Ambiente** Per limitare questi due inconvenienti, al prezzo di una maggiore inefficienza nelle operazioni di entrata ed uscita dal blocco, possiamo usare una diversa implementazione dello scope dinamico basata sulla *Tabella Centrale dell'Ambiente* o CRT (Central Referencing environment Table).

Secondo questa tecnica tutti i blocchi del programma, ai fini della definizione degli ambienti, fanno riferimento ad un'unica tabella centrale (la CRT, appunto). In tale tabella sono presenti tutti i nomi usati nel programma e per ogni nome è presente un flag, che indica se l'associazione per quel nome è attiva o meno, ed un valore che costituisce un puntatore alle informazioni sull'oggetto associato al nome (locazione di memoria, tipo ecc.). Se assumiamo che tutti gli identificatori usati nel programma siano noti a tempo di compilazione, ogni nome può avere una posizione fissa nella tabella. A run-time l'accesso alla tabella può quindi avvenire in tempo costante, sommando all'indirizzo di memoria dell'inizio della tabella un offset relativo alla posizione del nome che ci interessa. Se invece non tutti i nomi

A	AB	ABC	ABCD
x 1 $\alpha_1$	x 1 $\beta_1$	x 1 $\gamma_1$	x 1 $\gamma_1$
y 1 $\alpha_2$	y 1 $\alpha_2$	y 1 $\alpha_2$	y 1 $\alpha_2$
v 0 -	v 1 $\beta_2$	v 0 $\beta_2$	v 0 $\beta_2$
w 0 -	w 0 -	w 1 $\gamma_2$	w 1 $\delta_1$

x	$\alpha_1$
---	------------

x	$\alpha_1$
x	$\beta_1$

x	$\alpha_1$
x	$\beta_1$
w	$\gamma_2$

**Figura 7.20** Ambiente per il blocco D di Figura 7.16 dopo la sequenza di chiamate A,B,C,D, con scope dinamico realizzato mediante CRT con pila nascosta.

sono noti a tempo di compilazione, la ricerca della posizione di un nome nella tabella può essere fatta efficientemente a run-time usando tecniche di hashing. Le operazioni di entrata e uscita dal blocco adesso sono però più complicate: quando da un blocco A si entra in un blocco B, la tabella centrale deve essere modificata per descrivere il nuovo ambiente locale di B, tuttavia le associazioni deattivate dovranno essere opportunamente salvate per poter essere ripristinate quando si uscirà dal blocco B e si tornerà al blocco A. Al solito una pila è la struttura dati adatta per memorizzare queste associazioni.

Si noti che le associazioni relative ad un blocco non sono necessariamente memorizzate in posizioni contigue nella CRT. Per effettuare le operazioni viste sulla CRT all'entrata ed all'uscita dai blocchi si dovranno considerare quindi i singoli elementi della tabella. Questo può essere fatto convenientemente associando ad ogni entrata della tabella, ossia ad ogni nome presente, una specifica pila che contenga nella prima posizione (ossia al top) l'associazione valida e nelle posizioni successive le associazioni disattivate per quel nome. Questa soluzione è descritta in Figura 7.19.

Alternativamente, possiamo usare un'unica pila "nascosta", separata dalla tabella centrale, per memorizzare per tutti i nomi solo le associazioni deattivate. In questo caso la tabella contiene (nella seconda colonna) per ogni nome un flag che indica se l'associazione per quel nome è attiva o meno e quindi il riferimento all'oggetto denotato dal nome in questione (nella terza colonna). Quando un'associazione viene deattivata essa viene memorizzata nella pila nascosta, per poi essere ripresa dalla pila quando torna attiva. Sempre facendo riferimento alla struttura di Figura 7.16 e alla sequenza di chiamate A,B,C,D, l'evoluzione della CRT è mostrata nella parte superiore della Figura 7.20, mentre la parte inferiore di tale figura mostra l'evoluzione della pila nascosta.

Usando la CRT, con pila nascosta o meno, l'accesso ad una associazione dell'ambiente avviene mediante un accesso alla tabella (diretto oppure mediante una

funzione hash) ed un accesso ad un'altra zona di memoria mediante il puntatore memorizzato nella tabella. Non serve quindi alcuna ricerca a run-time.

## 7.6 Sommario del capitolo

In questo capitolo abbiamo esaminato le principali tecniche di gestione della memoria, sia statiche che dinamiche, illustrando i motivi che rendono necessaria la gestione dinamica con pila e che richiedono l'uso di un heap. Resta da vedere un'importante eccezione: in presenza di un particolare tipo di ricorsione (detta in coda) la memoria può essere gestita in modo statico, ma questo caso sarà trattato in dettaglio nel prossimo capitolo.

Riguardo alla gestione della pila, abbiamo illustrato in dettaglio:

- il formato dei record di attivazione per i blocchi in-line e per le procedure;
- come la pila sia gestita da particolari frammenti di codice, inseriti sia nel chiamante che nel chiamato, che servono per realizzare le varie operazioni di allocazione dei record di attivazione, inizializzazione, modifica dei campi di controllo, passaggio dei valori, restituzione dei risultati ecc.

Nel caso dello heap abbiamo visto:

- alcune delle tecniche più comuni per la sua gestione, sia con blocchi di dimensione fissa che con blocchi di dimensione variabile;
- il problema della frammentazione e alcuni metodi che si usano per limitarlo.

Abbiamo infine discusso le specifiche strutture dati e gli algoritmi usati per realizzare l'ambiente e, in particolare, per implementare le regole di scope. Ci siamo soffermati in dettaglio su:

- la catena statica;
- il display;
- la lista di associazioni;
- la tabella centrale dell'ambiente (CRT).

Questo ci ha permesso di capire meglio quanto accennato nel Capitolo 6 riguardo alla maggiore difficoltà nell'implementazione delle regole di scope statico rispetto alle regole di scope dinamico. Nel primo caso infatti, sia che si usino i puntatori di catena statica, sia che si usi il display, occorre che il compilatore fornisca opportune informazioni sulla struttura delle dichiarazioni. Queste informazioni sono raccolte dal compilatore usando tabelle dei simboli gestite da opportuni algoritmi, quali ad esempio quello di LeBlanc-Cook, i cui dettagli esulano dagli scopi del presente testo. Nel caso dello scope dinamico invece la gestione può essere, in linea di principio, fatta interamente a run-time, anche se, anche in questo caso, spesso si usano strutture dati ausiliarie (ad esempio la Central Referencing Table) per ottimizzare le prestazioni.

## 7.7 Nota bibliografica

La gestione statica della memoria è usualmente trattata nei testi sui compilatori, quale il classico [4]. La determinazione degli scope (statici) da associare ai nomi presenti nella tabella dei simboli può essere fatta in molti modi. Tra questi uno dei più noti è quello dovuto a LeBlanc e Cook [29]. Le tecniche di gestione dello heap sono discusse in molti testi, si veda ad esempio [90].

La gestione a pila delle procedure e dello scope fu introdotta con ALGOL, la cui implementazione è descritta in [83].

Per la gestione delle memorie nei singoli linguaggi di programmazione si vedano i testi specifici per ogni linguaggio, alcuni dei quali sono citati alla fine del Capitolo 16.

## 7.8 Esercizi

- Si scriva, in un qualsiasi pseudolinguaggio, un frammento di codice tale che il numero massimo dei RDA presenti a run-time sulla pila non sia determinabile staticamente.
- Si scriva, in un qualsiasi pseudolinguaggio, una funzione ricorsiva tale che il numero massimo dei RDA per essa presenti a run-time sulla pila sia determinabile staticamente. Si può generalizzare l'esempio?
- Consideriamo il seguente frammento di codice:

```
A: {int x= 1;
    ...
    B: { x = 3;
        ...
    }
    ...
}
```

Supponiamo il *B* si trovi annidato di un livello rispetto ad *A*. Perché, per risolvere il riferimento a *X* presente in *B*, non basta considerare il record di attivazione che sulla pila precede immediatamente quello di *B*? Si fornisca un controsenso riempendo gli spazi indicati con i puntini con del codice opportuno.

- Si consideri il seguente frammento di programma scritto in uno pseudolinguaggio che usa scope statico:

```
void P1 {
    void P2 { corpo-di-P2
    }
    void P3 {
        void P4 { corpo-di-P4
        }
        corpo-di-P3
    }
    corpo-di-P1
}
```

Si descriva graficamente la pila dei record di attivazione, limitatamente ai puntatori di catena statica e dinamica, dopo la sequente successione di chiamate: P1, P2, P3, P4, P2 (si intende che in tale momento sono tutte attive: nessuna ha ancora ritornato il controllo).

- È dato il seguente frammento di codice in uno pseudolinguaggio con goto (si veda il Paragrafo 8.3.1), scope statico e blocchi annidati etichettati (indicati con A :{ . . . }):

```
A: { int x = 5;
    goto C;
    B: {int x = 4;
        goto E;
    }
    C: {int x = 3;
        D: {int x = 2;
            ...
        }
        goto B;
        E: {int x = 1; // (**)
            }
    }
}
```

La catena statica è gestita mediante display. Si illustri graficamente la situazione del display e della pila nel momento in cui l'esecuzione raggiunge il punto segnato con il commento (\*\*). Per quanto riguarda i record di attivazione, si indichi la sola informazione necessaria alla gestione del display.

- È più semplice da implementare la regola di scope statico o quella di scope dinamico? Si motivi brevemente la risposta.
- Si consideri il seguente schema di codice scritto in uno pseudolinguaggio che usa scope statico e passaggio per riferimento (si veda il Paragrafo 9.1.2):

```
int x = 0;
int A(reference int y) {
    int x = 2;
    y=y+1;
    return B(y)+x;
}
int B(reference int y){
    int C(reference int y){
        int x = 3;
        return A(y)+x+y;
    }
    if (y==1) return C(x)+y;
    else return x+y;
}
write (A(x));
}
```

Supponiamo che lo scope statico sia implementato mediante display. Si dia graficamente la situazione del display e della pila dei record di attivazione al momento in cui il controllo entra per la seconda volta nella funzione A. Per ogni record di attivazione si dia solo il valore del campo destinato a salvare il valore precedente del display.

## Strutturare il controllo

Affronteremo in questo capitolo il controllo di sequenza, un importante aspetto nella definizione dell'interprete di una generica macchina astratta che riguarda la gestione del flusso di esecuzione delle istruzioni di un programma.

Nei linguaggi di basso livello il controllo di sequenza è realizzato in modo molto semplice aggiornando opportunamente il valore del registro PC (*program counter*, contatore di programma). Nei linguaggi di alto livello, invece, sono presenti opportuni costrutti, diversi a seconda del tipo di linguaggio considerato, che permettono di strutturare il controllo e di realizzare meccanismi molto più astratti di quelli presenti nella macchina fisica. Si pensi, ad esempio, alla semplice valutazione di un'espressione aritmetica: anche se per noi ovvia e naturale, un'operazione di questo genere comporta l'uso di opportuni meccanismi di controllo per specificare l'ordine con cui devono essere valutati gli operandi, le precedenze fra gli operatori ecc.

In questo capitolo vedremo i costrutti che si usano nei linguaggi di programmazione per specificare, implicitamente o esplicitamente, il controllo di sequenza. Vedremo innanzitutto le espressioni, soffermandoci sia sugli aspetti sintattici, relativi alle notazioni usate per rappresentare le espressioni, sia sugli aspetti semantici relativi alla loro valutazione. Passeremo quindi ai comandi e, dopo aver discusso la nozione di variabile ed il comando di assegnamento, vedremo i principali comandi per il controllo di sequenza presenti nei linguaggi moderni, evidenziando la differenza fra controllo strutturato e controllo non strutturato e illustrando brevemente i principi della programmazione strutturata. Esamineremo infine alcuni aspetti significativi relativi alla ricorsione e chiariremo anche un'importante distinzione terminologica fra linguaggi imperativi e linguaggi dichiarativi.

Rimandiamo invece al prossimo capitolo l'esame di quei costrutti che permettono di realizzare meccanismi di astrazione sul controllo.

### 8.1 Le espressioni

Le *espressioni*, insieme ai comandi ed alle dichiarazioni, costituiscono uno dei componenti di base di ogni linguaggio di programmazione. Possiamo anzi dire

che le espressioni sono il componente essenziale di ogni linguaggio, perché se esistono linguaggi dichiarativi nei quali non sono presenti i comandi, le espressioni, numeriche o simboliche, sono presenti in tutti i linguaggi.

Cerchiamo innanzitutto di chiarire quali sono gli oggetti dei quali ci stiamo occupando.

**Definizione 8.1 (Espressione)** *Un'espressione è un'entità sintattica la cui valutazione produce un valore oppure non termina, nel qual caso l'espressione è indefinita.*

La caratteristica essenziale di un'espressione, che la differenzia da un comando, è dunque che la sua valutazione produce un valore. Gli esempi di espressioni numeriche sono familiari a tutti:  $4+3*2$ , ad esempio, è un'espressione la cui valutazione è ovvia. Tuttavia si noti che, anche in un caso così semplice, per poter ottenere il risultato corretto abbiamo fatto un'assunzione implicita (derivata dalla convenzione della notazione matematica) sulla precedenza fra gli operatori. Tale assunzione, che ci dice che  $*$  ha precedenza su  $+$  (e che dunque il risultato della valutazione è 10 e non 14), specifica un aspetto di controllo di sequenza relativo alla valutazione delle espressioni. Vedremo più avanti altri aspetti più sottili che possono concorrere a modificare il risultato della valutazione di un'espressione.

Le espressioni possono anche essere non numeriche, ad esempio in LISP possiamo scrivere `(cons a b)` per indicare un'espressione che, se valutata, restituisce la cosiddetta coppia (puntata) formata da `a` e `b`.

### 8.1.1 Sintassi delle espressioni

In generale un'espressione è composta da un'entità singola (costante, variabile ecc.) oppure da un operatore (quali `+`, `cons` ecc.) applicato ad un certo numero di argomenti (o operandi) che sono a loro volta espressioni. Abbiamo visto nel Capitolo 2 come la sintassi delle espressioni possa essere espressa in modo preciso da una grammatica libera e come un'espressione possa essere rappresentata da un albero di derivazione che, oltre alla sintassi, esprima anche alcune informazioni semantiche relative alla valutazione dell'espressione stessa. E, in effetti, spesso si usano strutture ad albero per rappresentare le espressioni internamente al calcolatore. Tuttavia, per poter usare convenientemente le espressioni nel testo dei programmi, sono necessarie delle notazioni lineari che permettano di scrivere un'espressione come una sequenza di simboli. In sostanza, le varie notazioni si differenziano a seconda di come si rappresenta l'applicazione di un operatore ai suoi operandi. Possiamo distinguere tre tipi principali di notazioni.

**Notazione infissa** Secondo questa notazione, il simbolo di un operatore binario è posto fra le espressioni che rappresentano i due operandi. Ad esempio scriviamo `x+y` per indicare la somma di `x` e `y` oppure `(x+y)*z` per indicare la moltiplicazione di `z` per il risultato della somma di `x` e `y`. Si noti che, per evitare ambiguità nell'applicazione degli operatori agli operandi, sono necessarie le

### Il Lisp e le S-espressioni

Il linguaggio di programmazione LISP (acronimo di LISt Processor), sviluppato agli inizi degli anni '60 ad opera di John McCarthy e di un gruppo di ricercatori del M.I.T. (Massachusetts Institute of Technology), è un linguaggio concepito per l'elaborazione simbolica che è stato molto importante soprattutto nell'ambito dell'intelligenza artificiale. Per 30 anni si sono sviluppate molte versioni diverse del linguaggio. In particolare, negli anni '70, da un dialetto di LISP si sviluppò il linguaggio Scheme, tuttora molto usato in ambito accademico.

Un programma LISP è costituito da una sequenza di espressioni che devono essere valutate dall'interprete del linguaggio. Alcune espressioni sono usate per definire delle funzioni, che poi sono chiamate in altre espressioni. Il controllo è gestito usando la ricorsione (non vi è alcun costrutto iterativo).

Come indica il nome stesso del linguaggio, un programma LISP manipola soprattutto espressioni, costituite sostanzialmente da liste. La struttura dati di base di LISP infatti è la coppia puntata, ossia una coppia scritta con un punto di separazione fra le due componenti, come `(A . B)`. Una tale coppia è rappresentata da una cella `cons`, ossia dall'applicazione di un operatore `cons` a due argomenti, come in `(cons A B)`. Dato che i due argomenti di una operazione `cons` possono essere, oltre ad elementi atomici (interi, numeri in virgola mobile, stringhe di caratteri), anche altre coppie puntate, questa struttura dati permette di realizzare espressioni simboliche, le cosiddette S-espressioni. Si tratta sostanzialmente di alberi binari che, come caso particolare, permettono di rappresentare le liste: ad esempio, la lista `A B C` può essere rappresentata da `(cons A (cons B (cons C nil)))` dove `nil` è un valore particolare che indica la lista vuota. Fra le molte caratteristiche interessanti di LISP è da notare che i programmi ed i dati in LISP usano la stessa sintassi e la stessa rappresentazione interna. Questo permette di valutare delle strutture dati come se fossero dei programmi e di modificare i programmi come se fossero dati.

parentesi e opportune regole di precedenza. Per gli operatori non binari si ricorre sostanzialmente ad una loro rappresentazione in termini di più simboli binari, anche se in questo caso la notazione infissa non è quella più naturale da usare. Un linguaggio di programmazione che insiste nella notazione infissa anche per le funzioni definite dall'utente è Smalltalk, un linguaggio orientato agli oggetti.

La notazione infissa è quella più comunemente usata in matematica e, di conseguenza, è quella adottata dalla maggior parte dei linguaggi di programmazione, almeno per gli operatori binari e a livello di notazione visibile all'utente. Spesso, infatti, tale notazione costituisce solo un'abbreviazione o, come si dice con un'espressione colorita, uno "zucchero sintattico" usato per rendere più leggibile il codice: in Ada, ad esempio, `a + b` è un'abbreviazione per `+(a, b)`, mentre in C++ la stessa espressione è un'abbreviazione per `a.operator+(b)`.

**Notazione prefissa** In questo tipo di notazione, detta anche *notazione polacca prefissa*<sup>1</sup>, il simbolo che rappresenta l'operatore precede i simboli che rappresentano gli operandi (nella usuale scrittura, da sinistra a destra). Così, per indicare la somma di  $x$  e  $y$  possiamo scrivere  $+(x\ y)$  o anche, senza usare le parentesi,  $+ x\ y$  mentre per indicare l'applicazione della funzione  $f$  agli operandi  $a$  e  $b$  scriviamo  $f(a\ b)$  oppure  $f\ a\ b$ .

È importante notare che usando questo tipo di notazione non servono parentesi e regole di precedenza fra gli operatori, purché sia nota l'arietà (cioè il numero di operandi) di ogni operatore. Infatti non vi è ambiguità su quale sia l'operatore da applicarsi a determinati operandi, dato che questo è sempre quello che precede immediatamente gli operandi stessi. Ad esempio, se scriviamo

$*(+ (a\ b) + (c\ d))$

oppure

$* + a\ b + c\ d$

indichiamo l'espressione che nella più usuale forma infissa è rappresentata da  $(a+b)\ * (c+d)$ .

La maggior parte dei linguaggi di programmazione usa la notazione prefissa per gli operatori unari (spesso con le parentesi che racchiudono l'argomento) e per le funzioni definite dall'utente. Alcuni linguaggi di programmazione usano la notazione prefissa anche per gli operatori binari. LISP usa per le funzioni la particolare notazione infissa nota come "polacca di Cambridge" che include anche gli operatori fra parentesi; ad esempio, la precedente espressione in questa notazione è

$(* (+ a\ b) (+ c\ d))$ .

**Notazione postfissa** Questa notazione, detta anche *polacca inversa*, è analoga alla precedente, con la differenza che il simbolo dell'operatore segue quello degli operandi. Ad esempio, l'ultima espressione vista sopra in forma postfissa è scritta come

$a\ b + c\ d + *$ .

La notazione postfissa è impiegata nel codice intermedio usato da alcuni compilatori ed è usata anche da alcuni linguaggi di programmazione (ad esempio, Postscript).

In generale, un vantaggio della notazione polacca (sia prefissa che inversa) rispetto a quella infissa è che la prima può essere usata per rappresentare in modo uniforme operatori con un numero qualsiasi di operandi, mentre, nel caso della notazione infissa, per rappresentare operatori con più di due operandi dobbiamo introdurre degli operatori ausiliari. Un secondo vantaggio, come già detto, è la

<sup>1</sup>Questa terminologia deriva dal fatto che il matematico polacco W. Lukasiewicz fu colui che diffuse la notazione prefissa senza parentesi.

possibilità di evitare completamente le parentesi anche se, per motivi di leggibilità, sia la notazione prefissa matematica  $f(a\ b)$  che la polacca di Cambridge ( $f\ a\ b$ ) usano le parentesi. Un ulteriore vantaggio della notazione polacca, come vedremo nel prossimo paragrafo, è che essa permette di valutare un'espressione in modo molto semplice. Per questo motivo questa notazione ebbe un certo successo negli anni '70-'80, quando fu usata dalle prime macchine calcolatrici tascabili.

### 8.1.2 Semantica delle espressioni

A seconda di come si rappresenta un'espressione, varia il modo in cui se ne determina la semantica e quindi la sua modalità di valutazione. In particolare, per la rappresentazione infissa l'assenza di parentesi può causare problemi di ambiguità se non sono definite chiaramente le regole di precedenza fra operatori diversi e di associatività per ogni operatore binario. Quando, poi, si consideri il caso dei linguaggi di programmazione più comuni, nella valutazione delle espressioni si deve anche considerare il fatto che spesso queste vengono rappresentate internamente sotto forma di albero. In questo paragrafo discuteremo queste problematiche, partendo dalla valutazione delle espressioni nelle tre notazioni viste in precedenza.

**Notazione infissa: precedenza e associatività** Usando la notazione infissa si paga la facilità e la naturalezza d'uso con una maggiore complicazione nei meccanismi di valutazione delle espressioni. Innanzitutto, se non si usano estensivamente le parentesi occorre chiarire le precedenze fra i vari operatori.

Se scriviamo  $4+3*5$ , ad esempio, evidentemente intendiamo il valore 19, e non 35, come risultato dell'espressione: le convenzioni matematiche, infatti, ci dicono che dobbiamo fare *prima* la moltiplicazione e *poi* la somma, ossia la precedente espressione è da intendersi come  $4+(3*5)$  e non come  $(4+3)*5$ . Nel caso di operatori meno familiari, presenti nei linguaggi di programmazione, le cose si complicano: se ad esempio in Pascal scrivessimo

$x=4 \text{ and } y=5$

dove *and* è l'operatore logico, contrariamente a quello che probabilmente molti si aspettarebbero, otterremmo un errore (statico di tipo) perché secondo le regole di precedenza di Pascal tale espressione viene interpretata come

$x=(4 \text{ and } y)=5$

e non come

$(x=4) \text{ and } (y=5)$ .

Per evitare un uso eccessivo di parentesi (che comunque, nel dubbio, è bene utilizzare), i linguaggi di programmazione usano dunque delle *regole di precedenza* che specificano una gerarchia fra gli operatori del linguaggio relativamente all'ordine di valutazione. I vari linguaggi differiscono notevolmente in queste regole, anche se, quasi sempre, almeno le convenzioni della notazione matematica sono rispettate.

Un secondo problema nella valutazione di un'espressione è relativo all'associatività degli operatori che vi compaiono: se scriviamo  $15-5-3$ , infatti, potremmo intendere questa espressione sia come  $(15-5)-3$ , sia come  $15-(5-3)$ , con risultati evidentemente diversi. Anche in questo caso la convenzione matematica ci dice che il significato inteso è il primo: detto in termini formali, l'operatore “ $-$ ” associa da sinistra a destra<sup>2</sup>. In effetti la maggior parte degli operatori aritmetici nei linguaggi di programmazione associano *da sinistra a destra*, ma vi sono eccezioni. L'operatore di esponenziazione, ad esempio, associa spesso da destra a sinistra, come del resto accade anche nella notazione matematica: se scriviamo  $5^{3^2} 0$ , usando una notazione più familiare ai programmati,  $5**3**2$ , intendiamo  $5^{(3^2)}$ , ossia  $5 * (3 ** 2)$ , e non  $(5^3)^2$ , cioè  $(5 * 3) ** 2$ . Anche nel caso in cui si usi un operatore è quindi bene, nel dubbio, fare uso di parentesi, anche perché non mancano i linguaggi particolari che hanno dei comportamenti controidintuitivi. In APL, ad esempio, l'espressione  $15-5-3$  è interpretata come  $15-(5-3)$  invece che come ci aspetteremmo: la ragione di questa apparente stranezza è che in APL, essendo presenti molti operatori nuovi (progettati per operare su vettori) che non hanno un corrispondente immediato in altri formalismi, è stato deciso di non adottare alcuna precedenza fra gli operatori e di valutare tutte le espressioni da destra a sinistra.

Anche se non è difficile immaginarsi un algoritmo diretto che permetta di valutare un'espressione scritta usando la notazione infissa, la presenza implicita delle regole di precedenza e associatività, unitamente alla presenza esplicita delle parentesi, complica un poco le cose. Infatti, non è possibile valutare un'espressione con un'unica scansione da sinistra a destra (o da destra a sinistra), dato che in alcuni casi dobbiamo prima valutare la parte successiva di un'espressione per poi ritornare a quella che stiamo considerando (ad esempio, nel caso di  $5+3*2$ , quando nella scansione da sinistra a destra arriviamo al  $+$ , dobbiamo sospendere la valutazione di questo operatore, andare avanti per valutare  $3*2$  e quindi riprendere la valutazione del  $+$ ).

**Notazione prefissa** Le espressioni scritte usando la notazione polacca prefissa si prestano ad essere valutate in modo semplice scandendo l'espressione da sinistra a destra ed usando una pila. Supponiamo che la sequenza dei simboli che costituiscono l'espressione sia sintatticamente corretta e inizialmente non vuota. L'algoritmo per la valutazione è descritto dai seguenti passi, dove usiamo una normale pila (con le operazioni di push e pop) ed un contatore  $C$  per memorizzare il numero di operandi richiesto dall'ultimo operatore letto:

1. Leggi il prossimo simbolo dell'espressione e mettilo sulla pila.
2. Se il simbolo letto è un operatore, inizializza il contatore  $C$  con il numero di argomenti dell'operatore e torna a (1).

<sup>2</sup>Un operatore binario op si dice associativo se vale che  $x \text{ op } (y \text{ op } z) = (x \text{ op } y) \text{ op } z$ , ossia se associare da destra a sinistra o da sinistra a destra non fa differenza.

3. Se il simbolo letto è un operando decrementa  $C$ .
4. Se  $C \neq 0$ , torna a (1).
5. Se  $C = 0$  esegui le seguenti operazioni:
  - (a) applica l'ultimo operatore inserito nella pila agli operandi inseriti successivamente, memorizza il risultato in  $R$ , elimina operatore ed operandi dalla pila e memorizza il valore di  $R$  sulla pila;
  - (b) se non vi sono simboli di operatore sulla pila vai a (6);
  - (c) inizializza il contatore  $C$  con  $n - m$  dove  $n$  è il numero di argomenti dell'operatore che si trova al top della pila e  $m$  è il numero di operandi presenti sulla pila sopra tale operatore;
  - (d) torna a (4).
6. Se la sequenza che rimane da leggere non è vuota torna a (1).

Il risultato della valutazione si troverà alla fine sulla pila. Si noti come per poter valutare un'espressione usando il precedente algoritmo abbiamo bisogno di conoscere preventivamente il numero di operandi di ogni operatore, cosa che ci costringe, ad esempio, a distinguere sintatticamente l'operatore — unario da quello binario. Inoltre, in generale è necessario controllare se sulla pila si hanno abbastanza operandi per l'operatore più in alto presente sulla pila stessa (punto 5.(c)). Questo controllo non è necessario se si usa la notazione postfissa, come vediamo qui sotto.

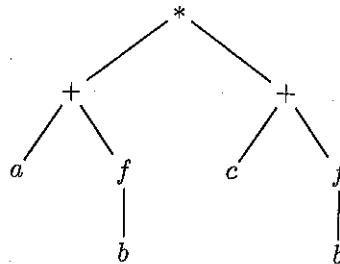
**Notazione postfissa** La valutazione delle espressioni nel caso della notazione polacca inversa è ancora più semplice: in questo caso infatti non serve controllare che sulla pila ci siano tutti gli operandi per l'ultimo operatore inserito, dato che gli operandi vengono letti (da sinistra a destra) prima degli operatori. L'algoritmo per la valutazione è dunque il seguente (al solito, supponiamo che la sequenza sia sintatticamente corretta e non vuota):

1. Leggi il prossimo simbolo dell'espressione e mettilo sulla pila.
2. Se il simbolo letto è un operatore, applicalo agli operandi immediatamente precedenti sulla pila, memorizza il risultato in  $R$ , elimina operatore ed operandi dalla pila e memorizza il valore di  $R$  sulla pila.
3. Se la sequenza che rimane da leggere non è vuota torna a (1).
4. Se il simbolo letto è un operando torna a (1).

Anche in questo caso, comunque, abbiamo bisogno di conoscere preventivamente il numero di operandi di ogni operatore.

### 8.1.3 Valutazione delle espressioni

Come abbiamo detto all'inizio del capitolo e come abbiamo discusso estesamente nel Capitolo 2, le espressioni, così come gli altri costrutti di un linguaggio di programmazione, possono essere convenientemente rappresentate da alberi. In particolare, un'espressione può essere rappresentata da un albero (detto *albero sintattico* dell'espressione) nel quale:



**Figura 8.1** Un'espressione.

- ogni nodo che non è una foglia è etichettato con un operatore;
- ogni sottoalbero che ha come radice un figlio di un nodo  $N$  costituisce un operando per l'operatore associato a  $N$ ;
- ogni nodo che è una foglia è etichettato con una costante, una variabile o un altro operando elementare.

Questi alberi possono essere ottenuti in modo diretto dagli alberi di derivazione di una grammatica (non ambigua) per le espressioni, sostanzialmente eliminando i simboli non terminali e ridisponendo opportunamente i nodi. Si noti anche che, data la rappresentazione ad albero di un'espressione, le rappresentazioni lineari infissa, prefissa e postfissa si ottengono rispettivamente dalle visite dell'albero in ordine *simmetrico*, *anticipato* e *differito*. La rappresentazione ad albero di un'espressione chiarisce (senza bisogno di parentesi) precedenza e associatività degli operatori: i sottoalberi che si trovano più in basso costituiscono gli operandi, e dunque gli operatori più in basso devono essere valutati prima di quelli più in alto. Ad esempio, l'albero della Figura 8.1 rappresenta l'espressione

$(a+f(b)) * (c+f(b))$

che può essere ottenuta (parentesi a parte) dalla visita in ordine simmetrico ( $f$  è qui un generico operatore unario).

Per i linguaggi implementati in modo compilativo, come abbiamo visto, il parser realizza l'analisi sintattica costruendo un albero di derivazione. Nel caso specifico delle espressioni dunque, la notazione infissa presente nel codice sorgente viene tradotta in una rappresentazione ad albero. Questa rappresentazione è poi usata nelle fasi successive del processo di compilazione per generare il codice oggetto che realizzerà, a run time, la valutazione dell'espressione. Tale codice oggetto ovviamente dipenderà dal tipo di macchina per la quale il compilatore è costruito. Nel caso in cui si tratti di una macchina fisica tradizionale, ad esempio, il codice che permette la valutazione delle espressioni conterrà una sequenza di istruzioni di tipo "tradizionale" (ossia nella forma codice istruzione, operando 1, operando 2) che useranno anche registri e locazioni di memoria temporanee per memorizzare i risultati intermedi della valutazione.

In alcuni casi particolari, invece, il codice oggetto può essere rappresentato in forma prefissa o postfissa, per poi essere valutato da una architettura a pila: questo è il caso, ad esempio, dei codici eseguibili dei programmi in SNOBOL4 in molte implementazioni.

Anche nel caso in cui il linguaggio sia interpretato è conveniente tradurre le espressioni, normalmente presenti nel codice sorgente in notazione infissa, in una rappresentazione ad albero che poi può essere valutata direttamente mediante una visita dell'albero. È questo il caso, ad esempio, delle implementazioni interpretative di LISP, nelle quali l'intero programma è rappresentato come un albero.

Non è negli scopi del presente testo entrare nei dettagli dei meccanismi di generazione del codice o di valutazione delle espressioni da parte di un interprete. Tuttavia, è importante chiarire alcuni punti problematici che spesso sono causa di ambiguità. Per nostra comodità faremo riferimento alla valutazione di espressioni rappresentate in forma infissa. Quanto vedremo si può comunque applicare anche alla valutazione diretta di un'espressione rappresentata da un albero nonché ai meccanismi di generazione di codice.

**Ordine di valutazione delle sotto-espressioni** Le regole di precedenza e associatività, nella notazione infissa, o la struttura, nella rappresentazione ad albero, non forniscono alcuna indicazione su quale ordine seguire nel valutare i vari operandi di uno stesso operatore (ossia i nodi allo stesso livello). Ad esempio, nell'espressione della Figura 8.1 nessuno ci dice se deve essere valutata prima  $(a+f(b))$  oppure  $(c+f(b))$ . Non è neanche esplicito se la valutazione degli operandi debba precedere o meno quella dell'operatore, né, più in generale, se espressioni che sono matematicamente equivalenti possano essere sostituite tra loro senza modificare la semantica (ad esempio,  $(a-b+c)$  e  $(a+c-b)$  potrebbero essere considerate equivalenti).

Mentre in termini matematici queste differenze non hanno importanza (comunque il risultato non cambia), dal nostro punto di vista queste questioni sono estremamente rilevanti, per i seguenti cinque motivi.

**Effetti collaterali** Nei linguaggi di programmazione di tipo imperativo la valutazione di un'espressione può modificare il valore di alcune variabili mediante i cosiddetti *effetti collaterali* (o *lateral*). Un effetto collaterale è un'azione che influenza i risultati (parziali o finali) di una computazione senza però restituire esplicitamente un valore al contesto nel quale essa è presente. La possibilità di effetti collaterali fa sì che l'ordine di valutazione degli operandi sia rilevante ai fini del risultato. Nel nostro solito esempio della Figura 8.1, se la valutazione della funzione  $f$  modificasse, per effetto collaterale, il valore del suo operando, evidentemente eseguire prima  $(a+f(b))$ , oppure  $(c+f(b))$  potrebbe cambiare il valore ottenuto dalla valutazione (si veda l'Esercizio 1). Riguardo agli effetti collaterali i linguaggi seguono varie filosofie: a parte i linguaggi dichiarativi puri che non ammettono effetti collaterali *tout court*, nei linguaggi che

invece li permettono in alcuni casi si vieta l'uso, nelle espressioni, di funzioni che possono causare effetti collaterali. In altri casi (ed è la situazione più comune) si ammette la presenza di effetti collaterali, specificando però chiaramente, nella definizione del linguaggio, l'ordine con cui vengono valutate le espressioni. Java, ad esempio, impone la valutazione da sinistra a destra per le espressioni mentre C non fissa alcun ordine.

**Aritmetica finita** Dato che l'insieme dei numeri rappresentabili in un calcolatore è finito (si veda anche il Paragrafo 10.3), il riordinamento di espressioni può causare problemi di overflow: ad esempio, se  $a$  ha il valore del massimo intero rappresentabile e  $b$  e  $c$  sono positivi con  $b > c$ , la valutazione da sinistra a destra di  $(a-b+c)$  non produce overflow, mentre si ha overflow valutando da sinistra a destra  $(a+c-b)$ . Inoltre, anche in assenza di overflow, la precisione limitata dell'aritmetica del calcolatore fa sì che cambiando l'ordine degli operandi si possano ottenere risultati diversi (ciò è particolarmente rilevante nel caso di computazioni in virgola mobile).

**Operandi non definiti** Quando si valuta l'applicazione di un operatore ad un operando si possono seguire due strategie di valutazione: la prima, detta *eager*, consiste nel valutare prima tutti gli operandi e poi passare ad applicare l'operatore ai valori ottenuti dalla valutazione degli operandi. Questa strategia probabilmente appare la più ragionevole se ragioniamo in termini dei normali operatori aritmetici. Le espressioni che usiamo nei linguaggi di programmazione però pongono qualche problema in più rispetto a quelle aritmetiche, in quanto alcune di esse possono essere definite anche se alcuni operandi che vi compaiono non lo sono. Consideriamo ad esempio un'espressione condizionale della forma

```
a == 0 ? b : b/a
```

che possiamo scrivere in C per indicare il valore di  $b/a$ , nel caso in cui  $a$  sia diverso da zero e il valore di  $b$  altrimenti. Una tale espressione risulta dall'applicazione di un unico operatore (espresso in forma infissa usando due operatori binari `?` e `:`) a tre operandi (l'espressione booleana  $a == 0$  e le espressioni aritmetiche  $b$  e  $b/a$ ). Se usassimo una strategia di valutazione eager evidentemente si perderebbe il senso di una simile espressione condizionale, in quanto l'espressione  $b/a$  dovrebbe essere valutata anche nel caso in cui  $a$  sia eguale a zero, producendo così comunque un errore.

In una tale situazione è dunque più indicato usare una strategia di valutazione *lazy* che sostanzialmente consiste nel *non* valutare gli operandi prima dell'operatore, ma nel passare gli operandi non valutati all'operatore il quale, al momento della sua valutazione, deciderà quali operandi effettivamente sono necessari, valutando solo questi.

La strategia di valutazione lazy, usata in alcuni linguaggi dichiarativi, è molto più costosa da implementare di quella eager e per questo motivo la maggior parte dei linguaggi imperativi adotta quest'ultima tecnica di valutazione (con l'importante eccezione delle espressioni condizionali, come vedremo tra un attimo). Vi sono linguaggi che usano un mix di entrambe le tecniche (ALGOL ad esempio). Discuteremo più approfonditamente le diverse strategie di valutazione

delle espressioni quando parleremo dei linguaggi funzionali nel Capitolo 13.

**Valutazione con "corto-circuito"** Il problema delineato al punto precedente si presenta con particolare evidenza nella valutazione delle espressioni booleane: ad esempio, nell'espressione (in sintassi C)

```
a == 0 || b/a > 2
```

se il valore di  $a$  è 0 e vengono valutati entrambi gli operandi di `||` evidentemente otteniamo un errore (in C "`||`" indica l'operatore logico di disgiunzione). Per evitare questo problema, ed anche per migliorare l'efficienza del codice, C, come altri linguaggi, usa la valutazione *lazy*, detta anche con "corto-circuito", delle espressioni booleane: se il primo operando di una disgiunzione ha valore *vero* allora il secondo non viene valutato, dato che comunque il risultato globale avrà sicuramente valore *vero*. L'espressione è "corto-circuitata" nel senso che si arriva al valore finale prima di conoscere il valore di tutti gli operandi. Analogamente, se il primo operando di una congiunzione ha valore *falso* allora il secondo non viene valutato, dato che comunque il risultato globale avrà valore *falso*.

È opportuno ricordare che non tutti linguaggi usano una tale strategia per le espressioni booleane: contare sulla presenza di una valutazione corto-circuito, senza essere certi che il linguaggio la adotti, può condurre ad errori. Ad esempio, potremmo scrivere in Pascal

```
p := lista;
while (p <> nil) and (p^.valore <> 3) do
  p := p^.prossimo;
```

con lo scopo di scorrere una lista fino a quando non si arriva al termine oppure viene trovato il valore 3. Si tratterebbe tuttavia di codice mal scritto, che potrebbe produrre errore a tempo d'esecuzione. Pascal, infatti, non usa la valutazione "corto circuito": nel caso in cui si abbia  $p = \text{nil}$ , si valuterebbe comunque anche il secondo operando della congiunzione  $(p!^.valore <> 3)$  producendo così un errore, perché si dereferenzia un puntatore nullo. Un simile codice invece, *mutatis mutandis*, potrebbe essere scritto in C senza causare alcun problema. Alcuni linguaggi (ad esempio C e Ada) per evitare ambiguità forniscono esplicitamente operatori booleani distinti per la valutazione corto-circuito. Infine si noti che tale valutazione può comunque essere simulata usando un comando condizionale (si veda l'Esercizio 2).

**Ottimizzazione** Spesso l'ordine di valutazione delle sotto-espressioni influenza l'efficienza della valutazione di un'espressione a causa di considerazioni che hanno a che fare con la struttura della macchina hardware. Ad esempio, se abbiamo il codice

```
a = vettore[i];
b = a*a + c*d;
```

nella seconda espressione è probabilmente meglio valutare prima  $c*d$  in quanto il valore di  $a$ , dovendo essere prelevato dalla memoria (con la prima istruzione), potrebbe non essere ancora disponibile, per cui il processore dovrebbe attendere prima di calcolare  $a*a$ . In alcuni casi, i compilatori possono cambiare

l'ordine degli operandi nelle espressioni per ottenere codice più efficiente, ma semanticamente equivalente.

L'ultimo punto spiega molti dei problemi semanticici che si presentano con la valutazione delle espressioni. Data l'importanza dell'efficienza del codice oggetto prodotto dal compilatore, spesso si lascia a quest'ultimo un'ampia libertà nella definizione precisa del metodo di valutazione delle espressioni usato, senza che questo venga specificato a livello di descrizione semantica del linguaggio (come abbiamo detto, Java è una rara eccezione). Il risultato di questo tipo di approccio è che, a volte, implementazioni diverse dello stesso linguaggio producono risultati diversi per la stessa espressione, oppure si hanno degli errori a run time dei quali non si riesce a capire con facilità l'origine.

Volendo capitalizzare in una prescrizione di tipo pragmatico quanto sin qui detto, se non si conoscono bene il linguaggio di programmazione e la specifica implementazione che si stanno usando, per scrivere codice corretto è opportuno usare tutti i mezzi possibili (parentesi, operatori booleani specifici, variabili ausiliarie nelle espressioni ecc) per eliminare al massimo le possibili fonti di ambiguità nella valutazione delle espressioni.

## 8.2 La nozione di comando

Se, come dicevamo poc'anzi, le espressioni sono presenti in tutti i linguaggi di programmazione, non è così per i comandi, che sono costrutti presenti tipicamente nei linguaggi cosiddetti imperativi (ma non solo in questi).

**Definizione 8.2 (Comando)** *Un comando è una entità sintattica la cui valutazione non necessariamente restituisce un valore, ma può avere un effetto collaterale.*

Un comando o, più in generale, un qualsiasi altro costrutto, ha un effetto collaterale se esso influenza il risultato della computazione senza che la sua valutazione restituisca alcun valore al contesto nel quale esso si trova.

Questo punto è abbastanza delicato e merita di essere chiarito con un esempio. Se il comando `stampa` in un ipotetico linguaggio di programmazione permette di stampare la stringa di caratteri che sono forniti come argomento, quando valutiamo il comando `stampa "pippo"` non otteniamo alcun valore ma solo un effetto collaterale (visibile) costituito dall'apparire dei caratteri "pippo" sul dispositivo di uscita.

Il lettore attento si sarà accorto che la definizione di comando, così come la precedente definizione di espressione, non è molto precisa, dato che abbiamo fatto riferimento ad una nozione informale di valutazione, quella fatta dalla macchina astratta del linguaggio a cui il comando o l'espressione appartengono. È evidente che potremmo sempre modificare l'interprete in modo tale da ottenere qualche valore come risultato della valutazione di un comando: questo è quanto avviene in alcuni linguaggi (ad esempio in C l'assegnamento restituisce anche il valore a destra di `=`, Paragrafo 8.2.2).

x 3

Figura 8.2 La variabile modificabile.

Una definizione e distinzione precisa tra espressioni e comandi sulla base della loro semantica è possibile solo in sede di definizione formale della semantica di un linguaggio. In tale contesto, la differenza tra i due concetti risulta dal fatto che, fissato uno stato di partenza, il risultato della valutazione di un'espressione è un valore (insieme ai possibili effetti collaterali), mentre il risultato della valutazione di un comando è un nuovo stato, che differisce da quello di partenza proprio per le modifiche apportate dagli effetti collaterali del comando stesso (e che sono dovute in via principale agli assegnamenti). Un comando è dunque un costrutto il cui scopo è la modifica dello *stato*; la nozione di stato può essere definita in vario modo: nel Paragrafo 2.5 ne abbiamo vista una semplice versione, quella che prende in considerazione i valori di tutte le variabili presenti nel programma.

Se scopo di un comando è modificare lo stato, è chiaro che il comando di assegnamento costituisce l'unità elementare nel meccanismo di computazione dei linguaggi con comandi. Prima di trattare di esso, tuttavia, occorre chiarire la nozione di variabile.

### 8.2.1 La variabile

In matematica una variabile è un'incognita che può assumere tutti i valori numerici di un insieme prefissato. Nell'ambito dei linguaggi di programmazione, anche se l'intuizione rimane valida, conviene specificare ulteriormente questa nozione perché, come vedremo anche nei Paragrafi 13.1 e 14.3, il paradigma imperativo usa un modello di variabile sostanzialmente diverso da quello presente nei paradigmi funzionale e logico.

Il paradigma imperativo classico usa la *variabile modificabile*. Secondo questo modello la variabile è vista come una sorta di contenitore, o locazione (con evidente riferimento alla memoria fisica), al quale si può dare un nome e che può contenere dei valori (solitamente di tipo omogeneo, ad esempio interi, reali, caratteri ecc.). Questi valori possono cambiare nel tempo, in seguito all'esecuzione di comandi di assegnamento (da cui l'aggettivo "modificabile"). Questa terminologia potrebbe sembrare tautologica all'informatico medio, quasi sempre conoscitore di un linguaggio imperativo e dunque abituato a modificare variabili. Il lettore attento tuttavia avrà notato che in effetti le variabili non sempre sono modificabili: in matematica una variabile rappresenta un valore sconosciuto, ma quando tale valore viene definito il legame così creatosi non può essere successivamente modificato.

La situazione che si costituisce con la variabile modificabile è quella descritta dalla Figura 8.2: la "scatolina" che rappresenta la variabile di nome `x` può essere "riempita" con un valore (nella figura il valore 3). Si noti che la variabile (la

scatola) è cosa diversa dal nome  $x$  che la denota, anche se è comune dire “la variabile  $x$ ” invece che “la variabile di nome  $x$ ”.

Nel contesto di alcuni linguaggi imperativi (soprattutto orientati agli oggetti), viene adottato un modello di variabile diverso da quello appena visto. Secondo questo modello alternativo, una variabile non è costituita da un contenitore per un valore, ma è un *riferimento* per (cioè un meccanismo che permette di accedere a) un valore, quest'ultimo tipicamente memorizzato sullo heap. Si tratta quindi di una nozione analoga a quella di puntatore (ma senza la possibilità di manipolazione che i puntatori in genere portano con sé), come vedremo nel prossimo paragrafo dopo aver introdotto il comando di assegnamento. Questo modello delle variabili è chiamato in [88] “reference model” (modello a riferimento), mentre in [57], dove è discusso nell’ambito del linguaggio CLU, è chiamato “object model”. Nel seguito del testo ci riferiremo ad esso come il *modello a riferimento* delle variabili.

I linguaggi funzionali (puri), come vedremo meglio nel Paragrafo 13.1, usano una nozione di variabile simile a quella della notazione matematica: una variabile altro non è che un identificatore che denota un valore. Anzi, spesso si dice che i linguaggi funzionali “non hanno variabili”, intendendo con questo che (nelle loro versioni pure) non possiedono le variabili modificabili.

Anche i linguaggi logici usano come variabili degli identificatori associati a valori e, come per i linguaggi funzionali, una volta creato un legame fra un identificatore di variabile ed un valore tale legame non può essere eliminato. Vi è però il modo, pur non modificando il legame, di modificare il valore associato ad una variabile, come vedremo nel Paragrafo 14.3.

## 8.2.2 L’assegnamento

L’*assegnamento* è il comando di base che permette di modificare il valore delle variabili modificabili, e quindi dello stato, in un linguaggio imperativo. Si tratta di un comando apparentemente molto semplice; tuttavia, come vedremo, vi sono varie sottigliezze da considerare nei diversi linguaggi di programmazione.

Vediamo innanzitutto il caso probabilmente più familiare al lettore, ossia quello di un linguaggio imperativo che usa le variabili modificabili e nel quale l’assegnamento sia considerato solo come un comando (e non anche come un’espressione). Un tale caso, ad esempio, è quello di Pascal dove possiamo scrivere

```
x := 2
```

per indicare che alla variabile  $x$  è assegnato il valore 2. L’effetto di un tale comando è che, dopo la sua esecuzione, il contenitore associato alla variabile (di nome  $x$ ) conterrà il valore 2 al posto del valore che vi era precedentemente contenuto. Si noti che questo è un “effetto collaterale”, dato che la valutazione del comando non ha, di per sé, restituito alcun valore; tuttavia, ogni accesso ad  $x$  nel seguito del programma restituirà il valore 2 e non quello precedentemente memorizzato.

Si consideri adesso il seguente comando:

```
x := x+1
```

il cui effetto, come sappiamo, è quello di assegnare alla variabile  $x$  il suo valore precedente, incrementato di 1. Osserviamo qui il diverso uso del nome  $x$  della variabile nei due operandi dell’operatore di assegnamento: la  $x$  che compare a sinistra del simbolo  $:=$  è usata per indicare il contenitore (la locazione), al cui interno troviamo il valore della variabile; l’occorrenza di  $x$  a destra di  $:=$  indica il valore all’interno del contenitore. Questa importante distinzione è formalizzata nei linguaggi di programmazione usando due diversi insiemi di valori: gli *l-valori* (o valori sinistri, “l” sta per left) sono quei valori che sostanzialmente indicano locazioni e quindi sono i valori di espressioni che possono stare alla sinistra di un comando di assegnamento. Gli *r-valori* (o valori destri, “r” sta per right) sono invece i valori che possono essere contenuti nelle locazioni, e quindi sono i valori di espressioni che possono stare alla destra di un comando di assegnamento. In generale dunque, il comando di assegnamento ha la sintassi di un operatore binario in forma infissa

```
exp1 OpAss exp2
```

dove *OpAss* indica il simbolo usato nel particolare linguaggio per indicare l’assegnamento ( $:=$  in Pascal,  $=$  in C, FORTRAN, Snobol e Java,  $\leftarrow$  in APL ecc.). Il significato di un tale comando (nel caso della variabile modificabile) è il seguente: calcola lo *l*-valore di *exp1*, determinando così un contenitore *loc*; calcola lo *r*-valore di *exp2* e modifica il contenuto di *loc* sostituendo il valore così calcolato a quello precedente<sup>3</sup>. Quali espressioni possano denotare (nell’opportuno contesto a sinistra di un assegnamento) un *l*-valore dipende dal linguaggio di programmazione; casi usuali sono le variabili, gli elementi di array, i campi di record (si osservi, di conseguenza, che il calcolo di un *l*-valore può essere arbitrariamente complesso, perché potrebbe coinvolgere delle chiamate a funzione, per esempio nella determinazione dell’indice di un array).

In alcuni linguaggi, ad esempio C, l’assegnamento è considerato come un operatore la cui valutazione, oltre a produrre un effetto collaterale, restituisce anche lo *r*-valore calcolato. Così, se in C scriviamo

```
x = 2;
```

la valutazione di un tale comando, oltre ad assegnare a  $x$  il valore 2, restituisce il valore 2. Dunque in C possiamo anche scrivere

```
y = x = 2;
```

con il significato di

```
(y = (x = 2));
```

che assegna sia ad  $x$  che ad  $y$  il valore 2. In C, come in altri linguaggi, sono presenti anche altri operatori di assegnamento, con lo scopo sia di migliorare la leggibilità del codice, sia di evitare alcuni effetti collaterali non previsti. Riconsideriamo l’esempio di incremento di una variabile:

<sup>3</sup> Alcuni linguaggi (p.e. Java) prevedono che il membro sinistro sia valutato prima del membro destro; altri (p.e. C) lasciano questa decisione all’implementazione.

```
x = x+1;
```

Questo comando, a meno di ottimizzazioni del compilatore, in linea di principio richiede due accessi alla variabile *x*: una per determinare lo l-valore, una per prelevarne lo r-valore. Se da un punto di vista di efficienza si tratta di un problema, tutto sommato, non grave (e facilmente ottimizzabile dal compilatore), vi è una questione ben più significativa connessa agli effetti collaterali. Consideriamo infatti il codice

```
b = 0;
a[f(3)] = a[f(3)]+1;
```

dove *a* è un vettore ed *f* una funzione definita come segue

```
int f (int n){
    if b == 0{
        b=1;
        return 1;
    }
    else return 2;
}
```

e tale che il riferimento non locale *a* *b* nel corpo di *f* si riferisce alla stessa variabile *b* che viene azzerata nel frammento precedente. Dato che *f* modifica la variabile non locale *b*, è evidente che l'assegnamento

```
a[f(3)] = a[f(3)]+1
```

non ha l'effetto di incrementare il valore dell'elemento *a* [*f*(3)] dell'array, come forse avremmo voluto che fosse (ma ha l'effetto di assegnare il valore di *a*[1]+1 ad *a*[2] qualora la valutazione del membro destro dell'assegnamento preceda la valutazione del membro sinistro). Si osservi, d'altra parte, che il compilatore non può ottimizzare il calcolo degli r-valori, perché il programmatore potrebbe aver desiderato questo comportamento apparentemente anomalo.

Per ovviare a questo problema possiamo ovviamente usare una variabile auxiliaria e scrivere

```
int j = f(3);
a[j] = a[j]+1;
```

Così facendo, tuttavia, oscuriamo il codice e introduciamo una variabile poco espressiva. Per ovviare a tutto ciò, linguaggi come C mettono a disposizione opportuni operatori di assegnamento, che permettono di scrivere

```
a[f(3)] += 1;
```

che, appunto, incrementa della quantità presente a destra dell'operatore *+=* lo r-valore della espressione presente a sinistra e quindi assegna il risultato alla locazione ottenuta come l-valore dell'espressione di sinistra. Vi sono molti comandi di assegnamento specifici, analoghi al precedente. La seguente è una lista, non completa, dei comandi di assegnamento presenti in C, con la relativa descrizione:

- *x = Y*: assegna lo r-valore di *Y* alla locazione ottenuta come l-valore di *x* e restituisce lo r-valore di *X*;

- *x += Y* (risp. *x -= Y*): incrementa (risp. decrements) *x* della quantità data dallo r-valore di *Y* e restituisce il nuovo r-valore;
- *++x* (risp. *--x*): incrementa (risp. decrements) *x* e restituisce il nuovo r-valore di *x*;
- *x++* (risp. *x--*): restituisce lo r-valore di *x* e incrementa (decrements) *x*.

Vediamo adesso in che senso il modello a riferimento per le variabili è diverso dalla tradizionale variabile modificabile. In un linguaggio con modello a riferimento (per esempio CLU e, come vedremo, in certi casi Java) un assegnamento del tipo

*x=e*

fa sì che *x* sia un riferimento per l'oggetto ottenuto dalla valutazione dell'espressione *e*. Si noti che questo è diverso dal copiare il valore di *e* nella locazione associata a *x*. La differenza diviene chiara se consideriamo un assegnamento tra due variabili nel modello a riferimento:

*x=y*

Dopo un tale assegnamento *x* e *y* sono due riferimenti per lo stesso oggetto. Nel caso in cui tale oggetto sia modificabile (ad esempio un record o un array), una modifica fatta attraverso la variabile *x* diviene visibile attraverso la variabile *y*, e viceversa. In questo modello, dunque, le variabili si comportano in modo simile alle variabili di tipo puntatore dei linguaggi che hanno questo tipo di dati. Come vedremo meglio nel Paragrafo 10.4.5, un valore di tipo puntatore non è altro che la locazione (ossia, sostanzialmente, un indirizzo di una zona di memoria) di un particolare dato. In molti linguaggi che hanno il tipo puntatore i valori di tale tipo possono essere manipolati esplicitamente (con diversi problemi conseguenti, si veda ancora il Capitolo 10). Nel caso del modello a riferimento, invece, tali valori possono essere manipolati solo implicitamente, mediante assegnamenti tra variabili. Java (che non ha puntatori) adotta il modello a riferimento delle variabili per tutti i tipi classe, mentre adotta le variabili modificabili tradizionali per i tipi primitivi (interi, reali in virgola mobile, booleani e caratteri).

Nel seguito, se non diversamente specificato, quando parliamo di variabile intendiamo la variabile modificabile.

### 8.3 Comandi per il controllo di sequenza

L'assegnamento costituisce il comando di base nei linguaggi imperativi (e in quelli dichiarativi "non puri"), quello che esprime il passo di computazione elementare. I rimanenti comandi servono per definire il controllo di sequenza, ossia servono a specificare l'ordine con cui le modifiche di stato espresse dagli assegnamenti devono essere effettuate. Tali comandi possono essere divisi in tre categorie:

**I comandi per il controllo di sequenza esplicito:** questi sono il comando sequenziale ed il goto. Consideriamo inoltre in questa categoria anche il comando composto, che permette di considerare più comandi come un unico comando.

## Ambiente e memoria

Nel Capitolo 2 abbiamo definito la semantica di un comando facendo riferimento ad una semplice nozione di stato, definito come una funzione che associa ad ogni variabile presente nel programma il valore assunto da tale variabile. Una tale nozione di stato, pur adeguata per gli scopi didattici del mini-linguaggio visto, non è sufficiente quando si voglia descrivere la semantica di un linguaggio di programmazione reale, che usi variabili modificabili e assegnamento. Abbiamo infatti già visto nel Capitolo 6 (e lo vedremo meglio nei prossimi due capitoli), che i meccanismi di passaggio dei parametri ed i puntatori possono creare facilmente situazioni nelle quali due nomi diversi, ad esempio  $x$  e  $y$ , denotano la stessa variabile, ossia la stessa locazione di memoria. Una tale situazione di aliasing non può essere descritta da una semplice funzione  $Stato : Nomi \rightarrow Valori$  perché con una simile funzione non riusciamo ad esprimere il fatto che una modifica del valore associato alla (variabile denotata da)  $x$  si riflette anche sul valore associato a  $y$ . Per esprimere correttamente il significato delle variabili modificabili si usano dunque due funzioni separate. La prima, detta *ambiente*, sostanzialmente corrisponde alla nozione di ambiente che abbiamo introdotto nel Capitolo 6: si tratta di una funzione  $Ambiente : Nomi \rightarrow Valori\ Denotabili$  che associa ai nomi gli oggetti che essi denotano. L'insieme (o, come si dice nel gergo della semantica, il dominio) dei nomi spesso coincide con quello degli identificatori. Il dominio *Valori Denotabili* invece include tutti i valori ai quali si può dare un nome: quali siano questi valori dipende dal linguaggio di programmazione ma se il linguaggio prevede le variabili modificabili allora tale dominio include sicuramente le locazioni di memoria.

I valori associati alle locazioni sono invece espressi da una funzione  $Memoria : Locazioni \rightarrow Valori\ Memorizzabili$  che (informalmente) associa ad ogni locazione il valore in essa memorizzato. Anche in questo caso, cosa sia un valore memorizzabile dipende dal linguaggio specifico.

Quindi, quando diciamo che “nello stato corrente la variabile  $x$  ha valore 5”, formalmente intendiamo dire che abbiamo un ambiente  $\rho$  ed una memoria  $\sigma$  tali che  $\sigma(\rho(x)) = 5$ . Si noti che quando una variabile è intesa come l-valore ci interessa solo la locazione denotata dal nome, e quindi ci serve solo l’ambiente, mentre quando è intesa come r-valore ci serve anche la memoria. Ad esempio, dato un ambiente  $\rho$  ed una memoria  $\sigma$  l’effetto del comando  $x=Y$  è produrre un nuovo stato nel quale il valore  $\sigma(\rho(y))$  è associato a  $\rho(x)$ . Ricordiamo, per completezza, che un terzo dominio di valori importante nella semantica dei linguaggi è costituito dai *Valori Esprimibili*, ossia da quei valori che possono essere il risultato della valutazione di un’espressione complessa.

**Comandi condizionali (o di selezione):** si tratta di quei comandi che permettono di specificare alternative su come proseguire la computazione, in dipendenza dal verificarsi di determinate condizioni.

**Comandi iterativi:** questi permettono di ripetere un determinato comando per un numero di volte che può essere predefinito, oppure può dipendere dal verifi-

carsi di specifiche condizioni.

Vediamo in dettaglio queste tipologie di comandi.

### 8.3.1 Comandi per il controllo di sequenza esplicito

**Comando sequenziale** Il *comando sequenziale*, indicato in molti linguaggi da un “;” permette di specificare in modo diretto l’esecuzione sequenziale di due comandi: se scriviamo

$C1 ; C2$

l’esecuzione di  $C2$  inizia dopo che è terminata quella di  $C1$ . Nei linguaggi in cui la valutazione di un comando restituisca anche un valore, il valore restituito dalla valutazione del comando sequenziale è quello del secondo argomento.

Ovviamente possiamo indicare anche una sequenza di comandi, quale

$C1 ; C2 ; \dots ; Cn$

con l’assunzione implicita che l’operatore “;” associ a sinistra.

**Approfondimento** 8.1

**Comando composto** Nei linguaggi imperativi moderni, come abbiamo già visto nel Capitolo 6, è possibile raggruppare una sequenza di comandi in un *comando composto* usando opportuni delimitatori quali quelli di Algol

`begin`

`...`

`end`

oppure quelli di C

`{`

`...`

`}`

Un tale comando composto, detto anche blocco, può essere usato in un qualsiasi contesto nel quale ci si aspetta un comando semplice e quindi, in particolare, può essere usato all’interno di un altro comando composto per creare così strutture annidate di arbitraria complessità.

**Goto** Un posto particolare nel panorama dei comandi per il controllo di sequenza esplicito è rappresentato dal *goto*. Presente in varie versioni (condizionali oppure dirette) sin dai primi linguaggi di programmazione, tale comando è ispirato direttamente alle istruzioni di salto dei linguaggi assembly e quindi al modello di controllo di sequenza della macchina hardware. L’esecuzione del comando

`goto A`

### Linguaggi imperativi e dichiarativi

I valori denotabili, memorizzabili ed esprimibili, anche se hanno un'intersezione non vuota, sono insiemi concettualmente distinti. In effetti, molte differenze importanti fra i vari linguaggi dipendono da come sono definiti questi domini. Ad esempio, le funzioni sono denotabili ma non esprimibili in Pascal, mentre sono esprimibili in LISP e in ML. Una differenza particolarmente importante fra i vari linguaggi riguarda la presenza dei valori memorizzabili e della funzione semantica *Memoria* che abbiamo visto nel riquadro precedente. Infatti, in modo un po' sintetico, possiamo classificare come *imperativi* quei linguaggi che hanno sia la funzione ambiente che la memoria, mentre sono *dichiarativi* quelli che hanno solo l'ambiente. I linguaggi imperativi, pur essendo linguaggi di alto livello, sono ispirati alla struttura fisica dei calcolatori: per essi è fondamentale il concetto di memoria (o stato), inteso come insieme delle associazioni fra locazioni di memoria e valori che in tali locazioni sono memorizzati. Un programma, secondo questo paradigma, è un insieme di comandi "imperativi" e la computazione consiste in una sequenza di passi che modificano lo stato, usando come comando elementare per questo scopo l'assegnamento. La terminologia "imperativo" qui ha a che fare con il linguaggio naturale: così come, con una frase imperativa, diciamo "taglia quella mela" per esprimere un comando, così con un comando imperativo possiamo dire "assegna ad x il valore 1". La maggior parte dei linguaggi di programmazione di uso comune è di tipo imperativo (FORTRAN, ALGOL, Pascal, C ecc.).

I linguaggi dichiarativi (funzionali e logici) sono stati introdotti con lo scopo di offrire un paradigma di programmazione di più alto livello, vicino alla notazione della matematica e della logica, astraendo dalle caratteristiche della macchina fisica sulla quale il programma viene eseguito. Nei linguaggi dichiarativi (almeno nelle versioni "pure") non esistono comandi che modificano lo stato, dato che non esistono né le variabili modificabili né la funzione semantica memoria. I programmi sono costituiti da un insieme di dichiarazioni (da qui il nome) di funzioni o di relazioni, che definiscono nuovi valori. A seconda del meccanismo elementare utilizzato per specificare le caratteristiche del risultato, i linguaggi dichiarativi vengono suddivisi in due classi: i linguaggi funzionali e i linguaggi logici. Nei primi, la computazione consiste nella valutazione delle funzioni definite dal programma secondo regole di tipo matematico (sostanzialmente composizione e applicazione). Nei secondi invece la computazione è basata sulla deduzione della logica del prim'ordine. Ricordiamo che, nella pratica, esistono linguaggi funzionali e logici "non puri" che hanno anche caratteristiche imperative (in particolare, hanno l'assegnamento). Vedremo sia i linguaggi funzionali che quelli logici in due prossimi capitoli.

trasferisce il controllo al punto del programma nel quale è presente l'etichetta A (al solito, i vari linguaggi differiscono in cosa siano esattamente le etichette ma tali differenze non sono qui rilevanti).

Nonostante l'apparente semplicità e naturalezza, il comando `goto` è stato al centro di un acceso dibattito a partire dagli anni '70 (si veda, ad esempio, il famoso articolo di Dijkstra citato nella nota bibliografica) e, dopo circa 30 anni di discussioni, possiamo dire che i detrattori hanno avuto la meglio sui sostenitori di questo comando. Per chiarire il senso di una tale discussione osserviamo innanzitutto che il `goto` non è essenziale per l'espressività di un linguaggio di programmazione: un teorema di Böhm e Jacopini infatti dimostra che un qualsiasi programma può essere "tradotto" in uno, equivalente, che non usi il `goto` (la formulazione del teorema, ovviamente, è molto più precisa di quanto qui riportato). Questo risultato tuttavia non dà indicazioni definitive. Se da un lato si può sostenere che, in linea di principio, il `goto` non serve, dall'altro si potrebbe obiettare (come fu fatto a suo tempo) che proprio questo risultato mostra che è lecito usare il `goto` nei programmi: se proprio si vuole eliminare tale comando infatti, questo può essere fatto successivamente, mediante la trasformazione di Böhm e Jacopini (che, oltretutto, distrugge completamente la struttura del programma originale).

Il nocciolo della questione in realtà non è di tipo teorico ma è di natura pragmatica. Usando il `goto` si può facilmente scrivere codice che diventa ben presto incomprensibile e che rimane tale anche in caso di una eventuale eliminazione successiva del `goto`. Si pensi, ad esempio, ad un programma di una certa dimensione dove siano inseriti dei salti fra punti distanti qualche migliaio di linee di codice. Oppure si pensi all'uscita da sottoprogrammi realizzata, mediante `goto`, in punti diversi a seconda delle condizioni che si verifichino. Questi ed altri usi arbitrari di questo costrutto rendono il codice difficilmente comprensibile, e quindi di difficile modifica, correzione e manutenzione, con ovvie conseguenze negative anche in termini di costi. A tutto ciò si aggiunge che il `goto`, col suo modo primitivo di trasferire il controllo, mal si accorda con molti altri meccanismi presenti nei linguaggi di alto livello. Cosa succede, ad esempio, se si salta *all'interno* di un blocco? Quando e come viene inizializzato il RdA di quel blocco in modo che tutto funzioni correttamente?

Se il `goto` venisse usato in modo estremamente controllato, localmente a piccole porzioni di codice, gran parte di questi svantaggi scomparirebbero. Tuttavia, i casi in cui può essere utile usare questo comando, quali uscita da cicli, ritorno da sottoprogrammi, gestione delle eccezioni, nei moderni linguaggi di programmazione possono essere gestiti da opportuni costrutti specifici. Si può quindi affermare che nei linguaggi di alto livello moderni il `goto` è un costrutto il cui uso va scomparendo. Java è il primo linguaggio commerciale ad averlo bandito del tutto dall'insieme dei comandi ammissibili.

**Altri comandi di controllo di sequenza** Se il `goto` è pericoloso nella sua forma generale, vi sono suoi usi locali e limitati che riescono utili in determinate circostanze. Molti linguaggi mettono a disposizione forme limitate di salto proprio per venire incontro a queste necessità pragmatiche senza dover ricorrere alla forza bruta di un `goto`. Tra questi comandi troviamo (al solito in varie forme nei vari linguaggi) costrutti quali `break` (per terminare l'esecuzione di un'iterazione, o di un `case`, o, in taluni linguaggi, del blocco nel quale si trova), `continue` (per

terminare l'iterazione corrente di un comando iterativo e forzare l'inizio di quella successiva) o `return` (per terminare la valutazione di una funzione, restituendo il controllo al chiamante e talvolta passando anche un valore).

Un controllo di sequenza più elaborato è infine possibile con il meccanismo delle eccezioni, che tratteremo in modo approfondito nel prossimo capitolo, al Paragrafo 9.3.

### 8.3.2 Comandi condizionali

I comandi condizionali, o di selezione, esprimono un'alternativa fra due o più possibili prosecuzioni della computazione sulla base di opportune condizioni logiche. Possiamo dividere i comandi condizionali in due gruppi.

**If** Il comando `if`, introdotto originariamente nel linguaggio ALGOL 60, è presente in quasi tutti i linguaggi imperativi, ed anche in alcuni linguaggi dichiarativi, in varie forme sintattiche che comunque, nella sostanza, si possono ricondurre alla forma

```
if Bexp then C1 else C2
```

dove `Bexp` è un'espressione booleana, mentre `C1` e `C2` sono comandi. Informalmente, la semantica di un tale comando esprime un'alternativa nel proseguimento della computazione, sulla base della valutazione dell'espressione `Bexp`: se tale valutazione restituisce vero si esegue il comando `C1`, altrimenti si esegue il comando `C2`. Questo comando è spesso presente anche nella forma senza il ramo `else`

```
if Bexp then C1
```

e in questo caso, se la condizione è falsa, non si esegue alcun comando (e si passa a quello successivo al comando condizionale). Come abbiamo visto nel Capitolo 2 la presenza di `if` annidati come nel comando

```
if Bexp1 if Bexp2 then C1 else C2
```

può porre dei problemi di ambiguità, che possono essere risolti usando un'opportuna grammatica che descriva formalmente la regola adottata dal linguaggio (ad esempio, il ramo `else` appartiene allo `if` più interno tra tutti quelli ai quali potrebbe appartenere, che è la regola che abbiamo visto per Java ed è seguita da quasi tutti i linguaggi). Per evitare problemi di ambiguità alcuni linguaggi usano un "terminatore" che indichi dove termina il comando condizionale, come ad esempio in

```
if Bexp then C1 else C2 endif
```

Inoltre, in alcuni casi, invece che usare una lista di `if then else` annidati si usa un comando `if` esplicito con più rami, analogo al seguente

```
if Bexp1 then C1
elseif Bexp2 then C2
...
```

```
elseif Bexpn then Cn
else Cn+1
endif
```

L'implementazione del comando condizionale non pone problemi, sfruttando le istruzioni di test e salto della macchina fisica sottostante<sup>4</sup>. La valutazione dell'espressione booleana può usare la tecnica del corto-circuito già vista in precedenza.

**Case** Il comando `case` è una specializzazione del comando `if` con più rami che abbiamo appena discusso. Nella sua forma più semplice può essere scritto come segue:

```
case Exp of
    label1: C1;
    label2: C2;
    ...
    labeln: Cn;
else Cn+1
```

dove `Exp` è un'espressione il cui valore è di un tipo compatibile con quello delle etichette `label1`, ..., `labeln`, mentre `C1`, ..., `Cn+1` sono comandi. Ogni etichetta può essere rappresentata da una o più costanti e le costanti usate per etichette diverse sono diverse fra di loro. I tipi permessi per le etichette, così come il loro formato variano a seconda dei linguaggi: in molti casi sono permessi tutti i tipi discreti (vedi il Paragrafo 10.3), inclusi enumerazioni ed intervalli. Così, ad esempio, possiamo usare le costanti 2 e 4 per indicare un'etichetta, ma in alcuni linguaggi possiamo anche scrivere 2..4, per indicare sia il valore 2 che il valore 4, oppure 2..4, per indicare tutti i valori compresi fra 2 e 4 (estremi inclusi).

Il significato di un tale comando, come detto, è analogo a quello di un `if` con più rami: viene valutata l'espressione `Exp` e quindi viene eseguito il comando presente nell'unico ramo la cui etichetta include il valore ottenuto. Il ramo `else` viene eseguito qualora non ci sia alcun ramo la cui etichetta soddisfi la condizione suddetta.

Anche se, evidentemente, quello che si può fare con un `case` lo si può sicuramente esprimere con una serie di `if` annidati, molti linguaggi includono una forma di `case` tra le loro istruzioni, sia per migliorare la leggibilità del codice, sia perché è possibile compilare un `case` assai più efficientemente di una lunga serie di `if` annidati. Il `case` è infatti implementato a livello assembly usando un vettore di celle contigue detto "tabella di salto" (*jump table*): ogni elemento della tabella contiene l'*indirizzo* della prima istruzione dei comandi dei vari rami. L'uso di una tale tabella è mostrato nella Figura 8.3 dove, per semplicità, si assume che le etichette `label1`, ..., `labeln` siano le costanti consecutive 0, 1, ..., n-1. Come risulta chiaro dalla figura, innanzitutto viene valutata l'espressione che compare

<sup>4</sup> A livello di linguaggio assembly (e quindi a livello del linguaggio della macchina hardware) sono presenti operazioni di salto (*jump*), condizionali o meno, analoghe al `goto` dei linguaggi di alto livello.

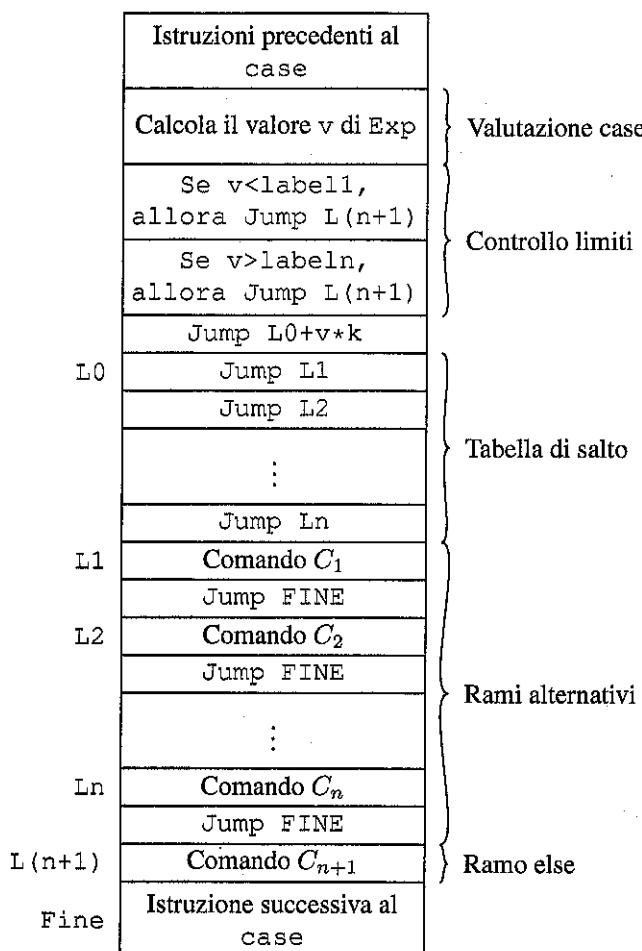


Figura 8.3 Implementazione del case.

come argomento del `case`; il valore ottenuto è quindi usato come offset (indice) per calcolare la posizione, nella tabella di salto, dell'istruzione che permette il salto al comando del ramo scelto. Il salto al ramo `else` è invece fatto sulla base del controllo esplicito dei limiti delle etichette. L'estensione di questo meccanismo al caso generale in cui si usino anche etichette con insiemi e intervalli è semplice.

Questo meccanismo di implementazione del `case` permette una maggiore efficienza rispetto all'uso di una serie di `if` annidati. Usando la tabella di salto, una volta calcolato il valore dell'espressione, con due istruzioni di salto si arriva al codice del comando da eseguire. Usando `if` annidati, invece, in presenza di

$n$  rami alternativi si devono valutare (nel caso peggiore di `if` sbilanciati)  $O(n)$  condizioni ed effettuare  $O(n)$  salti prima di giungere al comando che ci interessa. Lo svantaggio nell'uso della tabella di salto è che, essendo una struttura lineare dove elementi contigui corrispondono a valori successivi delle etichette, può consumare molto spazio nel caso in cui i valori delle etichette siano dispersi su di un intervallo molto ampio o in cui le singole etichette usino valori di tipo intervallo  $n_1 \dots n_2$  molto ampi. In questi casi si possono usare metodi alternativi per il calcolo dell'indirizzo di salto, quali test sequenziali (cioè l'implementazione con `if` annidati), tecniche di hashing o anche metodi basati su ricerca binaria.

I diversi linguaggi esibiscono differenze significative nel comando `case`. In C, ad esempio, il comando `switch` ha la seguente sintassi (presente anche in C++ e Java):

```
switch (Exp) corpo
```

dove `corpo` può essere un qualunque comando; in generale, tuttavia, il corpo è costituito da un blocco nel quale alcuni comandi possono essere etichettati, cioè essere della forma

```
case label : comando
```

mentre l'ultimo comando del blocco è della forma

```
default : comando
```

Al momento dell'esecuzione l'espressione `Exp` è valutata ed il controllo è trasferito al comando la cui etichetta coincide col valore di `Exp`; se non ci sono etichette con tale valore, il controllo passa al comando con etichetta `default`; se non c'è un tale comando, il controllo passa alla prima istruzione che segue lo `switch`. Si osservi che, una volta che è stato selezionato un ramo dello `switch`, il controllo "fluisce" nei rami successivi. Per ottenere un costrutto dalla semantica analoga a quella del `case` che abbiamo discusso, occorre inserire dei trasferimenti esplicativi alla fine del blocco, usando un `break`:

```
switch (Exp) {
    case label1: C1 break;
    case label2: C2 break;
    ...
    case labeln: Cn break;
    default: Cn+1 break;
}
```

Si osservi che in uno `switch` il valore restituito dalla valutazione dell'espressione può anche essere assente nelle etichette, nel qual caso l'intero comando non ha alcun effetto. Infine nelle etichette non sono ammessi liste o range di valori. Questa però non è una vera limitazione, dato che le liste di valori possono essere realizzate usando la possibilità di passare da un ramo a quello successivo in assenza del comando `break`. Se ad esempio scriviamo

```
switch (Exp) {
    case 1:
    case 2: C2 break;
    case 3: C3 break;
```

```
default: C4 break;
}
```

nel caso in cui il valore di `Exp` sia 1, dato che nel ramo corrispondente non è presente il comando `break`, dal ramo `case 1` la valutazione passa al ramo `case 2` e dunque è come se avessimo usato la lista di valori 1, 2 per l'etichetta di `C2`.

### 8.3.3 Comandi iterativi

I comandi sin qui visti, escluso il `goto`, permettono solo di esprimere computazioni finite, la cui lunghezza massima è determinabile staticamente in base alla lunghezza del testo del programma<sup>5</sup>. Un linguaggio che avesse solo tali comandi risulterebbe di espressività molto limitata: non sarebbe certamente Turing completo (si ricordi il Paragrafo 5.2.1), in quanto non permetterebbe di esprimere tutti i possibili algoritmi (si pensi, ad esempio, anche solo alla scansione di un vettore di  $n$  posizioni, con  $n$  non noto a priori).

Nei linguaggi di basso livello per raggiungere il potere espressivo necessario ad esprimere tutti i possibili algoritmi si usano opportune istruzioni di salto, che permettono di ripetere gruppi di istruzioni “saltando” all’indietro nel codice. Nei linguaggi di alto livello, visto che per i motivi già detti si vogliono evitare comandi simili al `goto`, per raggiungere un tale obiettivo si usano due meccanismi di base: l'*iterazione strutturata* e la *ricorsione*. Il primo, trattato in questo paragrafo, è più familiare nell’ambito dei linguaggi imperativi (che comunque quasi sempre permettono anche la ricorsione): opportuni costrutti linguistici (che possiamo ancora una volta immaginare come versioni speciali del comando di salto) consentono di realizzare in modo compatto dei cicli nei quali si ripetono, o *iterano*, dei comandi. A livello linguistico si distinguono l'*iterazione indeterminata*, realizzata da costrutti che permettono di iterare fino al verificarsi di una data condizione, e l'*iterazione determinata*, realizzata mediante costrutti che permettono di iterare per un numero prefissato di volte. Presenteremo queste due forme di iterazione nel seguito, partendo da quella indeterminata; come vedremo, si tratta di due meccanismi con una sostanziale diversità di potere espressivo.

La ricorsione, che considereremo nel prossimo paragrafo, permette invece di esprimere dei cicli in modo implicito, ammettendo la possibilità per una funzione (o procedura) di richiamare se stessa, ripetendo quindi il proprio corpo un numero arbitrario di volte. L’uso della ricorsione è più comune nell’ambito dei linguaggi dichiarativi (in molti linguaggi funzionali e logici infatti non esiste iterazione).

**Iterazione indeterminata** L’iterazione indeterminata, o iterazione controllata logicamente, è realizzata da costrutti linguistici costituiti da due parti: una *condizione* (o *guardia*) del ciclo ed un *corpo*, costituito da un comando (eventualmente

<sup>5</sup>Si può facilmente vedere che la lunghezza massima della computazione è una funzione lineare nella dimensione del programma.

composto). In esecuzione il corpo viene ripetutamente eseguito sino a quando la guardia diviene falsa (o vera, a seconda dei costrutti).

Nella sua forma più comune questo tipo di iterazione assume la forma del comando `while`, introdotto originariamente in ALGOL:

```
while (Bexp) do C
```

Il significato di questo comando è noto: (1) viene valutata l'espressione booleana `Bexp`; (2) se tale valutazione restituisce vero si esegue il comando `C` e si torna a (1), altrimenti il comando `while` termina.

In alcuni linguaggi sono presenti anche comandi che controllano la condizione *dopo* l'esecuzione del comando (che quindi è sempre eseguito almeno una volta). Questo costrutto è ad esempio presente in Pascal con la seguente forma

```
repeat C until Bexp
```

che non è altro che un'abbreviazione per

```
C;  
while not Bexp do C
```

(`not Bexp` qui indica la negazione dell'espressione `Bexp`). In C un costrutto analogo è `do`:

```
do C while (Bexp)
```

che tuttavia corrisponde a

```
C;  
while Bexp do C
```

(si osservi che la guardia non è negata come nel caso di `repeat`).

Il costrutto `while` è di semplice implementazione, dato che corrisponde direttamente ad un ciclo realizzabile sulla macchina fisica usando l'istruzione di salto condizionato. La semplicità di implementazione non deve trarre in inganno circa la potenza di questo costrutto: la sua aggiunta ad un linguaggio di programmazione che contenga solo comandi di assegnamento e condizionali rende subito il linguaggio Turing completo. Il nostro mini-linguaggio del Capitolo 2 è dunque Turing completo (permette di esprimere tutte le funzioni calcolabili). Lo stesso non accade con l'iterazione determinata, che trattiamo nel prossimo paragrafo.

**Iterazione determinata** L’iterazione determinata (detta talvolta anche iterazione controllata numericamente) è realizzata da costrutti linguistici più complessi di quelli dell’iterazione indeterminata ed ha anche una semantica apparentemente più elaborata. Le forme con le quali compare nei linguaggi sono molteplici e non sempre “pure”, come vedremo tra poco. Il modello che adottiamo in questa discussione è quello di ALGOL, seguito poi da molti altri linguaggi della stessa famiglia (ma *non* da C o Java).

L’iterazione determinata è realizzata mediante una qualche variante del comando `for` che, senza voler far riferimento ad una sintassi specifica, può essere descritto come:

```
for I = inizio to fine by passo do
    corpo
```

dove  $I$  è una variabile, detta *indice*, o contatore, o *variabile di controllo*; *inizio* e *fine* sono espressioni (per semplicità possiamo supporre di tipo intero; in generale devono essere di un tipo discreto); *passo* è una costante (a tempo di compilazione) intera diversa da zero; *corpo* è il comando che si vuole iterare. Un tale costrutto, nella versione "pura" che qui stiamo descrivendo, è soggetto all'importante vincolo di semantica statica che la variabile di controllo non può essere modificata (né esplicitamente, né implicitamente) nel corso dell'esecuzione del corpo.

La semantica del costrutto di iterazione determinata può essere descritta informalmente come segue, supponendo *passo* positivo:

1. vengono valutate le espressioni *inizio* e *fine* ed i valori così determinati sono "congelati" in opportune variabili (indisponibili al programmatore), che possiamo indicare, rispettivamente, con *inizio\_save* e *fine\_save*;
2.  $I$  viene inizializzata con il valore di *inizio\_save*;
3. se il valore di  $I$  è strettamente maggiore del valore di *fine\_save*, termina l'esecuzione del *for*;
4. si esegue *corpo* e si incrementa  $I$  del valore di *passo*;
5. si torna a (3).

Nel caso di *passo* negativo, il test del punto (3) verifica, invece, che  $I$  sia strettamente minore di *fine\_save*.

Vale la pena di sottolineare il senso del punto (1) qui sopra e del vincolo che la variabile di controllo non sia modificata nel corpo: il loro effetto combinato è quello di *determinare* il numero di volte che il corpo verrà eseguito *prima* che il ciclo abbia inizio. Tale numero è dato dalla quantità *ic* (*iteration count*, contatore d'iterazione) definita come

$$ic = \left\lfloor \frac{\text{fine} - \text{inizio} + \text{passo}}{\text{passo}} \right\rfloor$$

se questa quantità è positiva, mentre è 0 altrimenti. Si osservi, infine, come non vi sia modo alcuno di ottenere un ciclo infinito con questo costrutto.

Anche nel caso di questo comando vi sono notevoli differenze nei vari linguaggi, sia nella sintassi che nella sua semantica. In primo luogo, non tutti i linguaggi prevedono la non modificabilità della variabile di controllo e/o il congelamento delle espressioni. A rigor di termini, in tal caso non siamo più in presenza di un costrutto di iterazione determinata, perché viene meno la possibilità di calcolare una volta per tutte il valore *ic*. È comune, tuttavia, continuare a parlare di iterazione determinata anche quando il linguaggio non garantisca la determinatezza, ma questa sia ottenuta, in uno specifico ciclo, dal programmatore (non modificando direttamente né indirettamente la variabile di controllo e le espressioni di *inizio*, *fine* e *passo*). Anche diversi altri aspetti costituiscono importanti differenze tra linguaggi; ne menzioniamo quattro.

**Numero di iterazioni** Secondo la semantica che abbiamo dato, in caso di *passo* positivo, se inizialmente il valore di *inizio* è (strettamente) maggiore del valore di *fine*, *corpo* non viene eseguito neanche una volta. Questo, anche se valido per la maggior parte dei casi, non avviene in tutti i linguaggi: alcuni eseguono il test in (3) dopo aver eseguito *corpo*.

**Passo** La richiesta che *passo* sia una costante (non nulla) è necessaria per determinarne staticamente il segno, affinché il compilatore possa generare il codice opportuno per il test in (3). Alcuni linguaggi (tra cui Pascal e Ada) usano speciale sintassi per indicare che il *passo* è negativo, ad esempio usando *downto* o *reverse* al posto di *to*. Altri linguaggi, quali ad esempio alcune versioni di FORTRAN, evitano l'uso di una sintassi diversa per il *passo* negativo in quanto implementano il *for* usando direttamente il contatore d'iterazione invece che il test tra *I* e *fine*: si calcola il valore *ic* e se questo valore risulta positivo si usa tale quantità per controllare il ciclo, decrementandola di 1 fino a raggiungere il valore 0; se invece *ic* ha un valore negativo o eguale a 0 il ciclo non è mai ripetuto; Dall'uso di questa tecnica implementativa deriva il nome di iterazione controllata numericamente.

**Valore finale dell'indice** Un altro aspetto delicato riguarda il valore assunto dalla variabile di controllo *I* dopo la fine del ciclo. In molti linguaggi *I* è una variabile visibile anche fuori dal ciclo: l'approccio più naturale sembrerebbe quello di considerare come valore di *I* l'ultimo valore che gli è stato assegnato nel corso della valutazione del *for* stesso (nel caso in cui il ciclo termini normalmente e il *passo* sia positivo, l'ultimo valore assegnato all'indice *I* è il primo valore maggiore di *fine*). Quest'approccio, tuttavia, può generare ambiguità o errori di tipo. Supponiamo, ad esempio, che *I* sia dichiarata di tipo intervallo *1..10* (da 1 a 10); se usiamo un comando

```
for I = 1 to 10 by 1 do
    corpo
```

l'ultimo valore assegnato a *I* dovrebbe essere il successivo di 10, che evidentemente non fa parte dei valori ammessi. Un problema analogo si presenta, nel caso di valori interi, quando il calcolo di *I* generi un overflow. Per ovviare a questi problemi alcuni linguaggi (per esempio FORTRAN IV e Pascal) lasciano indefinito (cioè la definizione del linguaggio non specifica quale debba essere) il valore di *I* alla fine del ciclo. In altre parole, ogni implementazione di questi linguaggi può decidere di comportarsi come crede, con le immaginabili conseguenze in termini di (non) portabilità dei programmi. Altri linguaggi (per esempio ALGOL W, ALGOL 68, Ada e, sotto certe condizioni, C++) tagliano la testa al toro, decretando che la variabile di controllo è una variabile *locale* al *for*, dunque non visibile all'esterno del ciclo. In questo caso l'intestazione del *for* dichiara implicitamente la variabile indice, con il tipo determinato da quello delle espressioni *inizio* e *fine*.

**Salto nel ciclo** L'ultimo punto che merita attenzione, infine, riguarda la possibilità di "saltare", mediante un comando *goto*, all'interno di un ciclo *for*. La maggior parte dei linguaggi vieta tale possibilità per evidenti motivi semanticici, mentre vi sono meno restrizioni sulla possibilità di usare un *goto* per uscire da

un ciclo.

Abbiamo già fatto diverse considerazioni relative all'implementazione del comando `for`. Particolari accorgimenti possono essere usati dal compilatore per ottimizzare il codice prodotto (ad esempio, eliminando dei test che coinvolgono costanti) oppure per limitare situazioni di overflow che si potrebbero verificare nell'incremento dell'indice `I` (aggiungendo opportuni test).

**Espressività dell'iterazione determinata** Usando l'iterazione determinata possiamo esprimere la ripetizione di un comando per  $n$  volte, dove  $n$  è una quantità arbitraria, non nota al momento della stesura del programma, ma fissata al momento in cui l'iterazione inizia. È evidente che si tratta di qualcosa che non è possibile esprimere usando solo i comandi condizionali e l'assegnamento, perché in questo caso è possibile ripetere un comando solo ripetendo sintatticamente il comando nel corpo del programma: essendo ogni programma di lunghezza finita, abbiamo un limite al numero massimo di ripetizioni che possiamo includere in uno specifico programma.

Nonostante questo aumento del poter espressivo, l'iterazione determinata da sola non è sufficiente a rendere Turing completo un linguaggio di programmazione. Si pensi, ad esempio, ad una semplice funzione  $f$  definita come segue:

$$f(x) = \begin{cases} x & \text{se } x \text{ pari} \\ \text{non termina} & \text{se } x \text{ dispari} \end{cases}$$

Una tale funzione è certamente calcolabile: ogni programmatore saprebbe come realizzarla, usando un `while`, o un `goto` o una chiamata ricorsiva per ottenere una computazione che non termina mai. Tuttavia tale funzione non è rappresentabile mediante un linguaggio che abbia solo assegnamento, comando sequenziale, `if` e iterazione determinata, dato che, come si può verificare facilmente, in un tale linguaggio tutti i programmi terminano per ogni possibile input; in altri termini, in tale linguaggio si possono definire solo funzioni totali mentre la nostra funzione  $f$  è parziale<sup>6</sup>. Per ottenere un linguaggio Turing completo è necessaria l'iterazione indeterminata. La complicazione della semantica informale del `for` è solo apparente, mentre la facilità di traduzione in linguaggio macchina di un `while` non deve trarre in inganno: da un punto di vista formale, il `while` è semanticamente più complicato del `for`. Infatti, come abbiamo visto nel Capitolo 2, la semantica di un comando `while` è definita in termini di se stessa, cosa che, se a prima vista appare un po' strana, trova la sua giustificazione formale in tecniche di punto fisso che vanno oltre gli scopi di questo testo (e alle quali accenneremo nel paragrafo sulla ricorsione, più avanti). Anche se non abbiamo definito formalmente la semantica del `for`, il lettore può convincersi che questa può essere data in termini

<sup>6</sup> In realtà vi sono anche funzioni *totali* che non sono definibili usando solo assegnamento, comando sequenziale, `if` e `for`; un esempio celebre è quello della funzione di Ackermann, per la cui definizione rimandiamo ai testi di teoria della calcolabilità, per esempio [5].

più semplici (si veda l'Esercizio 3). La maggiore complicazione semantica del `while` rispetto al `for` corrisponde alla maggiore espressività del primo costrutto: è evidente infatti che ogni comando `for` può essere tradotto in modo semplice in un `while`.

Ci si può allora chiedere perché un linguaggio fornisca dei costrutti di iterazione determinata, quando quelli di iterazione indeterminata permettono di fare le stesse cose. La risposta è principalmente di natura pragmatica. Il comando `for` è una forma molto compatta di iterazione: mettendo sulla stessa linea, all'inizio dell'iterazione, le tre componenti di inizializzazione, controllo, e incremento da un parte rende più semplice la comprensione di quello che il ciclo sta facendo, dall'altra previene alcuni errori comuni, quali dimenticare l'inizializzazione o l'incremento della variabile di controllo. L'uso di un `for` invece di un `while` può dunque essere un importante strumento per facilitare la comprensione e, dunque, il testing e la manutenzione di un programma. Vi è poi, in alcuni linguaggi e su determinate architetture, anche una motivazione di tipo implementativo: un ciclo `for` può spesso essere compilato in modo più efficiente (e soprattutto può essere ottimizzato meglio) di un ciclo `while`, in specie per quanto riguarda l'allocazione dei registri.

**Il `for` di C** In C (e nei suoi successori, tra i quali Java), il `for` è ben lontano dall'essere, nel suo caso generale, un'iterazione determinata. La sua versione generale è

```
for (exp1; exp2; exp3)
    comando
```

la cui semantica è la seguente:

1. valuta  $\text{exp}_1$ ;
2. valuta  $\text{exp}_2$ : se è zero, termina l'esecuzione del `for`;
3. esegui il corpo, cioè `comando`;
4. valuta  $\text{exp}_3$  e riprendi da (2).

Come si vede non c'è alcun tentativo di congelare il valore delle espressioni di controllo, né vi è alcun vincolo sulla possibilità di modificare il valore dell'indice (che, in questo caso generale, non esiste neppure). È evidente come la semantica esprima il fatto che `for` è in tutto e per tutto, in C, un'abbreviazione per un `while`.

Sfruttando il fatto che, in C, un comando è anche un'espressione, si ottiene la forma più usuale con la quale un `for` si presenta anche in C:

```
for (i = inizio; i <= fine; i += passo) {
    comandi
}
```

che è un'abbreviazione (assai importante pragmaticamente) per

```
i = inizio;
while (i <= fine) {
    comandi
    i += passo;
}
```

**For-each** Uno degli usi più comuni dei costrutti d'iterazione è costituito dalla scansione sequenziale di tutti gli elementi di una struttura dati. Tipico esempio è il seguente, che presenta una funzione che calcola la somma dei valori di un array di interi:

```
int somma(int[] A) {
    int acc = 0;
    for(int i=0; i<lenght(A); i++)
        acc += A[i];
    return acc;
}
```

Questa funzione è piena di dettagli che il compilatore già conosce: il primo e l'ultimo indice di A, il controllo specifico che i abbia raggiunto il limite. Quanti più dettagli devono essere inseriti in un costrutto, tanto più facile è commettere un errore e tanto più difficile diviene comprendere a colpo d'occhio "cosa fa" quel costrutto. Nel caso di somma, quello che si vuol esprimere è semplicemente l'applicazione del corpo *ad ogni elemento* di A.

Alcuni linguaggi permettono di esprimere situazioni di questo genere mediante un costrutto apposito, che possiamo chiamare *for-each*, con la sintassi generale seguente:

**foreach** (*ParametroFormale* : *Espressione*) *Comando*

Il costrutto for-each esprime l'applicazione del *Comando* (nel quale può ovviamente comparire *ParametroFormale*) a tutti gli elementi di *Espressione*.

Usando tale costrutto la nostra funzione per la somma di un vettore potrebbe essere scritta come:

```
int somma(int[] A) {
    int acc = 0;
    foreach(int e : A)
        acc += e;
    return acc;
}
```

dove leggiamo l'intestazione del *foreach* come "per ogni elemento e in A". L'indice del vettore, insieme a tutti i suoi limiti è scomparso, in un costrutto più sintetico ed elegante.

L'uso del costrutto *foreach* non è limitato ai vettori, ma può essere applicato a tutte le collezioni sulle quali sia definita in modo naturale una nozione di scansione ("iterazione"). Oltre alle enumerazioni e agli insiemi (che vedremo nel Capitolo 10), menzioniamo il caso particolarmente significativo di linguaggi che permettono all'utente la definizione di tipi che sono "iterabili".

Tra i linguaggi più diffusi, Java nella versione 5 supporta il costrutto *foreach*. La parola chiave usata è la semplice *for*<sup>7</sup>, ma la sintassi diversa permette di

<sup>7</sup>Il costrutto *for-each* è stato aggiunto "in corsa", quando il linguaggio era ormai diffuso e utilizzato da anni. In tal caso la modifica dell'insieme delle parole riservate non è una buona scelta progettuale: vecchi programmi che avessero usato come identificatore la nuova parola chiave avrebbero cessato di funzionare.

disambiguare i due costrutti senza problemi. In Java il *for-each* (chiamato anche *enhanced for* nella documentazione) è applicabile a tutti i sottotipi del tipo di libreria *Iterable*.

## 8.4 Programmazione strutturata

L'ostracismo sviluppatosi nei confronti del comando *goto* a partire dagli anni '70 non è stato un fenomeno isolato, dovuto a una qualche idiosincrasia nei confronti di questo costrutto, ma è stato solo uno dei tanti aspetti di un processo più ampio che in quegli anni ha portato all'affermazione della cosiddetta *programmazione strutturata*. Questa può essere considerata l'antesignana delle moderne metodologie di programmazione e, come dice il nome stesso, consiste in una serie di "prescrizioni" volte a permettere uno sviluppo il più possibile strutturato del codice e, corrispondentemente, del flusso di controllo. Tali prescrizioni hanno sia una natura metodologica, fornendo dei precisi metodi di sviluppo dei programmi, sia una componente linguistica, indicando opportune tipologie di comandi da usarsi (nella sostanza, quelli sin qui visti salvo il *goto*). Vediamo più in dettaglio alcuni punti salienti della programmazione strutturata e delle relative implicazioni linguistiche.

**Progettazione del programma top-down o comunque gerarchica** Il programma è sviluppato per raffinamenti successivi, partendo da una prima specifica abbastanza astratta e aggiungendo successivamente ulteriori dettagli.

**Modularizzazione del codice** È opportuno raggruppare i comandi che corrispondono ad ogni specifica funzione dell'algoritmo che si vuole implementare. Per fare questo si useranno tutti gli strumenti linguistici messi a disposizione dal linguaggio che si usa, dai comandi composti ai costrutti per l'astrazione del controllo quali procedure, funzioni e anche veri e propri moduli, nei linguaggi che li permettono.

**Uso di nomi significativi** L'uso di nomi significativi per variabili, procedure ecc. semplifica molto la comprensione del codice e quindi la possibilità di fare successivamente interventi di manutenzione. Anche se questa pare (ed è) un'osservazione ovvia, nella pratica è troppe volte disattesa.

**Uso estensivo di commenti** Anche i commenti sono essenziali per la comprensione, il testing, la verifica, la correzione e la modifica del codice. Un programma senza alcun commento, oltre una certa dimensione, diviene ben presto ingestibile.

**Uso di tipi di dato strutturati** La possibilità di usare opportuni tipi di dato, quali ad esempio i record, per raggruppare e strutturare informazioni anche di tipo eterogeneo, facilita sia la progettazione del codice che la successiva manutenzione. Ad esempio, se possiamo usare un'unica variabile, di tipo "record studente", per memorizzare le informazioni relative a nome, cognome, numero di matricola ed anno di iscrizione di uno studente, evidentemente la scrittura del programma sarà molto facilitata rispetto al caso in cui si debbano usare quattro variabili diverse per le informazioni relative ad uno stesso studente.

**Uso di costrutti strutturati per il controllo** Questo, dal punto di vista linguistico, è l'aspetto essenziale: per la realizzazione della programmazione strutturata è necessario usare costrutti di controllo strutturati, ossia costrutti che, sostanzialmente, abbiano un solo punto di ingresso ed un solo punto di uscita.

L'ultimo punto è quello per noi più interessante e merita qualche attenzione ulteriore. L'idea essenziale dei costrutti di controllo strutturati è che essi, avendo un solo punto di ingresso ed un solo punto di uscita, permettono una strutturazione del codice nella quale la scansione lineare del testo del programma corrisponde al flusso di esecuzione: se il comando C2 segue testualmente il comando C1, all'uscita (unica) dal comando C1 il controllo sarà passato all'entrata (unica) del comando C2. Ogni comando al suo interno potrà avere strutture complesse: ramificazioni (come in un `if`) o cicli (come nel `for`), con una struttura del controllo non lineare o che permette salti all'indietro. L'importante è che ogni componente elementare, esternamente, sia visibile in termini di un solo ingresso ed una sola uscita. Questa proprietà, fondamentale per la comprensione del codice, è violata se la presenza di un comando quale `goto` permette di "saltare" avanti e indietro nel programma. In tal caso si può arrivare facilmente alle situazioni note come "codice spaghetti" dove, graficamente, il flusso del controllo fra le varie componenti del programma, invece che da un semplice grafo con pochi archi (che collegano l'uscita di un comando all'ingresso del successivo), è descritto da un grafo i cui archi ricordano... un piatto di spaghetti!

I costrutti per il controllo sin qui visti, salvo il `goto`, sono tutti costrutti strutturati e sono quelli rimasti nei linguaggi di programmazione moderni. Dal punto di vista teorico essi permettono di scrivere programmi per tutte le funzioni calcolabili, come già abbiamo osservato. Dal punto di vista pragmatico, sono sufficienti per esprimere tutte le tipologie di flusso di controllo presenti nelle applicazioni reali. I costrutti che abbiamo discusso al termine del Paragrafo 8.3.1 intervengono per gestire quei casi in cui si debba uscire da un ciclo, da una procedura o interrompere comunque un'elaborazione prima della terminazione "normale". Tutti questi casi potrebbero essere trattati in modo naturale mediante un `goto`. Ad esempio, se volessimo elaborare tutti gli elementi di un file che leggiamo dall'esterno potremmo usare un codice della forma<sup>8</sup>

```
while true do{
    read(X);
    if X = end_of_file then goto fine;
    elabora(X);
}
fine: ...
```

Si osservi che quest'uso di `goto` non viola il principio "un solo ingresso e una sola uscita", perché il salto non fa che anticipare l'uscita, che avviene comunque in un unico punto per tutto il costrutto. Il comando strutturato `break` (o suoi analoghi)

è la forma canonica di questo "salto alla fine di un ciclo": sostituito al posto di `goto fine` rende più chiaro il programma ed evita l'introduzione di un'etichetta (la destinazione del salto implicito nel `break` è l'unica uscita del costrutto).

Ricordiamo infine che la programmazione strutturata ha costituito una prima risposta alle esigenze della cosiddetta programmazione in grande (*in the large*)<sup>9</sup>, dato che essa permette di decomporre un sistema software di vaste dimensioni in varie componenti che hanno una certa indipendenza. Il grado di indipendenza dipende dai meccanismi di astrazione usati: ad esempio, se si usano procedure la comunicazione fra i vari componenti può avvenire unicamente attraverso i parametri. Risposte più significative alle esigenze della programmazione *in the large* non possono tuttavia essere date solo al livello linguistico di un linguaggio di programmazione. L'ingegneria del software ha studiato molteplici metodologie per gestire il progetto e la realizzazione di grandi sistemi software. Alcune di queste metodologie hanno anche dei risvolti linguistici, che però non possiamo affrontare compiutamente in questa sede. Il paradigma orientato agli oggetti, e alcuni formalismi di specifica dei progetti orientati agli oggetti (come UML), sono alcune delle risposte più vicine agli aspetti che trattiamo in questo testo.

## 8.5 La ricorsione

La ricorsione è un altro meccanismo, alternativo all'iterazione, per ottenere linguaggi di programmazione Turing equivalenti. In termini "empirici" una funzione (o procedura) ricorsiva è una procedura nel cui corpo compare una chiamata a se stessa. Si può avere anche ricorsione indiretta, o meglio *mutua ricorsione*, quando una procedura *P* chiama un'altra procedura *Q* che, a sua volta, chiama *P*. Abbiamo già visto nel Capitolo 7 l'esempio della funzione ricorsiva `fib` che calcola l'ennesimo termine della successione di Fibonacci:

```
int fib (int n){
    if (n == 0) return 1;
    else
        if (n == 1) return 1;
        else
            return fib(n-1) + fib(n-2);
}
```

Il fatto che una funzione, come `fib`, sia definita in termini di se stessa potrebbe suscitare qualche dubbio sulla natura della funzione che stiamo definendo. In realtà le definizioni ricorsive, dette anche definizioni induttive, sono abbastanza comuni in matematica: come mostrato più in dettaglio nel riquadro di pag. 233, l'idea è quella di descrivere il risultato dell'applicazione di una funzione *f* ad un argomento *X* in termini dell'applicazione di *f* stessa ad argomenti che siano "più piccoli" di *X*. Se il dominio su cui *f* è definita è tale da non ammettere catene

<sup>8</sup>Volendo evitare il `goto`, usando solo `while` e `if` ci troveremmo a dover scrivere un codice molto meno naturale.

<sup>9</sup>Con questo termine, come noto, si indica la realizzazione di sistemi software di dimensioni notevoli.

infinte di elementi “sempre più piccoli”, così facendo siamo sicuri che, dopo un numero finito di applicazioni della funzione  $f$ , si arriva ad un caso terminale, dalla definizione del quale possiamo ricostruire il valore di  $f$  applicata ad  $X$ . Ad esempio, ricordando che il fattoriale di un numero intero naturale  $n$  è dato dal prodotto  $1 \cdot 2 \cdots n$ , possiamo definire in modo induttivo la funzione che calcola il fattoriale come segue:

$$\begin{aligned} \text{fattoriale}(0) &= 1 \\ \text{fattoriale}(n+1) &= (n+1) \cdot \text{fattoriale}(n), \end{aligned}$$

dove  $n$  è un generico numero intero naturale. In modo analogo possiamo definire la funzione che calcola l’ennesimo termine della successione di Fibonacci, per la quale abbiamo fornito prima un programma ricorsivo.

Se dunque definizioni induttive in matematica e funzioni ricorsive nei linguaggi di programmazione sono molto simili, vi è tuttavia una differenza fondamentale. Nel caso delle definizioni induttive non tutte le possibili definizioni di una funzione in termini di se stessa vanno bene. Se, ad esempio, scriviamo

$$\begin{aligned} \text{foo}(0) &= 1 \\ \text{foo}(n) &= \text{foo}(n) + 1 \quad \text{per } n > 0, \end{aligned}$$

è evidente che nessuna funzione totale sui naturali soddisfa queste equazioni, quindi con esse non definiamo alcuna funzione. Se invece scriviamo

$$\text{fie}(1) = \text{fie}(1),$$

adesso il problema è opposto: “molte” funzioni soddisfano tale equazione, quindi di nuovo questa scrittura non fornisce una buona definizione.

D’altro canto, in un qualsiasi linguaggio di programmazione che supporti la ricorsione è perfettamente legittimo scrivere le funzioni

```
int foo1 (int n){
    if (n == 0) return 1;
    else
        return foo1(n) + 1;
}
```

ed anche

```
int fie1 (int n){
    if (n == 1) return fie1(1);
}
```

Si tratta di funzioni che in alcuni casi non terminano (quando  $n > 0$  per `foo1(n)` e per  $n = 1$  nel caso di `fie1(n)`), ma dal punto di vista semantico questo non è un problema, perché, come abbiamo visto nel Capitolo 5, i programmi definiscono funzioni parziali.

### Definizioni induttive

I numeri (interi) naturali  $0, 1, 2, 3, \dots$ , secondo una presentazione assiomatica dovuta a Giuseppe Peano, possono essere definiti come il minimo insieme  $X$  che soddisfa le due regole seguenti:

1.  $0 \in X$ ;
2. se  $n \in X$  allora  $n + 1 \in X$ ;

dove si assumono come primitive la nozione di 0 (zero), quella di numero (indicato con  $n$ ), quella di successore di  $n$  (indicato con  $n + 1$ ). Questa definizione dei naturali fornisce una giustificazione intuitivamente evidente del principio di induzione, uno strumento fondamentale della matematica ed anche, per certi versi, dell’informatica. Tale principio può essere enunciato come segue: per dimostrare che una proprietà  $P(n)$  è vera su tutti i numeri naturali basta dimostrare che valgono le due condizioni seguenti:

1.  $P(0)$  è vera;
2. per ogni naturale  $n$ , se  $P(n)$  è vera allora è vera anche  $P(n + 1)$ .

Oltre che per la dimostrazione di proprietà, l’induzione è uno strumento potente anche per la definizione di funzioni. Infatti si può dimostrare che se  $g : (\mathbb{N} \times X) \rightarrow X$  è una funzione totale e  $a$  è un elemento in  $X$ , allora esiste una sola funzione (totale)  $f : \mathbb{N} \rightarrow X$  tale che

1.  $f(0) = a$ ;
2.  $f(n + 1) = g(n, f(n))$ .

Una tale coppia di equazioni fornisce dunque una *definizione induttiva* della funzione  $f$ .

Quanto sin qui detto per l’induzione sui numeri interi naturali può essere generalizzato ad arbitrari insiemi su cui sia definita una relazione d’ordine  $\prec$  che sia ben fondata, ossia una relazione che non ammetta catene discendenti infinite  $\dots \prec x \prec \dots \prec x_1 \prec x_0$ . In questo caso il principio di induzione, detto di induzione ben fondata, può essere espresso come segue. Sia  $\prec$  una relazione ben fondata definita su un insieme  $A$ ; per dimostrare che vale  $P(a)$  per ogni  $a$  appartenente ad  $A$ , basta dimostrare l’implicazione seguente

per ogni  $a \in A$ , ( se vale  $P(b)$  per ogni  $b \prec a$ , allora vale  $P(a)$ ).

### 8.5.1 La ricorsione in coda

Nel Capitolo 7 abbiamo visto come, in generale, la presenza della ricorsione in un linguaggio di programmazione renda necessaria la gestione dinamica della memoria, in quanto non è possibile determinare staticamente il numero massimo di istanze di una stessa funzione attive contemporaneamente (e quindi il numero

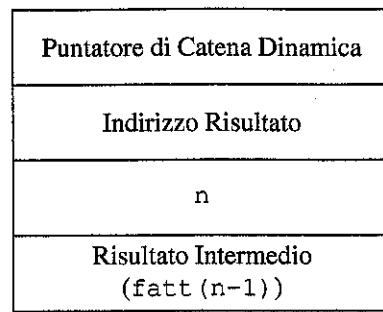


Figura 8.4 Record di attivazione per la funzione `fatt`.

massimo di record di attivazione necessari). Ad esempio, abbiamo visto che per la chiamata `fib(n)` della nostra funzione Fibonacci tale numero è eguale all' $n$ -esimo elemento della serie di Fibonacci e, ovviamente,  $n$  può non essere noto a tempo di compilazione. Se però consideriamo meglio la natura delle chiamate ricorsive ci accorgiamo che in alcuni casi possiamo evitare l'allocazione di nuovi record di attivazione per le chiamate successive di una stessa funzione, in quanto possiamo riutilizzare sempre lo stesso spazio di memoria. Per comprendere questo punto confrontiamo due funzioni ricorsive per il calcolo del fattoriale di un numero naturale. La prima è quella usuale

```
int fatt (int n) {
    if (n <= 1)
        return 1;
    else
        return n * fatt(n-1);
}
```

Il record di attivazione di una chiamata `fatt(n)`, un poco semplificato, ha la struttura mostrata nella Figura 8.4: il campo  $n$  conterrà il valore del parametro attuale della procedura; il campo *Risultato Intermedio* conterrà il risultato intermedio fornito dalla valutazione di `fatt(n-1)`; il campo *Indirizzo Risultato*, infine, contiene l'indirizzo della zona di memoria nella quale deve essere restituito il risultato (e cioè, l'indirizzo del campo *Risultato Intermedio* del chiamante per le chiamate successive alla prima). È importante notare che il valore del campo *Risultato Intermedio*, presente nel record di attivazione di `fatt(n)`, può essere determinato solo quando la chiamata ricorsiva di `fatt(n-1)` termina e che il valore di tale campo, come risulta dal codice di `fatt`, serve nel calcolo di  $n * \text{fatt}(n-1)$  per ottenere il valore di `fatt(n)`. Detto in altri termini, quando abbiamo la chiamata `fatt(n)`, prima che questa possa terminare dobbiamo conoscere il valore di `fatt(n-1)`; a sua volta, perché la chiamata di `fatt(n-1)` possa terminare dobbiamo conoscere il valore di `fatt(n-2)` e così via, ricorsivamente, fino al caso terminale `fatt(1)`; dunque tutti i record di attivazione delle

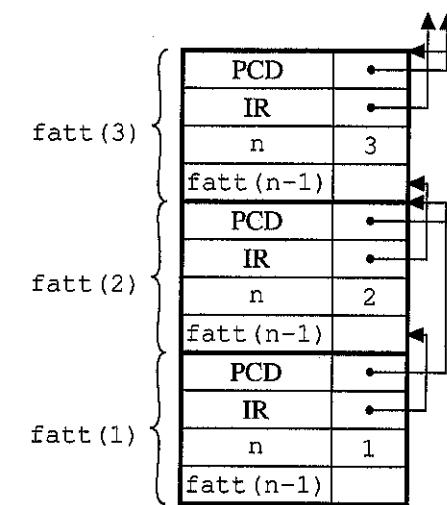


Figura 8.5 Pila dei RdA dopo la chiamata di `fatt(3)` e le due chiamate ricorsive `fatt(2)` e `fatt(1)`.

varie chiamate ricorsive `fatt(n)`, `fatt(n-1)`, ..., `fatt(1)` devono esistere contemporaneamente sulla pila, in zone di memoria distinte.

La Figura 8.5 mostra la pila dei record di attivazione creata in seguito alla chiamata di `fatt(3)` e delle successive chiamate ricorsive `fatt(2)` e `fatt(1)`. Quando raggiungiamo il caso terminale, la chiamata di `fatt(1)` termina immediatamente restituendo il valore 1 che, usando il puntatore contenuto nel campo *Indirizzo Risultato* del RdA di `fatt(1)`, sarà restituito al campo *Risultato Intermedio* del RdA di `fatt(2)`, come mostrato in Figura 8.6. A questo punto anche la chiamata di `fatt(2)` può terminare, restituendo il valore  $2 \cdot \text{fatt}(1) = 2 \cdot 1$  alla chiamata di `fatt(3)`, come mostrato in Figura 8.7. Infine anche la chiamata di `fatt(3)` terminerà, restituendo al programma chiamante il valore  $3 \cdot \text{fatt}(2) = 3 \cdot 2 = 6$ .

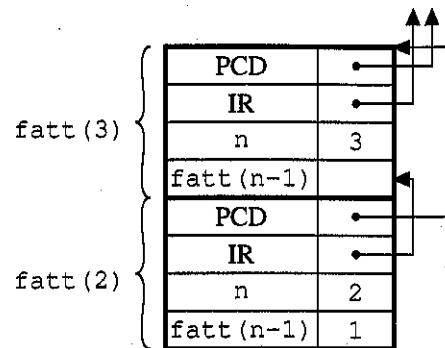
Consideriamo adesso la seguente funzione

```
int fattrc (int n, int res) {
    if (n <= 1)
        return res;
    else
        return fattrc(n-1, n * res)
}
```

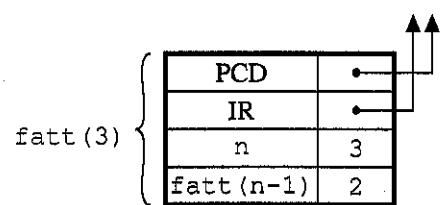
che, se chiamata con `fattrc(n, 1)`, restituisce anch'essa il fattoriale di  $n$ .

Anche in questo caso, la chiamata iniziale `fattrc(n, 1)` produce  $n - 1$  ulteriori chiamate ricorsive

`fattrc(n-1, n*1)`,



**Figura 8.6** Pila dei RdA dopo la terminazione della chiamata di `fatt(1)`.



**Figura 8.7** Pila dei RdA dopo la terminazione della chiamata di `fatt(2)`.

```
fattrc(n-2, (n-1)*n*1),
```

```
...,
```

```
fattrc(1, 2*...*(n-1)*n*1).
```

Tuttavia, notiamo che, adesso, per  $n > 1$  il valore restituito dalla generica chiamata `fattrc(n, res)` è *esattamente lo stesso* valore che viene restituito dalla successiva chiamata ricorsiva `fattrc(n-1, n*res)`, senza che vi sia alcuna computazione aggiuntiva. Il valore finale restituito dalla chiamata iniziale `fattrc(n, 1)` è quindi quello restituito dall'ultima chiamata ricorsiva `fattrc(1, 2*...*(n-1)*n*1)` (ed è dunque  $n \cdot (n - 1) \cdot (n - 2) \cdots 1$ ), senza che vi sia necessità di "risalire" la serie delle chiamate, utilizzando poi i risultati intermedi per calcolare il valore finale, come invece avveniva nel caso di `fatt`.

Da quanto detto appare evidente che, una volta che `fattrc(n, res)` ha chiamato ricorsivamente `fattrc(n-1, n*res)`, non c'è alcuna necessità di continuare a mantenere le informazioni presenti nel record di attivazione della chiamata `fattrc(n, res)` dato che tutte le informazioni che servono per calcolare il risultato finale sono state passate a `fattrc(n-1, n*res)`. Questo significa che il record di attivazione della chiamata ricorsiva `fattrc(n-1, n*res)` può sem-

plicemente riutilizzare lo spazio di memoria allocato per il record di attivazione di `fattrc(n, res)`. Tale considerazione vale anche per le chiamate successive e quindi, complessivamente, la funzione `fattrc` avrà necessità di una sola zona di memoria per l'allocazione di un solo record di attivazione, indipendentemente dal numero di chiamate ricorsive che vengono fatte. Abbiamo quindi ottenuto una funzione ricorsiva per la quale la memoria può essere allocata staticamente!

La ricorsione del tipo illustrato dalla funzione `fattrc` si dice *in coda* (tail recursion, usando la terminologia inglese) in quanto la chiamata ricorsiva è, per così dire, l'ultima cosa che viene fatta nel corpo della procedura: dopo la chiamata ricorsiva non deve essere compiuta alcuna ulteriore computazione. Più in generale possiamo dare la seguente definizione.

**Definizione 8.3 (Ricorsione in coda)** Sia  $f$  una funzione che nel suo corpo contenga la chiamata ad una funzione  $g$  (diversa da  $f$  oppure anche eguale ad  $f$  stessa). La chiamata di  $g$  si dice "chiamata in coda" (o tail call) se la funzione  $f$  restituisce il valore restituito da  $g$  senza dover fare alcuna ulteriore computazione. Diciamo che la funzione  $f$  ha la ricorsione in coda (è tail recursive) se tutte le chiamate ricorsive presenti in  $f$  sono chiamate in coda.

Ad esempio, nella funzione

```
int f (int n){
    if (n == 0)
        return 1;
    else
        if (n == 1)
            return f(0);
        else
            if (n == 2)
                return f(n-1);
            else
                return f(1)*2;
}
```

le prime due chiamate ricorsive sono in coda, la terza non è in coda; quindi la funzione  $f$  non ha la ricorsione in coda.

Il nostro interesse per la ricorsione in coda risiede nella possibilità di implementarla usando un solo record di attivazione, e quindi uno spazio di memoria costante. Le considerazioni che abbiamo fatto per la funzione `fattrc`, infatti, sono del tutto generali, e non dipendono dalla specifica forma di tale funzione, ma solo dal fatto che si tratta di una funzione con ricorsione in coda. Quanto detto non vale più, tuttavia, nel caso in cui si considerino anche funzioni di ordine superiore (ossia, qualora vi siano funzioni passate come parametro), come vedremo più avanti in questo paragrafo.

In generale è sempre possibile trasformare una funzione che non abbia la ricorsione in coda in una, equivalente, che la abbia, complicando opportunamente la funzione. L'idea è quella di fare in modo che tutta la computazione che deve essere fatta dopo la chiamata ricorsiva (e che rende la funzione non tail recursive) sia fatta prima della chiamata, per quanto possibile; la parte di lavoro che non

può essere fatta prima della chiamata ricorsiva (perché ad esempio utilizza risultati di questa) viene “passata” con opportuni parametri aggiuntivi, alla chiamata ricorsiva stessa. Questa tecnica è proprio quella che abbiamo applicato nel caso della funzione `tail recursive fattrc` dove, invece che calcolare ricorsivamente il prodotto  $n \cdot fatt(n-1)$  nel corpo della chiamata di `fatt(n)`, abbiamo aggiunto un parametro `res` che permette di passare alla generica chiamata ricorsiva `fattrc(j-1, j*res)` il prodotto  $n \cdot (n-1) \cdot (n-2) \cdots j$ . Dunque, in questo caso, il calcolo del fattoriale è fatto incrementalmente dalle successive chiamate ricorsive, in modo analogo a quanto verrebbe fatto da una funzione *iterativa* quale la seguente.

```
int fatt_it (int n, int res){
    res=1;
    for (i=n; i>=1; i--)
        res = res*i;
}
```

In modo analogo a quanto fatto con la funzione fattoriale, possiamo trasformare anche la funzione `fib` in una funzione con la ricorsione in coda `fibrc` aggiungendo due ulteriori parametri:

```
int fibrc (int n, int res1, int res2){
    if (n == 0)
        return res2;
    else
        if (n == 1)
            return res2;
        else
            return fibrc(n-1, res2, res1+res2);
}
```

La chiamata `fibrc(n, 1, 1)` restituisce l' $n$ -esimo termine della successione di Fibonacci. Ovviamente, sia nel caso di `fibrc` che in quello di `fattrc`, se vogliamo rendere invisibili i parametri aggiuntivi, possiamo incapsulare queste funzioni in altre che abbiano solo il parametro `n`. Ad esempio, nello scope della dichiarazione di `fibrc`, possiamo definire

```
int fibrcvera (int n){
    return fibrc(n, 1, 1);
}
```

La trasformazione di una funzione in una equivalente con la ricorsione in coda può essere fatta in modo automatico, usando una tecnica detta “continuation passing style” che, in sostanza, consiste nel rappresentare, in un dato punto di un programma, la parte rimanente del programma mediante una funzione detta continuazione. Nel caso in cui si voglia rendere `tail recursive` una funzione basta rappresentare mediante una continuazione quello che rimane da calcolare e passare tale continuazione alla chiamata ricorsiva. Questa tecnica però non sempre produce delle funzioni che possano essere eseguite con uno spazio di memoria costante, in quanto la continuazione, essendo una funzione, potrebbe contenere delle variabili che vanno valutate nell’ambiente del chiamante e quindi necessitano dell’RdA di quest’ultimo.

### 8.5.2 Ricorsione o iterazione?

Senza entrare nel dettaglio di risultati teorici (per altro, estremamente importanti ed interessanti) ricordiamo che ricorsione e iterazione (nella sua forma più generale) sono metodi alternativi per raggiungere lo stesso potere espressivo. L’uso dell’uno o dell’altro è spesso dovuto, oltre che alla predisposizione del programmatore, alla natura del problema: per l’elaborazione di dati che usino strutture rigide (matrici, tabelle ecc.), come normalmente accade nelle applicazioni di tipo numerico o nella gestione di dati di natura amministrativa, è spesso più facile usare costrutti iterativi; dove invece si usino strutture dati di natura simbolica, che si prestano ad essere definite naturalmente in modo ricorsivo (ad esempio liste, alberi ecc.), è spesso più naturale usare procedure ricorsive.

Spesso si considera la ricorsione molto più inefficiente dell’iterazione e quindi i linguaggi dichiarativi molto meno efficienti dei linguaggi imperativi. Le considerazioni svolte in precedenza sulla ricorsione in coda ci fanno comprendere che la ricorsione non è necessariamente meno efficiente dell’iterazione sia in termini di occupazione di memoria che di tempo d’esecuzione. Certamente, implementazioni naïf di funzioni ricorsive, quali quelle che spesso risultano dalla traduzione diretta di definizioni induttive, possono essere assai inefficienti. Questo è il caso, ad esempio, della nostra procedura `fib(n)` che ha tempo d’esecuzione e occupazione di memoria esponenziali in  $n$ . Tuttavia, come abbiamo visto, usando funzioni ricorsive più “astute”, quali quelle con la ricorsione in coda, possiamo ottenere prestazioni analoghe a quelle dei corrispondenti programmi iterativi. La funzione `fibrc(n)` infatti usa uno spazio di memoria di dimensione costante e richiede un tempo d’esecuzione lineare in  $n$ .

Riguardo poi al confronto fra linguaggi imperativi e dichiarativi le cose sono più complesse e verranno analizzate nei capitoli dedicati al paradigma funzionale e a quello logico.

## 8.6 Sommario del capitolo

In questo capitolo abbiamo analizzato le componenti di un linguaggio di alto livello che riguardano il controllo del flusso di esecuzione di un programma. Abbiamo visto innanzitutto le espressioni, per le quali abbiamo analizzato:

- i tipi di sintassi più usati per esse (ad albero, oppure in forma lineare prefissa, infissa e postfissa) e le relative regole di valutazione;
- le regole di precedenza e di associatività necessarie per la notazione infissa;
- i problemi legati, in generale, all’ordine di valutazione delle sottoespressioni di un’espressione: perché sia definita con precisione la semantica dell’espressione tale ordine deve essere definito con esattezza;
- particolari tecniche di valutazione (corto circuito, ossia valutazione lazy) usate in alcuni linguaggi, e come anche queste debbano essere considerate per poter definire il valore corretto dell’espressione.

Siamo quindi passati ai comandi, vedendo:

- il comando di base dei linguaggi imperativi, l'assegnamento; anche se si tratta di un comando sostanzialmente semplice, per poterne capire con esattezza la semantica si deve aver chiara la nozione di variabile;
- i vari comandi che permettono di esprimere in modo strutturato il controllo (condizionali e iterativi);
- i principi della programmazione strutturata, soffermandoci sull'annosa questione relativa all'uso del comando `goto`;

Il paragrafo finale, infine, ha trattato la ricorsione, un metodo alternativo all'iterazione per esprimere gli algoritmi, soffermandosi in particolare sulla ricorsione in coda, un tipo di ricorsione particolarmente efficiente sia in spazio che in tempo.

Restano da vedere ancora agli aspetti relativi ai meccanismi di astrazione sul controllo (procedure, parametri ed eccezioni), che saranno argomento del prossimo capitolo.

## 8.7 Nota bibliografica

Molti testi offrono una panoramica dei vari costrutti presenti nei linguaggi di programmazione; fra questi, i più completi sono [81] e [88].

Due articoli storici, di sicuro interesse per chi voglia approfondire la questione del `goto` sono [15], dove si dimostra il teorema di Böhm e Jacopini, e l'articolo di Dijkstra [33], dove viene discussa la "pericolosità" del comando di salto.

Un interessante articolo, anche se di lettura non immediata, che approfondisce le tematiche relative alle definizioni induttive è [3]. Per una introduzione più accessibile alla ricorsione e all'induzione si può consultare l'ottimo testo [106].

Secondo Abelson e Sussman [1] il termine "zucchero sintattico" è dovuto a Peter Landin, un pioniere dell'informatica cui si devono contributi fondamentali nell'area del progetto dei linguaggi di programmazione.

## 8.8 Esercizi

1. Si fornisca, in un qualsiasi linguaggio di programmazione, una funzione `f` tale che la valutazione dell'espressione  $(a+f(b)) * (c+f(b))$  fatta da sinistra a destra abbia un risultato diverso da quello ottenuto con la valutazione da destra a sinistra.
2. Si mostri come il costrutto `if then else` può essere usato per simulare la valutazione corto-circuito delle espressioni booleane in un linguaggio che valuta sempre tutti gli operandi prima dell'operatore booleano.
3. Si definisca la semantica operazionale del comando

```
for I = inizio to fine by passo do corpo
```

usando le tecniche viste nel Capitolo 2. Suggerimento: usando i valori delle espressioni `inizio`, `fine` e `passo` si può calcolare, prima di eseguire il `for`, il numero *ic* di ripetizioni che devono essere effettuate (si assume, come già

detto nel capitolo, che l'eventuale modifica di `I`, `inizio`, `fine` e `passo` nel corpo del `for` non ne influenzi la valutazione). Calcolato tale valore *n* il `for` può essere tradotto semplicemente in *ic* comandi in sequenza.

4. Si consideri la seguente funzione:

```
int novantuno (int x){
    if (x>100)
        return x-10;
    else
        return novantuno(novantuno(x+11));
}
```

Si tratta di una ricorsione di coda? Si motivi la risposta.

5. Il seguente frammento di codice è scritto in uno pseudolinguaggio che ammette iterazione determinata controllata numericamente, espressa con il costrutto `for`. Si dica cosa viene stampato.

```
z=1;
for i=1 to 5+z by 1 do{
    write(i);
    z++;
}
write(z);
```

6. Si dica cosa stampa il seguente blocco di codice, in un linguaggio con scope statico e passaggio per nome. La definizione del linguaggio riporta la seguente frase: "La valutazione dell'espressione  $E_1 \circ E_2$ , dove  $\circ$  è un qualunque operatore, consiste in (i) la valutazione di  $E_1$ ; (ii) successivamente, la valutazione di  $E_2$ ; (iii) infine, l'applicazione dell'operatore  $\circ$  ai due valori precedentemente ottenuti."

```
int x=5;
int P(name int m) {
    int x=2;
    return m+x;
}
write(P(x++)+x);
```

## Astrarre sul controllo

---

Il concetto di astrazione è un tema ricorrente di questo testo. Fin dal primo capitolo abbiamo incontrato le macchine *astratte* e le loro gerarchie. In quel contesto abbiamo usato “astratto” in opposizione a “fisico”, indicando con macchina astratta un insieme di algoritmi e strutture dati non direttamente presenti in una specifica macchina fisica, ma su questa rappresentabili mediante interpretazione. Tuttavia, nel concetto di macchina astratta è fondamentale anche il fatto che essa in qualche misura nasconde la macchina sottostante.

Astrarre significa sempre nascondere qualcosa; spesso, astraendo da alcuni dettagli concreti di più oggetti, si riesce a far emergere con più chiarezza un concetto comune a quegli stessi oggetti. Ogni descrizione di un fenomeno (naturale, artificiale, fisico ecc.) non è costituita dall’insieme di *tutti* i dati relativi al fenomeno stesso. Altrimenti sarebbe come una carta geografica in scala 1:1, precisissima ma inutile. Ogni disciplina scientifica descrive un certo fenomeno di suo interesse concentrandosi solo su alcuni suoi aspetti, quelli che sono ritenuti più rilevanti per gli scopi che ci si è prefissi. È per questo che il linguaggio scientifico dispone di opportuni meccanismi per esprimere queste “astrazioni”. I linguaggi di programmazione, essi stessi astrazioni sulla macchina fisica, non fanno eccezione. Anzi, la loro espressività dipende in modo essenziale dai meccanismi di astrazione che essi forniscono. Tali meccanismi sono gli strumenti principali a disposizione del progettista e del programmatore per descrivere in modo accurato, ma anche semplice e suggestivo, la complessità dei problemi che devono essere risolti.

In un linguaggio di programmazione si distinguono in genere due classi di meccanismi di astrazione: quelli che forniscono *astrazione sul controllo*, e quelli relativi all’*astrazione sui dati*. I primi forniscono al programmatore la possibilità di nascondere dettagli procedurali; i secondi permettono di definire e utilizzare tipi di dato sofisticati senza far riferimento a come tali tipi siano implementati. Tratteremo in questo capitolo dei principali meccanismi di astrazione sul controllo, mentre l’astrazione sui dati sarà l’oggetto del Capitolo 11, dopo che avremo visto nel prossimo capitolo i meccanismi di strutturazione dei dati.

```

int foo (int n, int a) {
    int tmp=a;
    if (tmp==0) return n;
    else return n+1;
}
...
int x;
x = foo(3,0);
x = foo(x+1,1);

```

Figura 9.1 Definizione e uso di una funzione.

## 9.1 Sottoprogrammi

Ogni programma di qualche complessità è composto da molte componenti, ognuna delle quali concorre a fornire una parte della soluzione globale. La scomposizione di un problema in sottoproblemi permette di gestirne meglio la complessità: un problema più ristretto è più semplice da risolvere; la soluzione al problema globale si ottiene componendo in modo opportuno le soluzioni ai sottoproblemi.

Affinché tutto ciò sia davvero efficace, tuttavia, è necessario che il linguaggio di programmazione fornisca un supporto linguistico che faciliti e renda possibile tale suddivisione e, quindi, la ricomposizione. Questo supporto linguistico permette di esprimere decomposizione e ricomposizione direttamente nel linguaggio, trasformando questi concetti da semplici suggerimenti metodologici in veri e propri strumenti di progetto e programmazione.

Il concetto chiave fornito da tutti i linguaggi moderni è quello di sottoprogramma, o procedura, o funzione<sup>1</sup>. Una *funzione* è una porzione di codice identificata da un nome, dotata di ambiente locale proprio e capace di scambiare informazioni col resto del codice mediante *parametri*. Questo concetto si traduce in due diversi meccanismi linguistici: la *definizione* (o dichiarazione) di una funzione, e il suo *uso* (o chiamata). Nella Figura 9.1, le prime cinque linee costituiscono la definizione di una funzione di nome `foo`, il cui ambiente locale è costituito dai tre nomi `n`, `a`, e `tmp`<sup>2</sup>; le ultime due linee costituiscono gli usi (le chiamate) di `foo`. Nella definizione di `foo`, la prima linea è l'*intestazione*, mentre le restanti linee costituiscono il *corpo* della funzione.

<sup>1</sup>Questi tre termini assumono significati diversi nei diversi linguaggi. Ad esempio, sottoprogramma è in genere il termine più generale, mentre nei linguaggi della famiglia ALGOL e loro discendenti una procedura è un sottoprogramma che modifica lo stato, mentre una funzione è un sottoprogramma che restituisce un valore. Almeno in questo capitolo, useremo i tre termini come sinonimi.

<sup>2</sup>Il nome `foo` fa parte dell'ambiente non locale della funzione.

```

int foo (int m, int b){
    int tmp=b;
    if (tmp==0) return m;
    else return m+1;
}

```

Figura 9.2 Ridenominazione dei parametri formali.

Una funzione scambia informazioni con il resto del programma mediante tre meccanismi principali: i parametri, il valore di ritorno, l'ambiente non locale.

**Parametri** Distinguiamo tra i *parametri formali*, che compaiono nella definizione di una funzione, e i *parametri attuali*, che compaiono invece nella chiamata. I parametri formali sono sempre nomi, che, ai fini dell'ambiente, si comportano come dichiarazioni locali alla funzione stessa.

Si comportano, in particolare, come *variabili legate*, nel senso che una loro ridenominazione consistente non ha effetto sulla semantica della funzione. Ad esempio la funzione `foo` della Figura 9.1 e quella della Figura 9.2 sono indistinguibili, anche se la seconda ha nomi diversi per i parametri formali.

A differenza dei parametri formali, i parametri attuali possono essere, in genere, espressioni (e non semplici nomi). La questione di cosa possa comparire come parametro attuale e, soprattutto, di come avvenga l'accoppiamento tra parametri formali e attuali, sarà trattata estesamente nel Paragrafo 9.1.1.

Il numero e il tipo dei parametri attuali e formali devono in genere coincidere, sebbene possano entrare in gioco molte regole di compatibilità tra tipi (si veda il Paragrafo 10.6) e vi sia talvolta anche la possibilità di dichiarare funzioni con un numero variabile di parametri.

**Valore di ritorno** Oltre che mediante i parametri, alcune funzioni scambiano informazioni col resto del programma anche restituendo un valore come risultato della funzione stessa. La nostra funzione `foo`, ad esempio restituisce un intero. Il linguaggio mette in tal caso a disposizione un meccanismo che consente di esprimere questa “restituzione di valore” (per esempio il costrutto `return`, che ha anche l'effetto di terminare l'esecuzione della funzione corrente). In alcuni linguaggi il nome “funzione” è riservato ai sottoprogrammi che restituiscono un valore, mentre sono chiamate “procedura” quei sottoprogrammi che interagiscono col chiamante solo mediante i parametri o l’ambiente non locale.

Nei linguaggi che derivano la propria sintassi da C, tutti i sottoprogrammi sono, linguisticamente, funzioni. Se il tipo del risultato di una funzione è `void`, la funzione non restituisce alcun valore significativo (e il corrispondente comando per restituire tale valore e terminare l'esecuzione è `return`).

### Le variabili "static"

In tutto quello che abbiamo detto, abbiamo sempre supposto che l'ambiente locale di una funzione abbia lo stesso tempo di vita della funzione stessa. In tale situazione non esiste un meccanismo primitivo col quale una certa istanza di una funzione possa comunicare informazioni con un'altra istanza della *stessa funzione*. L'unico modo in cui ciò può accadere è tramite l'uso di una variabile non-locale.

In alcuni linguaggi, invece, è possibile far sì che una certa variabile (locale ad una funzione) mantenga il proprio valore tra un'invocazione della funzione e la successiva. In C, ad esempio, ciò si ottiene col modificatore `static` (FORTRAN usa `SAVE`, ALGOL own ecc.). Una variabile `static` permette di scrivere funzioni con memoria; la funzione seguente, ad esempio, restituisce quante volte essa stessa era già stata chiamata:

```
int how_many_times() {
    static int count;
    /* C garantisce che una variabile static sia
       inizializzata a zero alla prima
       attivazione della funzione */
    return count++;
}
```

La dichiarazione di una variabile `static` introduce in ambiente un'associazione con tempo di vita illimitato (nell'ambito della vita del programma, ovviamente).

Si osservi come una variabile `static` fornisca maggiore astrazione rispetto ad una variabile globale. Essa non è infatti visibile dall'esterno della funzione: i meccanismi di visibilità garantiscono, dunque, che essa venga modificata solo all'interno corpo.

**Ambiente non locale** Si tratta del meccanismo meno sofisticato col quale una funzione scambia informazione col resto del programma: se il corpo di una funzione modifica una variabile non locale è evidente che questa modifica si ripercuote in tutte le porzioni di programma dove quella variabile è visibile.

### 9.1.1 Astrazione funzionale

Da un punto di vista pragmatico, i sottoprogrammi sono meccanismi che permettono al progettista di software di ottenere *astrazione funzionale*. Una componente software è un'entità che fornisce servizi al suo ambiente. I clienti di tale componente non hanno interesse a conoscere *come* tali servizi sono resi, ma solo come richiederli. La possibilità di associare una funzione ad ogni componente permette di separare quello che il cliente ha bisogno di sapere (espresso nell'intestazione della funzione: il suo nome, i suoi parametri, il tipo del suo risultato, se c'è) da quello che non è bene che conosca (e che è nascosto nel corpo). Si ha vera astrazione funzionale quando il cliente non dipende dal corpo di una funzione, ma solo dalla sua intestazione. In questo caso la sostituzione (ad esempio per motivi di efficienza) del corpo con un altro che abbia la stessa semantica, è trasparente al

sistema software nella sua interezza. Se un sistema gode di astrazione funzionale, le tre azioni di specifica, implementazione e uso di funzione potranno avvenire in modo indipendente l'una dall'altra e senza conoscere il contesto nel quale le altre azioni avverranno.

L'astrazione funzionale è un principio metodologico, al quale le funzioni forniscono un supporto linguistico. È evidente che non si tratta di un supporto definitivo: è necessario che il programmatore utilizzi correttamente le funzioni, per esempio limitando l'interazione tra funzione e chiamante al passaggio dei parametri, dal momento che l'uso dell'ambiente non locale per scambiare informazioni tra funzioni e resto del programma distrugge l'astrazione funzionale. Al contrario, l'astrazione funzionale è tanto più garantita quanto più l'interazione tra componenti è limitata al comportamento esterno, espresso dall'intestazione delle funzioni.

### 9.1.2 Passaggio dei parametri

Le modalità con cui i parametri attuali sono accoppiati ai parametri formali, e la semantica che ne risulta, sono dette *modi*, o *modalità di passaggio dei parametri*. Secondo la terminologia ormai tradizionale, una specifica modalità è costituita sia dal tipo di comunicazione che permette di ottenere, sia dall'implementazione usata per ottenere tale forma di comunicazione. La modalità è fissata al momento della definizione della funzione, può essere distinta da parametro a parametro, e si applica a tutti gli usi della funzione.

Da un punto di vista strettamente semantico, la classificazione del tipo di comunicazione permessa da un parametro è semplice, potendo individuare tre classi di parametri, visti dal punto di vista del sottoprogramma:

- di ingresso;
- di uscita;
- sia di ingresso che di uscita.

Un parametro è di ingresso se permette una comunicazione solo unidirezionale dal chiamante alla funzione (il "chiamato"); è di uscita se permette una comunicazione solo unidirezionale dal chiamato al chiamante; è sia di ingresso che di uscita quando permette una comunicazione bidirezionale.

Si noti che si tratta di una classificazione linguistica, presente nella definizione del linguaggio, e non derivata dall'uso che dei parametri viene fatto: un parametro d'ingresso e d'uscita rimane tale anche se usato solo in modo unidirezionale (ad esempio dal chiamante al chiamato).

È chiaro che ciascuno di questi tipi di comunicazione si può ottenere in molti modi diversi. Le specifiche tecniche implementative costituiscono, appunto, le "modalità di passaggio", che ora passeremo analiticamente in rassegna, descrivendo per ogni modalità:

- quale tipo di comunicazione permetta di stabilire;
- quale forma di parametro attuale sia permessa;

```

int y = 1;
void foo (int x) {
    x = x+1;
}
...
y = 1;
foo(y+1);
// qui y vale 1

```

**Figura 9.3** Passaggio per valore.

- la semantica della modalità;
- l'implementazione canonica;
- il suo costo.

Delle modalità che discuteremo, le prime due (per valore e per riferimento) sono di gran lunga le più importanti e diffuse. Le altre sono più che altro delle varianti sul tema del passaggio per valore. Fa eccezione il passaggio per nome, che discuteremo per ultimo: ormai in disuso come meccanismo di passaggio dei parametri, fornisce tuttavia la possibilità di discutere in un caso semplice di cosa voglia dire “passare un ambiente” ad una procedura.

**Passaggio per valore** Il passaggio per valore è una modalità che corrisponde ad un parametro di ingresso. L'ambiente locale della procedura è esteso con un'associazione tra il parametro formale ed una nuova variabile. Il parametro attuale può essere una generica espressione; al momento della chiamata, l'attuale viene valutato e lo *l*-valore così ottenuto viene assegnato al parametro formale. Al termine della procedura il parametro formale viene distrutto, come tutto l'ambiente locale della procedura stessa. Durante l'esecuzione del corpo, non c'è alcun legame tra il parametro formale e l'attuale. Non c'è alcun modo di sfruttare un parametro per valore per trasferire informazioni dal chiamato al chiamante.

La Figura 9.3 riporta un semplice esempio con un passaggio per valore: analogamente a C, C++, Pascal e Java, quando non indichiamo esplicitamente alcuna modalità di passaggio per un parametro formale, intendiamo che quel parametro è passato per valore. La variabile *y* non cambia mai il suo valore, che rimane sempre 1. Durante l'esecuzione di *foo*, *x* assume valore iniziale 2 per effetto del passaggio di parametro; viene poi incrementata a 3, per poi essere distrutta insieme a tutto il record di attivazione di *foo*.

Se assumiamo un modello di allocazione a pila, il formale corrisponde ad una locazione nel record di attivazione della procedura, nella quale viene memorizzato il valore dell'attuale durante la sequenza di chiamata della procedura stessa.

Osserviamo come si tratti di una modalità costosa nel caso in cui il parametro per valore corrisponda ad una struttura dati di grandi dimensioni: in tal caso

```

int y = 0;
void foo (reference int x) {
    x = x+1;
}
y=0;
foo(y);
// qui y vale 1

```

**Figura 9.4** Passaggio per riferimento.

l'intera struttura viene copiata nel formale<sup>3</sup>. Per contro, il costo dell'accesso al parametro formale è minimo, dal momento che coincide col costo dell'accesso ad una variabile locale al corpo.

Il passaggio per valore è un meccanismo molto semplice e di chiara semantica. Si tratta della modalità *default* di molti linguaggi (per esempio Pascal) ed è l'unico modo per passare parametri in linguaggi come C o Java.

**Passaggio per riferimento** Il passaggio per riferimento (detto anche *per variabile*) realizza un meccanismo nel quale il parametro è sia di ingresso che di uscita. Il parametro attuale *deve* essere un'espressione dotata di *l*-valore (si ricordi la definizione di *l*-valore di pag. 210); al momento della chiamata, viene valutato lo *l*-valore dell'attuale e l'ambiente locale della procedura è esteso con un'associazione tra il parametro formale e lo *l*-valore dell'attuale (creando così una situazione di *aliasing*). Il caso più comune è che il parametro attuale sia una variabile: in questo caso il formale e l'attuale sono due nomi per la stessa variabile. Al termine della procedura, insieme all'ambiente locale viene distrutto anche il legame tra il formale e lo *l*-valore dell'attuale. È chiaro come il passaggio per riferimento consenta una comunicazione bidirezionale: ogni modifica al formale è una modifica all'attuale.

La Figura 9.4 riporta un semplice esempio di passaggio per riferimento (che abbiamo annotato nello pseudocodice con il modificatore *reference*). Durante l'esecuzione di *foo*, *x* è un nome per *y*: l'incremento di *x*, nel corpo, è a tutti gli effetti un incremento di *y*. Dopo la chiamata, il valore di *y* è pertanto 1.

Si noti che, come mostrato in Figura 9.5, il parametro attuale non deve necessariamente essere una variabile ma può essere un'espressione di cui si riesca a determinare al momento della chiamata lo *l*-valore. Analogamente a prima, du-

<sup>3</sup>Il lettore che conosce C non si faccia fuorviare: in linguaggi con puntatori come avremo modo di discutere più avanti, spesso il passaggio di una struttura complessa consiste nel passaggio (per valore) del puntatore alla struttura dati. In tal caso è il puntatore che viene copiato, non la struttura dati.

```

int[] V = new int[10];
int i=0;
void foo (reference int x) {
    x = x+1;
}
...
V[1] = 1;
foo(V[i+1]);
// qui V[1] vale 2

```

**Figura 9.5** Un altro esempio di passaggio per riferimento.

rante l'esecuzione di `foo`, `x` è un nome per `V[1]` e l'incremento di `x`, nel corpo, è un incremento di `V[1]`. Dopo la chiamata, il valore di `V[1]` è pertanto 2.

Nel modello di macchina astratta a pila, al formale è dunque associata una locazione nel record di attivazione della procedura, nella quale la sequenza di chiamata memorizza lo l-valore dell'attuale; l'accesso al formale avviene per via indiretta, attraverso questa locazione.

Si tratta di una modalità di passaggio di costo molto contenuto: al momento della chiamata si deve solo memorizzare un indirizzo, mentre ogni riferimento al formale viene pagato con un accesso indiretto (realizzato implicitamente dalla macchina astratta), che può essere realizzato a basso costo su molte architetture.

Il passaggio per riferimento è un meccanismo di basso livello; possibile in Pascal (modificatore `var`) e in molti altri linguaggi, è stato escluso da linguaggi più moderni. In questi linguaggi, tuttavia, si può ottenere qualche forma di comunicazione bidirezionale tra chiamante e chiamato sfruttando l'interazione tra passaggio dei parametri e altri meccanismi, in primo luogo il modello scelto per le variabili, o la disponibilità di puntatori nel linguaggio. Due riquadri riportano semplici esempi in C (e C++) e Java. La morale degli esempi è che in un linguaggio imperativo il passaggio per valore è sempre completato con altri meccanismi, affinché le procedure siano davvero uno strumento di programmazione versatile.

**Passaggio per costante** Abbiamo già osservato come il passaggio per valore sia costoso per dati di grandi dimensioni. Qualora però il parametro formale non riceva alcuna modifica nel corpo della funzione, possiamo immaginare di mantenere la semantica del passaggio per valore, implementandola attraverso il passaggio per riferimento. È quello che fa la modalità di passaggio per costante (o *read-only*).

Si tratta di una modalità che stabilisce una comunicazione di ingresso e nella quale sono messe generiche espressioni come parametro attuale. I parametri formali passati con questa modalità sono soggetti al vincolo statico di non poter essere modificati nel corpo, sia direttamente (un assegnamento) sia indirettamente (attraverso chiamate a funzioni che li modifichino). Da un punto di vista seman-

### Passaggio per riferimento in C

C ammette il solo passaggio per valore, ma permette anche la libera manipolazione di puntatori e indirizzi. Sfruttando questo fatto non è difficile simulare il passaggio per riferimento. Consideriamo il problema di scrivere una semplice funzione che scambi il valore di due variabili intere che sono passate alla funzione come parametri. Col solo passaggio per valore non c'è alcun modo per farlo. Possiamo tuttavia combinare il passaggio per valore con la manipolazione di puntatori, come nell'esempio che segue.

```

void swap (int *a, int *b) {
    int tmp = *a; *a=*b; *b=tmp;
}
int v1 = ...;
int v2 = ...;
swap (&v1, &v2);

```

I parametri formali (per valore) di `swap` sono di tipo puntatore a intero (`int *`); i parametri attuali, sfruttando l'operatore `&`, hanno come valore l'indirizzo (cioè lo l-valore) di `v1` e `v2`. Nel corpo di `swap`, l'uso dell'operatore (prefisso) `*` permette di dereferenziare i parametri. Ad esempio, possiamo parafrasare il significato di `*a=*b;` come: prendi il valore contenuto nella locazione il cui indirizzo è contenuto in `b` e memorizzalo nella locazione il cui indirizzo è contenuto in `a`. La nostra `swap` simula dunque in tutto un passaggio per riferimento.

### Comunicazione bidirezionale in Java

Una funzione come `swap` (nel riquadro "Passaggio per riferimento in C") non si può scrivere in Java. Tuttavia, possiamo sfruttare il fatto che Java adotta un modello delle variabili a riferimento (per i tipi classe) per ottenere qualche forma di comunicazione bidirezionale. Consideriamo ad esempio la seguente semplice definizione di classe:

```

class A {
    int v;
}

```

Possiamo certamente scrivere un metodo che scambia i valori del campo `v` di due oggetti di classe `A`:

```

void swap (A a, A b) {
    int tmp = a.v; a.v= b.v; b.v=tmp;
}

```

In questo caso, quello che è passato (per valore) a `swap` sono due riferimenti ad oggetti di classe `A`; sfruttando il modello a riferimento delle variabili di tipo classe, `swap` effettivamente scambia i valori dei campi. Si osservi, tuttavia, che non è possibile una vera simulazione del passaggio per riferimento, come invece è possibile in C.

### Passaggio per costante in Java e C

Il passaggio per costante esiste (con diverse varianti) in molti linguaggi. In Java lo si ottiene col modificatore `final`:

```
int succ(final int x) {
    return x+1;
}
```

L'analogo modificatore in C è `const`. La differenza sostanziale tra C e Java è che, sebbene in entrambi i linguaggi il compilatore controlli che nel corpo della funzione non vi siano assegnamenti al parametro passato per costante, nel caso di C un'eventuale violazione si traduce solo in un *warning*: la situazione viene segnalata, ma il programma viene comunque compilato. Il risultato di una modifica ad un parametro passato come `const` è "dipendente dall'implementazione", come dice la definizione di C. Java, invece, classifica come sintatticamente scorretto un programma nel quale si tenti la modifica di un parametro `final`.

tico, il passaggio per costante coincide in tutto e per tutto col passaggio per valore, mentre viene lasciata alla macchina astratta la scelta dell'implementazione. Per dati di piccole dimensioni, il passaggio per costante potrà essere implementato come il passaggio per valore; per strutture dati più grandi, si sceglierà di implementarlo mediante un riferimento, senza copia.

Il passaggio per costante è un ottimo modo per "annotare" un certo parametro di una procedura: leggendo l'intestazione si ha immediatamente l'informazione sul fatto che quel parametro è solo di ingresso; inoltre, si può contare sul controllo di semantica statica per verificare che quell'annotazione è davvero garantita.

**Passaggio per risultato** Il passaggio per risultato è l'esatto duale del passaggio per valore. Si tratta di una modalità che realizza una comunicazione di sola uscita. L'ambiente locale della procedura è esteso con un'associazione tra il parametro formale e una nuova variabile. Il parametro attuale deve essere un'espressione dotata di l-valore. Al momento della terminazione (normale) della procedura, subito prima della distruzione dell'ambiente locale, il valore corrente del parametro formale viene assegnato alla locazione ottenuta mediante lo l-valore dell'attuale. Dovrebbe esser chiaro che con il passaggio per valore devono essere precise diverse questioni, collegate all'ordine di valutazione: se vi è più di un parametro per risultato, in quale ordine vengono eseguiti i corrispondenti "assegnamenti all'indietro" dall'attuale al formale (per esempio, da sinistra a destra)? E ancora, quando viene determinato lo l-valore del parametro attuale? È giustificabile sia la scelta di determinarlo al momento della chiamata della funzione, che quella di fissarlo solo al momento della sua terminazione<sup>4</sup>. Si osservi che durante

<sup>4</sup>Si costruisca un esempio che dia luogo a risultati diversi a seconda se lo l-valore dell'attuale è determinato al momento della chiamata o della terminazione

```
void foo (result int x) {x = 8;}
...
int y = 1;
foo(y);
// qui y vale 8
```

Figura 9.6 Passaggio per risultato.

```
void foo (valueresult int x) {
    x = x+1;
}
...
y = 8;
foo(y);
// qui y vale 9
```

Figura 9.7 Passaggio per valore-risultato.

l'esecuzione del corpo, non c'è alcun legame tra il parametro formale e l'attuale. Non c'è alcun modo di sfruttare un parametro per risultato per trasferire informazioni dal chiamante al chiamato. Un esempio di passaggio per risultato è mostrato in Figura 9.6. L'implementazione del passaggio per risultato è analoga a quella del passaggio per valore, di cui condivide anche pregi e difetti. Da un punto di vista pragmatico, la modalità per risultato rende agevole il progetto di funzioni che devono restituire (cioè fornire come risultato) più valori in variabili distinte.

**Passaggio per valore-risultato** La combinazione del passaggio per valore e del passaggio per risultato dà luogo alla modalità detta per valore-risultato. Si tratta di una modalità che realizza una comunicazione bidirezionale, col formale che è a tutti gli effetti una variabile locale alla procedura. Il parametro attuale deve essere un'espressione dotata di l-valore; al momento della chiamata, il parametro attuale viene valutato e lo r-valore così ottenuto viene assegnato al parametro formale. Al termine della procedura, subito prima della distruzione dell'ambiente locale, il valore corrente del parametro formale viene assegnato alla locazione corrispondente al parametro attuale. Durante l'esecuzione del corpo, non c'è alcun legame tra il parametro formale e l'attuale. Un esempio di passaggio per valore-risultato è mostrato in Figura 9.7.

L'implementazione canonica del passaggio per valore-risultato è analoga a quella del passaggio per valore, anche se alcuni linguaggi (Ada, per esempio) scelgono di implementarlo come il passaggio per riferimento nel caso di dati di grandi dimensioni, per ovviare ai problemi di costo tipici del passaggio per valore.

```

void foo (reference/valueresult int x,
          reference/valueresult int y,
          reference int z){
    y = 2;
    x = 4;
    if (x == y) z = 1;
}

int a = 3;
int b = 0;
foo(a,a,b);

```

**Figura 9.8** Il passaggio per valore-risultato non è il passaggio per riferimento.

L'implementazione del passaggio per valore-risultato mediante un riferimento, tuttavia, non è corretta semanticamente. Si consideri, infatti, il frammento della Figura 9.8. A prima vista, il comando condizionale presente nel corpo di `foo` sembra inutile: `x` e `y` hanno appena ricevuto valori distinti. La realtà è che `x` e `y` hanno valori distinti solo *in assenza di aliasing*; se, al contrario, `x` e `y` sono due nomi diversi per una stessa variabile, è evidente che la condizione `x == y` è sempre vera.

Se, dunque, `x` e `y` sono passati per valore-risultato (non c'è aliasing), la chiamata `foo(a, a, b)` termina senza che il valore di `b` sia modificato. Se, invece, `x` e `y` sono passati per riferimento (c'è aliasing), `foo(a, a, b)` termina assegnando a `b` il valore 1.

**Passaggio per nome** Il passaggio per nome, introdotto in ALGOL come modalità semanticamente più pulita del passaggio per riferimento, non esiste oggi in nessun linguaggio imperativo di rilievo; tuttavia, per le sue caratteristiche e la sua implementazione, è una modalità di grande importanza concettuale, che vale la pena di studiare in dettaglio.

Il problema che i progettisti di ALGOL si posero era quello di dare una semantica *precisa*, ma allo stesso tempo espressa in modo elementare, di quale sia l'effetto di una chiamata di funzione con determinati parametri. La soluzione che scelsero fu quella di definire la semantica di una chiamata di funzione mediante la cosiddetta *regola di copia*. Senza perdita di generalità, la enunceremo nel caso di una funzione con un unico parametro:

Sia `f` una funzione con unico parametro formale `x` e sia `a` un'espressione compatibile col tipo di `x`. Una chiamata `a f` con parametro attuale `a` è semanticamente equivalente all'esecuzione del corpo di `f` nel quale tutte le occorrenze del parametro formale `x` sono state sostituite con `a`.

Si tratta, come ben si vede, di una regola molto semplice, che riduce la semantica della chiamata di una funzione ad un'operazione sintattica: quella dell'espansio-

```

int x=0;
int foo (name int y){
    int x = 2;
    return x + y;
}
...
int a = foo(x+1);

```

**Figura 9.9** Quale ambiente usare per valutare `x+1` nel corpo di `foo`?

ne del corpo della funzione dopo una sostituzione testuale del parametro attuale al posto del formale. Questa nozione di sostituzione, tuttavia, non è un concetto banale, perché occorre tener conto dell'eventuale presenza di più variabili distinte con lo stesso nome. Si consideri, ad esempio, la funzione della Figura 9.9. Se applichiamo la regola di copia senza cautela alcuna, la chiamata `foo(x+1)` risulta nell'esecuzione di `return x+x+1`. Questo comando, valutato nell'ambiente locale di `foo`, restituisce il valore 5. Ma è chiaro che si tratta di una cattiva applicazione della regola di copia, perché fa dipendere il risultato della funzione dallo specifico nome della variabile locale. Se il corpo di `foo` fosse stato

```
{int z=2; return z+y;}
```

la stessa chiamata sarebbe risultata nell'esecuzione di `return z+x+1`, con risultato 3.

Nella prima sostituzione che abbiamo proposto, diciamo che la `x` del parametro attuale è stata *catturata* dalla dichiarazione locale. La sostituzione di cui parla la regola di copia deve essere dunque una sostituzione *senza cattura di variabile*. Non potendo impedire che vi siano più variabili distinte con lo stesso nome, si ottiene una sostituzione senza cattura richiedendo che il parametro formale, anche dopo la sostituzione, venga valutato nell'ambiente del chiamante e non in quello del chiamato.

Possiamo quindi definire il passaggio per nome come quella modalità la cui semantica è data dalla regola di copia, nella quale però la nozione di sostituzione è sempre da intendersi senza cattura; equivalentemente, possiamo dire che ciò che viene sostituito non è il mero parametro attuale, ma il parametro attuale *insieme al proprio ambiente di valutazione*, che è fissato al momento della chiamata.

Si osservi come la regola di copia imponga che il parametro attuale sia valutato *ogni volta* che il formale viene incontrato durante l'esecuzione, con le relative conseguenze in presenza di eventuali effetti collaterali. Si consideri l'esempio della Figura 9.10, dove il costrutto `i++` ha la semantica di restituire il valore corrente della variabile `i` e, successivamente, incrementare di uno il valore di `i`.

La regola di copia impone che il costrutto `i++` sia valutato due volte: una volta per ogni occorrenza del parametro formale `y` in `f(y)`. La prima volta, la sua valutazione restituisce il valore 2 e incrementa il valore di `i` di uno; la seconda volta restituisce il valore 3 e incrementa nuovamente `i`.

```

int i = 2;
int fie (name int y) {
    return y+y;
}
...
int a = fie(i++);
// qui i ha valore 4; a ha valore 5

```

**Figura 9.10** Effetti collaterali nel passaggio per nome.

```

void fiefoo (valueresult/name int x, valueresult/name int y) {
    x = x+1;
    y = 1;
}
...
int i = 1;
int[] A = new int[5];
A[1]=4;
fiefoo(i,A[i]);

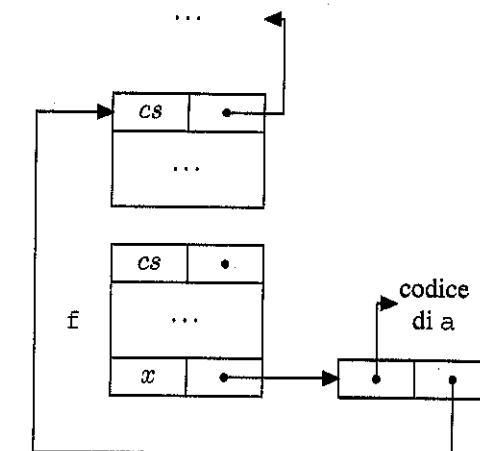
```

**Figura 9.11** Il passaggio per nome non è il passaggio per valore-risultato.

Già l'esempio appena discusso mostra come sia errato considerare il passaggio per nome come un modo complicato per descrivere il passaggio per valore-risultato. Le due modalità sono semanticamente diverse, come la Figura 9.11 si incarica di mostrare.

Supponiamo per prima cosa di eseguire `fiefoo`, con i parametri passati per valore-risultato: al termine dell'esecuzione avremo `A[1]` con valore 1 e `i` con valore 2, mentre il resto dell'array `A` non è stato toccato dall'esecuzione di `fiefoo`. Se, invece, eseguiamo la stessa procedura con i due parametri passati per nome, al termine avremo `A[1]` con valore 4, `i` con valore 2 e, cosa ancora più importante, anche l'elemento `A[2]` viene modificato, con valore 1. Si osservi che, in questo caso, valore-risultato e passaggio per riferimento avrebbero mostrato lo stesso comportamento.

Rimane da descrivere come il passaggio per nome possa essere implementato. Abbiamo già discusso della necessità per il chiamante di passare non solo l'espressione testuale che costituisce il parametro attuale, ma anche l'ambiente nel quale tale espressione deve essere valutata. Chiamiamo *chiusura* una coppia (espressione, ambiente), nella quale l'ambiente comprenda (almeno) tutte le varia-



**Figura 9.12** Implementazione del passaggio per nome.

bili libere presenti nell'espressione<sup>5</sup>. Possiamo pertanto dire che, nel caso del passaggio per nome, il chiamante passa una chiusura, costituita dal parametro attuale (come espressione testuale) e dall'ambiente corrente. Quando il chiamato incontra un riferimento al parametro formale, tale riferimento viene risolto mediante la valutazione della prima componente della chiusura nell'ambiente costituito dalla seconda componente. La Figura 9.12 descrive questa situazione nel caso particolare in cui si abbia a che fare con una macchina astratta con allocazione a pila. In tal caso una chiusura è una coppia costituita da due puntatori: il primo punta al codice di valutazione dell'espressione che costituisce il parametro formale; il secondo è un puntatore di catena statica, il quale indica il blocco che costituisce l'ambiente locale nel quale valutare l'espressione. Al momento della chiamata di una procedura `f` con parametro formale per nome `x` e parametro attuale `a`, il chiamante costruisce una chiusura la cui prima componente è un puntatore al codice di `a` e la cui seconda componente è un puntatore al proprio (del chiamante) record di attivazione; lega poi questa chiusura (per esempio mediante un altro puntatore) al parametro formale `x`, che risiede nel record di attivazione del chiamato.

Possiamo finalmente riassumere quello che sappiamo sul passaggio per nome. Il passaggio per nome è una modalità che corrisponde ad un parametro sia di ingresso che di uscita. Il parametro formale *non* corrisponde ad una variabile locale alla procedura; il parametro attuale può essere una generica espressione. Vi è la possibilità di aliasing tra formale e attuale; l'attuale deve valutare ad un

<sup>5</sup>Il termine "chiusura" proviene dalla logica matematica: una formula è chiusa quando non ha variabili libere. Una chiusura è un modo canonico per trasformare una porzione di codice con variabili non locali (cioè "libere") in un codice completamente specificato.

l-valore se il formale compare a sinistra di un assegnamento. La semantica della chiamata per nome è stabilita dalla regola di copia, che consente di mantenere durante l'esecuzione un legame costante tra parametro formale e parametro attuale. L'implementazione canonica è quella fornita da una chiusura: l'ambiente locale della procedura è esteso con un'associazione tra il formale e una chiusura, costituita dal parametro attuale e dall'ambiente in cui occorre la chiamata; ogni accesso al formale viene risolto mediante una valutazione *ex novo* del parametro attuale nell'ambiente fornito dalla chiusura. Si tratta di una modalità di passaggio dei parametri molto costosa, sia per la necessità di passare una struttura complessa, sia, soprattutto, per la valutazione ripetuta del parametro attuale in un ambiente diverso da quello corrente.

**Approfondimento**

9.1

## 9.2 Funzioni di ordine superiore

Una funzione è *di ordine superiore* qualora abbia come parametro, o restituisca come risultato, un'altra funzione; sebbene non vi sia accordo unanime nella letteratura, diremo che un linguaggio di programmazione è di ordine superiore qualora ammetta funzioni sia come parametro che come risultato di altre funzioni. Linguaggi con funzioni come parametro sono abbastanza diffusi; sono meno diffusi, invece, linguaggi che permettano di restituire una funzione come risultato. Quest'ultimo tipo di operazione, comunque, è uno dei meccanismi fondamentali dei linguaggi di programmazione *funzionale*, che tratteremo estesamente nel Capitolo 13. Discuteremo in questo paragrafo dei problemi linguistici e implementativi che si incontrano nei due casi, trattandoli separatamente.

### 9.2.1 Funzioni come parametro

Il caso generale che vogliamo analizzare è quello di un linguaggio con parametri funzionali, ambienti annidati e possibilità di definizione di funzioni ad ogni livello di annidamento<sup>6</sup>. Prendiamo in esame l'esempio riportato in Figura 9.13: con la notazione `void g (int h(int n)){...}` intendiamo, nel nostro pseudolinguaggio, la dichiarazione della funzione `g`, con unico parametro formale `h`, a sua volta specificato essere una funzione che restituisce un `int` ed ha un proprio parametro formale di tipo `int`<sup>7</sup>. I due punti chiave dell'esempio sono (i) il fatto che `f` è passata come parametro attuale a `g` e successivamente chiamata tramite il

```

1   int x = 1;
2   int f(int y){
3       return x+y;
4   }
5   void g (int h(int b)){
6       int x = 2;
7       return h(3) + x;
8   }
9   ...
10  int z = g(f);
11 }
12
13

```

Figura 9.13 Parametri funzionali.

parametro formale `h`; e (ii) vi è una definizione multipla del nome `x`, cosicché è necessario stabilire quale sia l'ambiente (non locale) in cui `f` sarà valutata. Relativamente a questa seconda questione, il lettore non si sorprenderà se osserviamo che vi sono due possibilità per l'ambiente non locale da utilizzare al momento dell'esecuzione di una funzione `f` invocata tramite un parametro formale `h`:

- utilizzare l'ambiente attivo al momento della *creazione del legame* tra `h` e `f` (che avviene alla linea 11); diciamo in questo caso che il linguaggio adotta una politica di *deep binding*;
- utilizzare l'ambiente attivo al momento della *chiamata* di `f` tramite `h` (che avviene alla linea 7); diciamo in questo caso che il linguaggio adotta una politica di *shallow binding*<sup>8</sup>.

Sebbene le due possibilità di binding richiamino immediatamente la distinzione tra scope statico e dinamico, sottolineiamo subito come la politica di binding (nel caso di funzioni di ordine superiore) sia da considerarsi un ulteriore grado di variabilità, indipendente dalla politica di scope. Tutti i principali linguaggi con scope statico adottano deep binding (perché la scelta di una politica shallow apparirebbe contraddittoria a livello metodologico); la cosa non è altrettanto pacifica per i linguaggi con scope dinamico, tra i quali troviamo sia linguaggi con deep binding che con shallow binding.

Riprendendo l'esempio di Figura 9.13, le diverse modalità di scope e di binding danno luogo ai seguenti comportamenti:

<sup>6</sup>C permette funzioni come parametro, ma è possibile definire una funzione solo nell'ambiente globale. Con questa limitazione il problema risulta estremamente semplificato (e invero questa semplicità è proprio uno dei motivi per i quali C non ammette funzioni annidate).

<sup>7</sup>Il nome del parametro formale di `h` (in questo caso `n`), non è di nessuna utilità e non ci sono modi col quale il programmatore possa utilizzarlo nel corpo di `g`.

- con scope statico e deep binding, la chiamata `h(3)` restituisce 4 (e `g` restituisce 6): la `x` nel corpo di `f` al momento della sua chiamata tramite `h` è quella del blocco più esterno;
- con scope dinamico e deep binding, la chiamata `h(3)` restituisce 7 (e `g` restituisce 9): la `x` nel corpo di `f` al momento della sua chiamata tramite `h` è quella locale al blocco nel quale avviene la chiamata `g(f)`;
- con scope dinamico e shallow binding, la chiamata `h(3)` restituisce 5 (e `g` restituisce 7): la `x` nel corpo di `f` al momento della sua chiamata tramite `h` è quella locale a `g`.

**Implementazione del deep binding** Lo shallow binding non pone ulteriori problemi implementativi rispetto alle tecniche utilizzate per lo scope dinamico: basta, almeno concettualmente, ricercare per ogni nome la sua ultima associazione presente. Non è così, invece, per il deep binding, che richiede strutture dati ausiliarie rispetto all'ordinaria catena statica o dinamica.

Per fissare le idee, poniamoci nel caso di un linguaggio con scope statico e deep binding (il caso dello scope dinamico è lasciato al lettore, Esercizio 6). Dal Paragrafo 7.5.1, già sappiamo che all'invocazione diretta (cioè che non avviene tramite un parametro formale) di una funzione `f` viene associata staticamente un'informazione (un intero) che esprime il livello di annidamento della definizione di `f` rispetto al blocco nel quale avviene la chiamata. Quest'informazione è utilizzata dinamicamente dalla macchina astratta per inizializzare opportunamente il puntatore di catena statica (cioè l'ambiente non locale) del record di attivazione di `f`. Quando, invece, una funzione `f` venga invocata tramite un parametro formale `h`, alla chiamata non può essere associata alcuna informazione, proprio perché si utilizza un parametro formale che, nel corso di diverse attivazioni della procedura in cui si trova, può essere associato a diverse funzioni (questo è il caso, ad esempio, della chiamata `h(3)` nella Figura 9.13).

È evidente, in altre parole, che con deep binding l'informazione relativa al puntatore di catena statica deve essere determinata al momento dell'associazione tra il parametro formale e il parametro attuale: al formale `h` deve essere associato non solo il codice di `f`, ma anche l'ambiente non locale nel quale il corpo di `f` deve essere valutato. Tale ambiente non locale può venire determinato in modo semplice: in corrispondenza di una chiamata della forma `g(f)` (la procedura `g` è chiamata con parametro funzionale attuale `f`), possiamo associare staticamente al parametro `f` l'informazione relativa al livello di annidamento della definizione di `f` rispetto al blocco nel quale compare la chiamata `g(f)`. Al momento dell'esecuzione di tale chiamata, la macchina astratta userà tale informazione per associare al parametro formale corrispondente ad `f` sia il codice di `f`, sia un puntatore al record di attivazione del blocco all'interno del quale `f` è dichiarata (tale puntatore viene determinato con le stesse regole discusse nel Paragrafo 7.5.1).

Al parametro formale funzionale viene dunque associata logicamente una coppia (testo, ambiente), rappresentata a livello implementativo da una coppia (puntatore al codice, puntatore ad un record di attivazione). Abbiamo già visto quando abbiamo discusso dell'implementazione del passaggio per nome che una

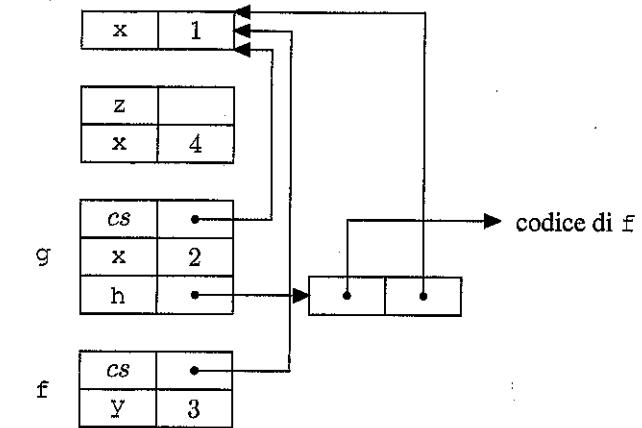


Figura 9.14 Chiusure e catena statica.

tale struttura dati si chiama *chiusura*. Dunque al parametro formale funzionale è associata una chiusura: al momento in cui il formale è usato per invocare una funzione (staticamente ignota), la macchina astratta trova il codice a cui trasferire il controllo nella prima componente della chiusura, e assegna il contenuto della seconda componente della chiusura al puntatore di catena statica del record di attivazione della nuova invocazione.

La Figura 9.14 riporta la situazione della pila di attivazione relativamente al codice della Figura 9.13 nel momento in cui si entra nella funzione `f` (chiamata attraverso `h` dal corpo di `g`). Al momento in cui la funzione `g` è chiamata con parametro attuale `f`, viene legata a `h` una chiusura: `f` è dichiarata a distanza 1 rispetto al luogo nel quale appare come parametro attuale (è infatti dichiarata all'interno del blocco *che contiene* quello in cui appare come parametro attuale). La seconda componente della chiusura è dunque determinata risalendo di un passo la catena statica (ottenendo un puntatore al blocco più esterno). Al momento in cui `f` è chiamata per il tramite del nome `h`, viene messo sulla pila il corrispondente record di attivazione. Il valore del puntatore di catena statica viene preso dalla seconda componente della chiusura.

Il lettore ricordi, ancora una volta, che i problemi che stiamo discutendo compaiono solo qualora il linguaggio permetta la definizione di funzioni con ambiente non locale, cioè permetta la definizione di funzioni *all'interno di blocchi annidati*. In caso contrario, per esempio in C, non essendovi ambiente non locale, non c'è nemmeno bisogno di chiusure: per passare una funzione basta passare un puntatore al suo codice. Tutti i riferimenti non locali nel corpo della funzione saranno risolti nell'ambiente globale.

```

void foo (int f(), int x) {
    int fie() {
        return x;
    }
    int z;
    if (x==0) z=f();
    else foo(fie,0);
}
int g() {
    return 1;
}
foo(g,1);
}

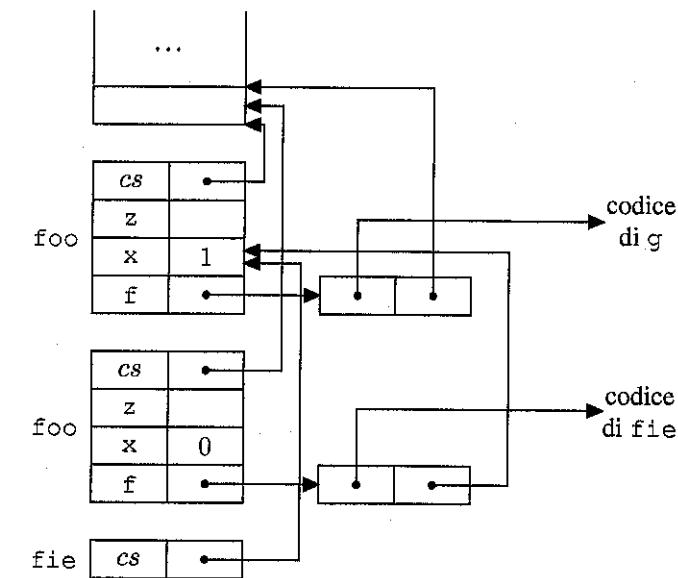
```

**Figura 9.15** La politica di binding è necessaria per determinare l'ambiente.

**Politiche di binding e scope statico** Abbiamo già osservato come tutti i linguaggi con scope statico adottino deep binding. A prima vista, anzi, potrebbe sembrare che deep o shallow binding non facciano alcuna differenza nel caso di scope statico: dopo tutto l'ambiente non locale di una funzione è determinato dalla posizione (statica) della sua dichiarazione e non dai modi della sua invocazione. Nel caso della Figura 9.13, sono le regole di scope (e non di binding) a stabilire che ogni invocazione di *f* (direttamente tramite il suo nome, oppure indirettamente tramite un parametro formale) sia valutata nell'ambiente non locale più esterno.

In generale, tuttavia, non è così. La ragione è da ricercare nella possibilità che sulla pila siano presenti più record di attivazione corrispondenti alla stessa funzione (il caso si presenta nel caso di funzioni ricorsive, o mutuamente ricorsive). Se dall'interno di una di queste attivazioni viene passata per parametro una procedura, si può creare una situazione nella quale le sole regole di scope non sono sufficienti a determinare quale ambiente non locale va utilizzato al momento dell'invocazione del parametro funzionale. A titolo d'esempio discuteremo il codice della Figura 9.15, che assumiamo scritto, come sempre, in uno pseudolinguaggio con scope statico.

Il cuore del problema è costituito dal riferimento (non locale) ad *x* all'interno di *fie*. Le regole di scope ci dicono che tale *x* si riferisce al parametro formale di *foo* (che per altro è la sola *x* dichiarata nel programma). Ma nel momento in cui *fie* viene invocata (tramite il formale *f*), vi sono attive due istanze di *foo* (e dunque due istanze del suo ambiente locale): una prima, attivata mediante la chiamata *foo(g,1)*, nella quale ad *x* è associata (una locazione che contiene) il valore 1; una seconda, attivata mediante la chiamata (ricorsiva) *foo(fie,0)*, nella quale ad *x* è associato il valore 0. È all'interno di questa seconda attivazione che avviene la chiamata a *fie* via *f*. Le regole di scope nulla dicono di quale delle due istanze di *x* si debba usare nel corpo di *f*. È a questo punto che interviene la politica di binding: usando deep binding, l'ambiente è stabilito al momento della



**Figura 9.16** Pila di attivazione nell'esempio di Figura 9.15.

creazione dell'associazione tra *fie* e *f*, cioè mentre *x* è associato al valore 1: a *z* sarà pertanto assegnato 1. Per facilitare la comprensione dell'esempio, la Figura 9.16 riporta la situazione della pila e delle chiusure al momento in cui *fie* viene eseguita.

In caso di shallow binding (che, lo ripetiamo, non è usato in regime di scope statico), l'ambiente sarebbe stato determinato al momento dell'invocazione di *f*: a *z* sarebbe stato assegnato 0.

**Cosa definisce l'ambiente** Prima di chiudere questo paragrafo, riprendiamo il problema, affrontato nel Capitolo 6, di quali regole intervengano nella determinazione dell'ambiente. Possiamo finalmente completare gli ingredienti che concorrono a determinare correttamente l'ambiente di valutazione in un linguaggio strutturato a blocchi. Sono necessarie:

- le regole di visibilità, di norma garantite dalla struttura a blocchi;
- le eccezioni alle regole di visibilità (che tengono conto, per esempio, della ridefinizione dei nomi e della possibilità o meno di utilizzare un nome prima della sua dichiarazione);
- le regole di scope;
- le regole relative alle modalità di passaggio dei parametri;
- la politica di binding.

### I thunk

Al lettore non sarà sfuggita la similarità tra l'implementazione del passaggio per nome e quella necessaria ai parametri funzionali. A ben vedere, infatti, un parametro formale per nome può essere considerato a tutti gli effetti come un parametro funzionale (senza argomenti). Analogamente, il relativo parametro attuale può essere considerato come la definizione (anonima, perché non introduce un nome) di una funzione senza argomenti: quella che corrisponde al codice di valutazione del parametro attuale. Durante l'esecuzione del corpo, ogni occorrenza del parametro per nome corrisponde ad una valutazione *ex novo* del parametro attuale, cioè proprio ad una nuova chiamata della funzione anonima corrispondente al parametro attuale, nell'ambiente fissato al momento dell'associazione tra parametro attuale e formale per nome.

Il gergo ALGOL ha introdotto il nome *thunk* per una struttura siffatta: una funzione senza argomenti ed il relativo ambiente di valutazione. Nel passaggio per nome, dunque, viene introdotto in ambiente un legame tra il parametro formale ed un thunk.

### L'ambiente in C

La struttura dell'ambiente in C è particolarmente semplice. Un programma C consiste infatti in una sequenza di dichiarazioni di variabili e di funzioni; le variabili dichiarate in questo modo (che in gergo C si chiamano variabili *esterne*) sono visibili in qualsiasi punto del programma: sono variabili globali, secondo la terminologia del Paragrafo 6.2.2. Al loro interno le funzioni sono strutturate in blocchi, e in ogni blocco possono essere dichiarate variabili locali, ma non è ammessa la definizione di funzioni all'interno di un'altra funzione.

L'ambiente di una funzione, pertanto, è composto da una parte locale e una parte globale: ogni riferimento ad un nome non locale viene risolto in modo univoco nell'ambiente globale. Con questa struttura semplificata, la gestione dell'ambiente è semplicissima: non occorre mantenere la catena statica e per passare una funzione come parametro è sufficiente passare un puntatore al codice.

Per motivi di efficienza, inoltre, non c'è alcuna gestione dinamica dei blocchi in-line: le variabili dichiarate in un qualsiasi blocco di una funzione sono allocate nel RdA della funzione.

L'efficienza d'esecuzione è uno dei principali obiettivi di C: per evitare il costo della gestione della pila degli RdA, il compilatore può scegliere di tradurre una chiamata di funzione mediante espansione del suo corpo (nel caso di una funzione ricorsiva, l'espansione avviene una sola volta).

### 9.2.2 Funzioni come risultato

La possibilità di generare funzioni come risultato di altre funzioni permette la creazione dinamica di funzioni a tempo d'esecuzione. È evidente come, in generale, la funzione restituita come risultato non potrà essere rappresentata a tempo

```
int x = 1;
void->int F () {
    int g () {
        return x+1;
    }
    return g;
}
void->int gg = F();
int z = gg();
```

**Figura 9.17** Funzioni come risultato.

d'esecuzione dal suo solo codice: sarà necessario anche l'ambiente nel quale tale funzione dovrà essere valutata. Consideriamo il primo, semplice esempio della Figura 9.17. Fissiamo innanzitutto la notazione: con `void->int` abbiamo indicato il *tipo* delle funzioni che non prendono alcun argomento (`void`) e restituiscono un valore di tipo `int`. La seconda linea del codice è dunque la dichiarazione di una funzione `F` che restituisce una funzione senza argomenti la quale restituisce a sua volta un intero (si osservi che `return g` restituisce "la funzione", non una sua applicazione). La prima linea dopo il corpo di `F` è la dichiarazione del nome `gg` al quale è associato (dinamicamente) il risultato della valutazione di `F`.

Non dovrebbe essere difficile convincersi che la funzione `gg` restituisce il successore del valore di `x`. In regime di scope statico, tale `x` è fissato dalla struttura del programma e non dalla posizione della chiamata a `gg`, che potrebbe comparire in un ambiente nel quale è presente un'altra definizione di un nome `x`.

Possiamo pertanto dire che, in generale, quando una funzione restituisce una funzione come risultato, tale risultato è una *chiusura*; di conseguenza, la macchina astratta dovrà essere opportunamente modificata per tener conto della chiamata ad una chiusura. Analogamente a quello che succede quando una funzione è invocata tramite un parametro formale, quando viene invocata una funzione il cui valore è stato ottenuto dinamicamente (come `gg`) il puntatore di catena statica del suo record di attivazione viene determinato utilizzando la chiusura associata (e non le regole canoniche discusse nel Paragrafo 7.5.1, che non sarebbero di alcun aiuto).

La situazione generale, tuttavia, è assai più complessa. Se è possibile restituire una funzione dall'interno di un blocco annidato, si può creare la possibilità che il suo ambiente di valutazione faccia riferimento a un nome che, secondo la disciplina a pila, andrebbe distrutto. Consideriamo, infatti, il codice della Figura 9.18, dove abbiamo spostato la dichiarazione di `x` all'interno di `F`. La Figura 9.19 presenta uno schema dei record di attivazione al momento della chiamata `gg()`.

Quando alla funzione `gg` è assegnato il risultato di `F()`, la chiusura che costituisce il suo valore punta ad un ambiente che contiene il nome `x`; ma tale ambiente è locale a `F` e verrebbe dunque distrutto alla sua terminazione. Come è dunque

```

void->int F () {
    int x = 1;
    int g () {
        return x+1;
    }
    return g;
}
void->int gg = F();
int z = gg();

```

**Figura 9.18** Funzioni come risultato e disciplina a pila.

possibile invocare *successivamente* gg, senza incorrere in una dangling reference per x? La risposta non può che essere drastica: abbandonando la disciplina a pila per i record di attivazione, che rimangono pertanto in vita indefinitamente, perché potrebbero costituire l'ambiente di funzioni che verranno valutate solo in seguito. In linguaggi con le caratteristiche che abbiamo appena discusso, gli ambienti locali hanno tempo di vita *illimitato*.

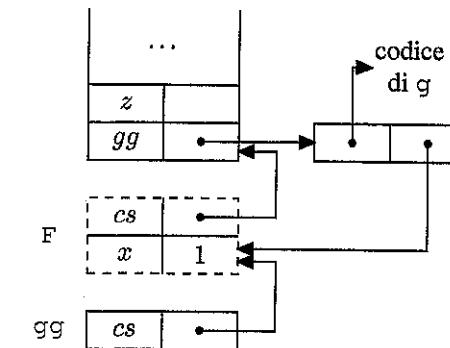
La soluzione più comune in questo caso è quella di allocare tutti i record di attivazione sullo heap, e lasciare ad un garbage collector la responsabilità di deallocarli quando si scoprissse che non vi sono riferimenti verso i nomi in essi contenuti.

Tutti i linguaggi funzionali sono costruiti attorno alla possibilità di restituire funzioni come risultato e devono pertanto affrontare di petto questo problema. Al contrario, proprio per mantenere una disciplina a pila per i record di attivazione, nei linguaggi imperativi la possibilità di restituire funzioni è una caratteristica rara. Nei linguaggi che lo permettono vi sono in genere numerose restrizioni volte a garantire che non si possa mai creare un riferimento ad un ambiente ormai disattivato (per esempio: non vi sono funzioni annidate (C, C++); è possibile restituire solo funzioni non annidate (Modula-2 e Modula-3); si vincola opportunamente lo scope delle funzioni annidate che sono restituite (Ada); ecc.).

### 9.3 Eccezioni

Un'eccezione è un evento particolare che si verifica durante l'esecuzione di un programma e che non deve (o non può) esser gestito dal normale flusso del controllo. Si può trattare del verificarsi di un errore di semantica dinamica (una divisione per zero, un overflow ecc.), o del verificarsi di una situazione nella quale il programmatore decide esplicitamente di terminare la computazione corrente e trasferire il controllo in un altro punto del programma, spesso al di fuori del blocco correntemente in esecuzione.

I linguaggi delle prime generazioni non fornivano alcuno strumento per gestire tali situazioni, che venivano spesso trattate mediante l'uso dei salti (goto).



**Figura 9.19** Record di attivazione nell'esempio di Figura 9.18.

Invece, molti linguaggi moderni quali C++, Ada e Java, hanno dei meccanismi *strutturati* di gestione delle eccezioni, che si configurano come veri e propri costrutti di astrazione. Tali costrutti permettono di interrompere una computazione e spostare il controllo fuori dal costrutto, o dal blocco, o dalla procedura corrente, risalendo la pila delle attivazioni dei blocchi, alla ricerca di un *gestore* per l'eccezione. Spesso, tali meccanismi consentono anche di passare dei dati attraverso il salto, risultando in uno strumento assai flessibile (e spesso anche efficiente) per la gestione di quei casi di "terminazione eccezionale" di una computazione che mal si trattano con i normali costrutti di controllo strutturato (cicli e condizionati). Pensate per gestire casi inusuali o eccezionali che si possono presentare in un programma, le eccezioni sono utili, come vedremo, anche per descrivere in modo compatto ed efficiente alcuni algoritmi "ordinari".

I meccanismi di gestione delle situazioni eccezionali variano moltissimo da linguaggio a linguaggio. Ci limiteremo qui a descrivere alcune linee comuni, senza pretesa di essere esaustivi. In generale, possiamo dire che, per gestire correttamente le eccezioni, un linguaggio deve:

1. specificare quali eccezioni sono gestibili e come possono essere definite;
2. specificare come un'eccezione può essere *solllevata*, cioè con quali meccanismi si provoca la terminazione eccezionale di una computazione;
3. specificare come un'eccezione possa essere *gestita*, cioè quali sono le azioni da compiere al verificarsi di un'eccezione e dove deve riprendere l'esecuzione del programma.

Relativamente al primo punto, in linea di massima troviamo eccezioni sollevate direttamente dalla macchina astratta (al momento in cui qualche vincolo di semantica dinamica venga violato) ed eccezioni definite dall'utente. Queste ultime possono essere valori di un tipo speciale (come nel caso di Ada, o di ML), o valori qualsiasi (come in C++), o una qualche via di mezzo (in Java, un'eccezione è un'istanza di una qualsiasi sottoclasse di Throwable). Quando un'eccezione è

```

class EmptyExcp extends Throwable {int x=0;};

int average(int[] V) throws EmptyExcp(){
    if (length(V)==0) throw new EmptyExcp();
    else {int s=0; for (int i=0, i<length(V), i++) s=s+V[i];}
    return s/length(V);
};

...
try{...}
    average(W);
    ...
}
catch (EmptyExcp e) {write('Array_vuoto'); }

```

**Figura 9.20** Gestore di eccezioni.

un tipo qualsiasi, in genere essa può contenere un valore generato dinamicamente, che viene passato al gestore.

Una volta definita un'eccezione, essa può venire sollevata implicitamente se si tratta di un'eccezione della macchina astratta, oppure esplicitamente dal programmatore, con un apposito costrutto.

Infine, circa il punto (3), la gestione di un'eccezione si sostanzia in genere in due diversi costrutti:

- un meccanismo per definire una capsula attorno ad una porzione di codice (il *blocco protetto*), con lo scopo di intercettare le eccezioni che si dovessero verificare all'interno della capsula stessa;
- la definizione di un gestore dell'eccezione, in genere legato staticamente ad un blocco protetto, al quale trasferire il controllo quando la capsula intercetta un'eccezione.

Prendiamo in esame l'esempio della Figura 9.20, scritto nel solito pseudolinguaggio ispirato a Java (e C++). La prima linea è la definizione dell'eccezione: nel caso in esame possono essere eccezioni tutte le istanze della classe `EmptyExcp`. Con qualche approssimazione possiamo immaginare un'istanza di tale classe come un record con un solo campo intero, etichettato con `x`. Passare l'eccezione vorrà dire creare un valore siffatto e specificarlo come parametro insieme al sollevamento dell'eccezione.

La seconda linea è la definizione della funzione `average`: la parola chiave `throws` introduce la lista delle eccezioni che *possono* essere sollevate nel corpo della funzione (nel nostro caso `EmptyExcp`). Tale clausola (obbligatoria in Java; opzionale in C++) ha un'importante funzione di documentazione: segnala ai clienti della funzione che essa, oltre al risultato intero, potrebbe risultare in una terminazione anomala, segnalata appunto dall'eccezione. Il costrutto per sollevare un'eccezione è `throw`; un blocco protetto è introdotto dalla parola chiave

`try`, mentre il gestore è contenuto in un blocco introdotto da `catch`. La funzione `average` calcola la media aritmetica degli elementi del vettore `V`; nel caso in cui il vettore sia vuoto, la funzione, invece di restituire alcunché, solleva un'eccezione di classe `EmptyExcp`. Al verificarsi di tale evento, la macchina astratta del linguaggio interrompe l'esecuzione del comando corrente (in questo caso il comando condizionale) e propaga l'eccezione. Vengono terminati tutti i blocchi aperti durante l'esecuzione, fino a quando non si incontra una capsula `try` che intrappoli (`catch`) questa eccezione. Nel caso della Figura 9.20, verrebbe terminata la funzione `average` e ogni blocco compreso tra l'unico `try` presente e la chiamata ad `average`. Quando l'eccezione viene intercettata da un `try`, il controllo viene passato al codice del blocco `catch`. Se non si incontra alcun `try` esplicito che intrappoli l'eccezione, questa viene catturata da un gestore default, che si preoccupa di terminare l'esecuzione del programma con qualche messaggio di errore.

Il gestore (il codice nel blocco `catch`) è legato staticamente al blocco protetto; al momento in cui si verifica un'eccezione, l'esecuzione del gestore rimpiazza la parte di blocco protetto che doveva essere ancora eseguita. In particolare, dopo l'esecuzione del gestore il controllo fluisce normalmente alla prima istruzione che segue il gestore stesso<sup>9</sup>.

Per quanto queste questioni siano fortemente dipendenti dal linguaggio, osserviamo due aspetti importanti:

1. un'eccezione non è un evento anonimo; ha un nome (spesso, anzi, come nel nostro caso, è un valore di un tipo del linguaggio) che viene esplicitamente menzionato nel costrutto `throw` e che viene usato dai costrutti di tipo `try-catch` per intrappolare una specifica classe di eccezioni.
2. sebbene l'esempio non lo mostri, l'eccezione potrebbe inglobare un valore, che il costrutto che solleva l'eccezione passa in tal modo al gestore come argomento su cui operare per "reagire" all'avvenuta eccezione (nel nostro caso: al momento del sollevamento, si potrebbe modificare il valore del campo `x` dell'eccezione).

È facile convincersi che la propagazione di un'eccezione non è un semplice salto. Supponiamo che l'eccezione si verifichi all'interno di qualche procedura: se l'eccezione non è gestita all'interno della procedura in esecuzione, occorre terminare la procedura e risollevarne l'eccezione nel punto in cui la suddetta procedura era stata invocata. Se l'eccezione non è gestita neppure dal chiamante, l'eccezione è propagata lungo la catena delle chiamate di procedura, fino a quando si incontra un gestore compatibile o si raggiunge il livello più alto, che fornirà un gestore default (terminazione con errore).

<sup>9</sup>Questa modalità di funzionamento viene detta in letteratura "gestione con *terminazione*" (perché il costrutto dove l'eccezione si è verificata viene terminato). Alcuni linguaggi più antichi (PL/I, per esempio, uno dei primi linguaggi ad introdurre meccanismi per la gestione delle eccezioni) seguono uno schema diverso, detto di "gestione con *ripresa (resumption)*"; in tal caso il gestore può prevedere che il controllo ritorni al punto dove l'eccezione si è verificata. Lo schema con *resumption* può favorire l'insorgere di errori assai difficili da localizzare.

```

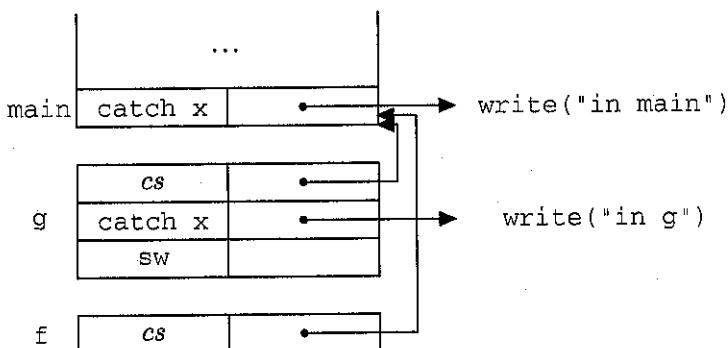
{
void f() throws X {
    throw new X();
}

void g (int sw) throws X {
    if (sw == 0) {f();}
    try {f();} catch (X e) {write("in_g");}
}

try {g(1);}
catch (X e) {write("in_main");}
}

```

**Figura 9.21** Le eccezioni si propagano lungo la catena dinamica.



**Figura 9.22** Pila di sistema per la Figura 9.21.

Un aspetto delicato che val la pena osservare esplicitamente è che le eccezioni si propagano lungo la catena dinamica, anche se il gestore è associato staticamente al blocco protetto. Per illustrare questo punto, consideriamo il codice della Figura 9.21, dove abbiamo supposto di aver già dichiarato una classe di eccezioni X.

L'eccezione X viene sollevata all'interno di due blocchi protetti: il try esterno e quello nel corpo di g. È quest'ultimo a intrappolare l'eccezione: il programma stampa la stringa "in g". La Figura 9.22 riporta la pila dei record di attivazione e dei gestori al momento in cui si esegue il corpo della funzione f. Sempre con riferimento alla Figura 9.21, si osservi che, nel caso in cui l'argomento di g nel blocco protetto sia una variabile il cui valore dipende dall'esecuzione, non è staticamente determinabile quale sia il gestore che verrà invocato.

```

type Nodo = {int chiave;
             Nodo FS;
             Nodo FD;
             }

int mul (Nodo alb) {
    if (alb == null) {return 1;}
    return alb.chiave * mul(alb.FS) * mul(alb.FD);
}

```

**Figura 9.23** Visita anticipata di un albero binario.

Riassumendo: un'eccezione viene gestita dall'ultimo gestore che sia stato posto sulla pila d'esecuzione. Si tratta di un comportamento ragionevole: l'eccezione viene intrappolata il più possibile vicino a dove si è verificata.

**Pragmatica** Abbiamo discusso l'uso delle eccezioni per gestire situazioni d'errore. Vi sono casi "ordinari", tuttavia, in cui un uso oculato delle eccezioni permette programmi più eleganti ed efficienti. Ci limiteremo a presentare un esempio, quello della visita di un albero binario, di cui si vuole calcolare il prodotto degli interi che etichettano i nodi. Il codice più ovvio è riportato in Figura 9.23, dove si è usata una visita anticipata dell'albero e si è supposto che il tipo Alb sia una struttura (un record, o una classe) con tre campi, di cui il primo intero e gli altri a loro volta di tipo Alb, per collegare la struttura; null è il valore nullo generico.

La funzione mul restituisce correttamente il prodotto dei nodi, ma è inefficiente nel caso di alberi molto grandi e con probabilità significativa che qualche nodo sia zero, dato che in tal caso la funzione potrebbe immediatamente terminare con risultato zero. Possiamo sfruttare il meccanismo delle eccezioni per forzare questa terminazione anticipata, senza per questo produrre codice scadente.

La Figura 9.24 mostra la funzione mulAus che solleva un'eccezione (di classe Zero, che supponiamo definita altrove) quando incontra un nodo etichettato con zero; mulAus è chiamata da mulEff<sup>10</sup> all'interno di un blocco protetto che gestisce l'eccezione restituendo zero.

### 9.3.1 Implementare le eccezioni

Il modo più semplice e intuitivo col quale una macchina astratta può implementare le eccezioni è quello di sfruttare la pila dei record di attivazione. Ogni volta che, durante l'esecuzione, si entra in un blocco protetto, nel record di attivazione

<sup>10</sup>La funzione mulEff è la funzione "esportata" verso i clienti di questo programma; in gergo si dice che mulEff è il wrapper (l'incarto) di mulAus.

```

int mulAus (Nodo alb) throws Zero{
    if (alb == null) {return 1;}
    if (alb.chiave == 0) {throw new Zero();}
    return alb.chiave * mulAus(alb.FS) * mulAus(alb.FD);
}

int mulEff (Nodo alb){
    try {return mulAus (alb);}
    catch (Zero e) {return 0;}
}

```

Figura 9.24 Una visita più efficiente.

```

class X extends Exception {};
class P{
    void f() throws X{
        throw new X();
    }
}

class Q{
    class X extends Exception {};
    void g(){
        P p = new P();
        try {p.f();} catch (X e){
            System.out.println("in_g");
        }
    }
}

```

Figura 9.25 Scope statico dei nomi delle eccezioni.

(della procedura corrente, o del blocco anonimo) viene inserito un puntatore al gestore corrispondente (insieme, ovviamente, al tipo delle eccezioni cui si riferisce). Quando si esce “normalmente” da un blocco protetto (cioè perché il controllo ne esce in modo naturale, non per effetto di un’eccezione), viene tolto dalla pila il riferimento al gestore. Infine, quando viene sollevata un’eccezione, la macchina astratta cerca un gestore per quell’eccezione nel record di attivazione corrente. Se non lo trova, usa le informazioni del record di attivazione per ripristinare lo stato della macchina, toglie il record dalla pila e risolleva l’eccezione. Si tratta di una soluzione concettualmente molto semplice, ma con un difetto non trascurabile: ogni volta che si entra in un blocco protetto, o se ne esce, la macchina astratta deve manipolare la pila. Questa implementazione, dunque, richiede lavoro esplicito

### Eccezioni e scope statico

Linguaggi quali Java e C++ combinano lo scope statico delle definizioni dei nomi (e quindi anche dei nomi delle eccezioni), con il legame dinamico dei gestori al blocco protetto, come abbiamo appena discusso. Tale combinazione è talvolta causa di incomprensioni, come il codice Java della Figura 9.25 cerca di mostrare. Ad una lettura superficiale il codice sembra sintatticamente corretto; inoltre, si direbbe che un’invocazione di `g` risulti nella stampa della stringa “in `g`”.

Se si tenta la compilazione del codice, invece, il compilatore rileva due errori di semantica statica relativi alla linea 12: (i) l’eccezione `X` che dovrebbe essere catturata dal `catch` corrispondente, non è mai sollevata nel `try`; (ii) l’eccezione `X`, sollevata da `f`, non è dichiarata nella clausola `throws` (che manca del tutto) di `g`.

Il fatto è che i nomi delle eccezioni (`X` in questo caso) hanno normale scope statico. Il metodo `f` solleva l’eccezione `X` dichiarata alla linea 1; mentre il `catch` di linea 12 intrappa l’eccezione `X` dichiarata nella linea 9 (e che più correttamente si indica con `Q.X`, essendo una classe annidata in `Q`). Proprio per evitare errori causati da situazioni del genere, Java impone ad ogni metodo di dichiarare nel suo `throws` tutte le eccezioni che si possono generare nel suo corpo.

Si può riprodurre una situazione analoga in C++, *mutatis mutandis*. In C++, tuttavia, la clausola `throws` è opzionale. Se compiliamo il corrispettivo C++ del codice della Figura 9.25, dove siano omesse le clausole `throws`, la compilazione va a buon fine. Ma ovviamente un’invocazione del metodo `f` solleva un’eccezione diversa da quella intrappolata nel corpo di `g`: un’invocazione di `g` risulta in un’eccezione `X` (della classe dichiarata alla linea 1) che viene propagata all’insù.

anche nel caso normale (e più frequente) in cui l’eccezione *non* si verifichi.

Una soluzione più efficiente a tempo d’esecuzione la si ottiene anticipando un po’ di lavoro a tempo di compilazione. In primo luogo, ad ogni procedura viene associato un blocco protetto nascosto costituito da tutto il corpo della procedura e il cui gestore nascosto è responsabile solo del ripristino dello stato e del risollevamento dell’eccezione immutata. Il compilatore prepara poi una tabella *EH* in cui, per ogni blocco protetto (inclusi quelli nascosti), inserisce due indirizzi (*ip*, *ig*) che corrispondono all’inizio del blocco protetto e all’inizio del corrispondente gestore; supponiamo che l’inizio del gestore coincida con la fine del blocco protetto. La tabella, ordinata sulla prima componente degli indirizzi, è passata alla macchina astratta. Quando si entra o si esce normalmente da un blocco protetto non si deve fare niente. Quando invece viene sollevata un’eccezione, si cerca nella tabella una coppia di indirizzi (*ip*, *ig*) tali che *ip* è il massimo indirizzo presente in *EH* per cui  $ip \leq pc \leq ig$ , dove *pc* è il valore del contatore di programma al momento in cui si è verificata l’eccezione: essendo *EH* ordinata, si può effettuare una ricerca binaria. La ricerca permette di determinare un gestore dell’eccezione (si ricordi che si è aggiunto un gestore nascosto per ogni procedura). Se tale gestore ri-solleva l’eccezione (perché non cattura “quella” eccezione, oppure perché è un gestore nascosto), il procedimento riprende, col valore attuale del contatore

di programma (se si trattava del gestore di un'altra eccezione) oppure con l'indirizzo di ritorno della procedura (se si trattava di un gestore nascosto). Questa soluzione è più costosa della precedente nel caso in cui si verifichi un'eccezione, almeno di un fattore logaritmico nel numero di gestori (e procedure) presenti nel programma (l'aggravio dipende dalla necessità di fare una ricerca in tabella ogni volta; il costo è logaritmico perché si utilizza una ricerca binaria). D'altra parte, non costa nulla all'ingresso e all'uscita da un blocco protetto, dove la soluzione precedente richiedeva un costo costante per ciascuna di queste operazioni. Siccome è ragionevole supporre che il verificarsi di un'eccezione sia un evento più raro del semplice ingresso in un blocco protetto, si tratta di una soluzione mediamente più efficiente della precedente.

## 9.4 Sommario del capitolo

Il capitolo ha trattato una questione centrale di ogni linguaggio di programmazione, quella dei meccanismi per l'astrazione funzionale. Ha discusso in particolar modo due dei meccanismi linguistici più importanti che permettono l'astrazione funzionale in un linguaggio di alto livello: le procedure e i costrutti per la gestione delle eccezioni. I concetti principali introdotti sono:

- *Procedure*: il concetto di procedura, o funzione, o sottoprogramma, costituisce l'unità fondamentale di modularizzazione di un programma. La comunicazione tra procedure è assicurata mediante valori di ritorno, parametri e l'ambiente non locale.
- *Modalità di passaggio dei parametri*: da un punto di vista semantico, vi sono parametri di ingresso, di uscita e di ingresso-uscita; da un punto di vista implementativo vi sono molti modi per passare un parametro:
  - per valore;
  - per riferimento;
  - secondo uno dei modi che sono varianti del passaggio per valore: per risultato, per costante, per valore-risultato;
  - per nome.
- *Funzioni di ordine superiore*: funzioni che prendono come parametro, o restituiscono come risultato, altre funzioni. Il secondo caso, in particolare, ha un impatto notevole sull'implementazione di un linguaggio, forzando l'abbandono della disciplina a pila per i record di attivazione.
- *Politiche di binding*: quando siamo in presenza di funzioni passate come argomento, la politica di binding specifica il momento in cui l'ambiente di valutazione è determinato: se al momento del legame tra la procedura e il parametro (deep binding), o se al momento dell'uso della procedura mediante il parametro (shallow binding).
- *Chiusure*: strutture dati costituite da una porzione di codice e un ambiente di valutazione; costituiscono la modalità canonica di implementazione del passaggio per nome e di tutte quelle situazioni in cui una funzione debba essere passata per parametro o restituita come risultato.

- *Eccezioni*: condizioni eccezionali che si possono verificare e che sono trattate nei linguaggi di alto livello mediante un *blocco protetto* ed un *gestore*; al verificarsi di un'eccezione, il relativo gestore è determinato risalendo la catena dinamica.

## 9.5 Nota bibliografica

Tutte le principali modalità di passaggio dei parametri originano dai lavori del comitato ALGOL, che si concretizzarono poi anche in altri linguaggi, quali ALGOL-W e Pascal. La definizione originale di ALGOL 60 [12] è una pietra miliare del progetto dei linguaggi di programmazione. Su ALGOL-W, che includeva il passaggio per nome (default), valore, risultato e valore-risultato, nonché puntatori e garbage collection, si può vedere il lavoro preparatorio [111] o il manuale di riferimento [92]. Su Pascal, che adotta per default il passaggio per riferimento, si veda la prima definizione [107] o il manuale di riferimento [48] della versione ISO Standard.

Le questioni connesse alla determinazione dell'ambiente nel caso di funzioni di ordine superiore sono spesso indicate col termine di *funarg problem* (*functional argument*): il *downward* funarg problem si riferisce al caso delle funzioni passate come argomento, e quindi alla necessità di trattare il deep binding; lo *upward* funarg problem si riferisce al caso in cui una funzione possa essere restituita come risultato [73]. Le relazioni tra politiche di binding e regole di scope sono discusse in [102].

Uno dei primi linguaggi con eccezioni è PL/I, con una gestione con ripresa, si veda [62]. La più moderna gestione con terminazione (che prevede il legame statico tra blocco protetto e gestore) discende da Ada, la quale a sua volta è ispirata al lavoro [40].

## 9.6 Esercizi

1. A pag. 244, commentando la Figura 9.1, si osserva che l'ambiente della funzione `foo` comprende (come non locale) il nome `foo`. A cosa serve la presenza di tale nome all'interno del blocco?
2. Si dica cosa viene stampato dal seguente frammento di codice scritto in uno pseudolinguaggio che ammette parametri per riferimento (sia `Y` che `J` sono passati per riferimento).

```
int X[10];
int i = 1;
X[0] = 0;
X[1] = 0;
X[2] = 0;
void foo (reference int Y,J) {
    X[J] = J+1;
    write(Y);
    J++;
}
```

```

    X[J]=J;
    write(Y);
}
foo(X[i],i);
write(X[i]);

```

3. Si dica cosa viene stampato dal seguente frammento di codice scritto in uno pseudo-linguaggio che ammette parametri per *valore-risultato*.

```

int X = 2;
void foo (valueresult int Y){
    Y++;
    write(X);
    Y++;
}
foo(X);
write(X);

```

4. Il seguente frammento di codice viene fornito ad un *interprete* di uno pseudo-linguaggio che ammette parametri per costante:

```

int X = 2;
void foo (constant int Y){
    write(Y);
    Y=Y+1;
}
foo(X);
write(X);

```

Si dica qual è il comportamento più probabile dell'interprete.

5. Si dica cosa viene stampato dal seguente frammento di codice scritto in uno pseudo-linguaggio che ammette parametri per *nome*.

```

int X = 2;
void foo (name int Y){
    X++;
    write(Y);
    X++;
}
foo(X+1);
write(X);

```

6. Relativamente alla discussione dell'implementazione del deep binding tramite chiusure, si descriva in dettaglio il caso di un linguaggio con scope dinamico.

7. Si consideri il seguente frammento in un linguaggio con eccezioni e passaggio per valore-risultato e per riferimento:

```

int y=0;
void f(int x){
    x = x+1;
    throw E;
    x = x+1;
}
try{ f(y); } catch E {};
write(y);
}

```

Si dica cosa viene stampato dal programma qualora il passaggio dei parametri avvenga: (i) per valore-risultato; (ii) per riferimento.

8. In uno pseudolinguaggio con eccezioni si incontra il seguente blocco di codice:

```

void ecc() throws X {
    throw new X();
}
void g (int para) throws X {
    if (para == 0) {ecc();}
    try {ecc();} catch (X) {write(3);}
}
void main () {
    try {g(1);} catch (X) {write(1);}
    try {g(0);} catch (X) {write(0);}
}

```

Si dica cosa viene stampato all'esecuzione di *main()*.

9. In uno pseudolinguaggio con eccezioni viene data la seguente definizione di funzione:

```

int f(int x){
    if (x==0) return 1;
    else if (x==1) throw E;
    else if (x==2) return f(1);
    else try {return f(x-1);} catch E {return x+1;}
}

```

Quale valore viene ritornato da *f(4)*?

10. La descrizione dell'implementazione delle eccezioni del Paragrafo 9.3.1 assume che il compilatore abbia accesso (direttamente o tramite la fase di linking) a tutto il codice del programma. Suggerire una modifica allo schema implementativo basato sulla tabella dei gestori nel caso di un linguaggio in cui sia permessa la compilazione separata di frammenti di programma (un esempio è Java, che permette la compilazione separata delle classi).

Ogni linguaggio di programmazione offre costrutti e meccanismi per la strutturazione dei dati. Invece delle sole semplici sequenze di bit della macchina fisica, un linguaggio di alto livello fornisce dati complessi e strutturati, che più facilmente si prestano a descrivere la struttura dei problemi che devono essere risolti. Questi costrutti e meccanismi sono costituiti da quello che si chiama il *sistema di tipi* di un linguaggio: lungi dall'essere un aspetto accessorio, i tipi rappresentano una delle caratteristiche salienti di un linguaggio di programmazione, che differenziano in modo sostanziale un linguaggio da un altro.

Prenderemo in esame in questo capitolo i sistemi di tipi in senso generale, discutendo i tipi primitivi e i meccanismi che vengono offerti per definirne di nuovi. Centrale alla nostra presentazione sarà la nozione di *sicurezza rispetto ai tipi*, che introdurremo nel Paragrafo 10.2. Affronteremo, quindi, la questione dell'equivalenza e della compatibilità tra tipi, cioè dei meccanismi che permettono di usare un valore di un certo tipo in un contesto che richiederebbe un tipo diverso, per poi discutere del polimorfismo e dell'overloading. Concluderemo il capitolo con alcune questioni relative alla gestione della memoria (*garbage collection*), che non sono un argomento propriamente riguardante i tipi di dato, ma che ben complementa la trattazione dei puntatori che invece dovremo fare.

Rimandiamo al prossimo capitolo una discussione più approfondita dell'astrazione sui dati, cioè di quei meccanismi per la definizione di tipi che permettono di nascondere alcune proprietà dei dati.

## 10.1 Tipi di dato

I tipi di dato sono presenti nei linguaggi di programmazione per almeno tre scopi diversi:

- a livello di progetto, come supporto all'organizzazione concettuale;
- a livello di programma, come supporto alla correttezza;
- a livello di traduzione, come supporto all'implementazione.

Prima di discutere in dettaglio questi aspetti nei prossimi paragrafi, diamo una definizione che, come spesso accade nel contesto dei linguaggi di program-

mazione, non è formalmente precisa, ma è sufficiente a spiegare i fenomeni che intendiamo studiare.

**Definizione 10.1 (Tipo di dato)** *Un tipo di dato è una collezione di valori omogenei ed effettivamente presentati, dotata di un insieme di operazioni che manipolano tali valori.*

Vediamo di chiarire i vari termini usati nella precedente definizione. Un tipo è una collezione di valori, come i numeri interi, o un intervallo di interi. L'aggettivo “omogenei”, per quanto informale, suggerisce che tali valori devono condividere alcune proprietà strutturali, che li rendono tutti simili tra loro. Ad esempio, siamo disposti a considerare un tipo i numeri interi compresi tra 5 e 10, mentre *non* chiameremo un tipo di dato l’insieme composto dal numero intero 2, il valore di verità `true` e il numero razionale 3.4. In secondo luogo, tali valori “vengono assieme” alle operazioni che li manipolano. Ad esempio, assieme ai numeri interi possiamo considerare le usuali operazioni di somma, sottrazione, moltiplicazione e divisione; oppure considerare *anche* operazioni meno usuali quali il resto della divisione intera e l’elevamento a potenza. In base alla nostra definizione, lo stesso insieme di valori, dotato di due insiemi distinti di operazioni, dà luogo a tipi di dato diversi. L’ultima componente della definizione è quell’“effettivamente presentati” riferito ai valori. Siccome stiamo parlando di linguaggi per descrivere algoritmi, siamo interessati a valori che sia possibile presentare (scrivere, nominare) in modo finito. I numeri reali (quelli “veri” della matematica, cioè l’unico campo ordinato archimedico completo) *non* sono effettivamente presentabili, perché vi sono numeri reali con espansione decimale infinita che non è possibile ottenere per mezzo di alcun algoritmo. Le loro approssimazioni nei linguaggi di programmazione (`real` o `float`) non sono nient’altro che un sottinsieme dei razionali.

### 10.1.1 Tipi come supporto all’organizzazione concettuale

La soluzione di ogni problema complesso ha una struttura concettuale che spesso rispecchia quella del problema. La presenza di tipi diversi permette al progettista di utilizzare quello più adatto per ogni classe di concetti. Ad esempio, un programma per la gestione di prenotazioni alberghiere, gestirà concetti quali clienti, date, prezzi, camere ecc. Ciascuno di questi concetti può essere descritto come un tipo diverso, con il suo insieme di operazioni. Il progettista può definire nuovi tipi e associarli a concetti diversi, anche se rappresentati con gli stessi valori. Ad esempio, camere e prezzi potrebbero essere entrambi numeri interi in un certo intervallo, ma la loro rappresentazione con tipi distinti rende esplicita la loro differenza concettuale.

L’uso di tipi distinti si presenta sia come strumento di progetto che come strumento di documentazione: alla lettura del programma è chiaro dalla dichiarazione del tipo che una variabile di tipo “camera” ha un ruolo diverso da quello di una variabile di tipo “prezzo”. In questo senso i tipi svolgono un ruolo analogo a quello dei commenti, con l’importante differenza che si tratta di commenti *effettivamente controllabili*, come vedremo nel prossimo paragrafo.

### 10.1.2 Tipi per la correttezza

Ogni linguaggio ha proprie regole di controllo dei tipi, che disciplinano come i tipi possono (o devono) essere usati in un programma. L’esempio più comune è quello di un assegnamento: affinché un comando della forma `x = exp;` sia corretto, la maggioranza dei linguaggi richiede che il tipo di `exp` coincida (o, meglio, sia compatibile) con il tipo di `x`. Analogamente, è vietato sommare interi e record, o chiamare (cioè trasferire il controllo a) un oggetto che non sia una funzione o una procedura.

Tali vincoli sono presenti nei linguaggi sia per evitare errori hardware a runtime (per esempio una chiamata ad un oggetto che non sia una funzione può generare un errore di indirizzamento), sia, soprattutto, per evitare gli errori logici che molto spesso si celano dietro la violazione di una regola di tipo. La somma di un intero e di una stringa raramente corrisponde a qualcosa di sensato.

I linguaggi di programmazione, in conclusione, assumono che la violazione di un vincolo di tipo corrisponda ad un possibile errore semantico, di progetto. Il punto cruciale è che molti linguaggi verificano che i vincoli di tipo siano tutti soddisfatti prima di procedere all’esecuzione (o alla generazione del codice) di un programma. È questo il ruolo del controllore dei tipi (*type checker*), una componente molto importante della fase di controllo della semantica statica in un compilatore (Paragrafo 2.3). Abbiamo parlato dei tipi come di commenti effettivamente controllabili perché con essi il programmatore comunica i modi legali con i quali determinati oggetti potranno essere utilizzati; ma (a differenza dei commenti) il compilatore (o la macchina astratta del linguaggio) rileva e segnala ogni tentativo di uso improprio di quegli oggetti.

È evidente che un programma che rispetta tutte le regole di tipo può essere comunque logicamente scorretto: i tipi assicurano una correttezza minimale, che tuttavia è di grande aiuto durante la fase di sviluppo di un programma. Un’analogia assai efficace è quella col controllo dimensionale in fisica: quando il fisico scrive una formula, prima ancora di verificare la sua correttezza ragionando a partire dai principi, egli verifica che le dimensioni siano corrette. Se la formula deve esprimere una velocità, deve essere uno spazio fratto un tempo; se un’accelerazione, uno spazio fratto un tempo al quadrato; e così via. Se la formula è dimensionalmente errata, non c’è da perder tempo su di essa: è certamente scorretta. Se invece è dimensionalmente corretta, allora *può* essere scorretta e deve essere controllata semanticamente.

Talvolta, tuttavia, le regole di tipo possono apparire troppo restrittive. Un programmatore C è abituato a manipolare liberamente i puntatori (cioè, in C, veri e propri indirizzi di memoria) e può ritenere inutilmente restrittivo il divieto di eseguire qualsiasi forma di aritmetica dei puntatori che trova in linguaggi quali Pascal o Java. In questo caso la risposta dei progettisti di questi ultimi linguaggi è che i benefici di un controllo stringente dei tipi sopravanzano di gran lunga la perdita di espressività e di stringatezza.

Un esempio più calzante è quello di un sottoprogramma per ordinare un vettore. In molti linguaggi, a causa della presenza dei tipi, occorrerà scrivere una funzione per ordinare un vettore di interi, un’altra per ordinare un vettore di ca-

ratteri, un'altra ancora per i reali, e così via. Tutte queste funzioni sono identiche nell'algoritmo e differiscono solo nella dichiarazione dei tipi dei parametri e delle variabili. La via di uscita, in questo caso, è l'adozione di regole di tipi più sofisticate, che, senza rinunciare ai controlli, permettano di scrivere una sola funzione, parametrica nel tipo. Vedremo nel seguito che linguaggi che permettono questa forma di *polimorfismo* sono sempre più diffusi.

Osserviamo, infine, che le regole che riguardano i tipi non sono sempre sufficienti a garantire che i vincoli che esse esprimono siano soddisfatti. Facciamo qui solo un semplice esempio. In un linguaggio che permette deallocazione esplicita di memoria sullo heap è possibile che si generino dei riferimenti (puntatori) che si riferiscono a memoria che non è più allocata al programma (*dangling references*). Un tentativo di accedere a tali riferimenti è un errore che può essere classificato come errore di tipo, ma non è detto che venga rilevato e segnalato dalla macchina astratta. Classificheremo pertanto i linguaggi di programmazione tra *sicuri* e *insicuri* rispetto ai tipi, proprio in base alla possibilità che vi possano essere durante l'esecuzione delle violazioni dei vincoli di tipo non rilevate dalla macchina astratta.

### 10.1.3 Tipi e implementazione

La terza motivazione principale per la presenza di tipi nei linguaggi di programmazione è che essi costituiscono importanti informazioni per la macchina astratta. La prima informazione, evidente, riguarda la dimensione della memoria da allocare per i vari oggetti. Il compilatore può allocare una parola per un intero, un byte per un valore booleano,  $n$  parole per un vettore di  $n$  interi ecc.: tutte queste informazioni, in presenza di tipi, sono disponibili staticamente, e non cambiano durante l'esecuzione.

Come conseguenza di tale allocazione statica, durante la compilazione è possibile ottimizzare le operazioni di accesso ad un oggetto. Abbiamo già discusso nel Paragrafo 7.3.2, come l'accesso ad una variabile allocata nel record di attivazione di una procedura avviene per offset rispetto al puntatore al RdA, senza bisogno di una ricerca per nome a tempo d'esecuzione. Questa forma di ottimizzazione è possibile perché le informazioni veicolate tramite i tipi permettono di determinare staticamente la dimensione di allocazione di quasi tutti gli oggetti presenti in un programma. Si osservi che tali ottimizzazioni sono possibili anche per oggetti allocati sullo heap. Vedremo tra poco che un record è costituito da una collezione di campi, ciascuno caratterizzato da un proprio nome e un proprio tipo. Ad esempio, prendendo la notazione in prestito da C, la seguente dichiarazione introduce il tipo `Professore`, un record con due campi:

```
struct Professore{
    char Nome[20];
    int Codice_corso;
}
```

Se adesso abbiamo una variabile `p` di tipo `Professore`, possiamo accedere ai suoi campi selezionandone il nome: `p.Nome` oppure `p.Codice_corso`. Dovunque

que sia allocato l'oggetto `p` (stack o heap), l'accesso ai suoi campi sarà sempre possibile mediante offset relativamente all'indirizzo iniziale di memorizzazione di `p`.

## 10.2 Sistemi di tipi

Prima di entrare nel vivo della trattazione, introdurremo in questo paragrafo un po' di terminologia, che illustreremo ampiamente nelle sezioni che seguono.

Per quanto argomentato nel paragrafo precedente, ogni linguaggio di programmazione ha un proprio *sistema di tipi*, ossia il complesso delle informazioni e delle regole che governano i tipi in quel linguaggio. Più precisamente, un sistema di tipi è costituito da<sup>1</sup>:

1. l'insieme dei tipi predefiniti dal linguaggio;
2. i meccanismi che permettono di definire nuovi tipi;
3. i meccanismi relativi al controllo dei tipi, tra i quali distingueremo:
  - le regole di equivalenza, che specificano quando due tipi formalmente diversi corrispondono allo stesso tipo;
  - le regole di compatibilità, che specificano quando un valore di un certo tipo può essere utilizzato in un contesto nel quale sarebbe richiesto un tipo diverso;
  - le regole e le tecniche di inferenza dei tipi, che specificano come il linguaggio attribuisce un tipo ad un'espressione complessa, partendo dalle informazioni delle sue componenti;
4. la specifica se i vincoli (o quali vincoli) siano da controllare staticamente o dinamicamente.

Un sistema di tipi (e, per estensione, un linguaggio) è *sicuro relativamente ai tipi* (o *type safe*)<sup>2</sup> quando nessun programma può violare le distinzioni tra tipi definite in quel linguaggio. Detto in altri termini, un sistema di tipi è sicuro quando nessun programma durante l'esecuzione può generare un errore non segnalato che deriva da una violazione di tipo. Ancora una volta non è sempre chiaro cosa sia una violazione di tipo, almeno in generale. Abbiamo già fatto alcuni esempi, quali l'accesso a memoria non allocata per il programma, o la chiamata di un valore che non sia una funzione; vedremo nel seguito del capitolo altri esempi di errori del genere.

Abbiamo definito un tipo come una coppia costituita da un insieme di valori ed un insieme di operazioni. In uno specifico linguaggio, i valori di un tipo possono corrispondere a diverse entità sintattiche (costanti, espressioni ecc.). Fissato

<sup>1</sup>“Un sistema di tipi è un metodo sintattico ragionevolmente efficiente per dimostrare l'assenza di certi comportamenti in un programma mediante la classificazione delle frasi [del programma stesso] a seconda delle categorie di valori che queste calcolano” [79].

<sup>2</sup>Molta letteratura utilizza il termine *fortemente tipizzato* (*strongly typed*) al posto di sicuro relativamente ai tipi.

un linguaggio di programmazione, possiamo classificare i suoi tipi a seconda di come i suoi valori possono venir manipolati e del genere di entità sintattiche che corrispondono a tali valori. Seguendo la classificazione che abbiamo già visto nel riquadro di pag. 214, abbiamo valori:

- *denotabili*, se possono essere associati ad un nome;
- *esprimibili*, se possono essere il risultato di un'espressione complessa (cioè diversa da un semplice nome);
- *memorizzabili*, se possono essere memorizzati in una variabile.

Facciamo qualche esempio. I valori del tipo delle funzioni da `int` a `int` sono denotabili in quasi tutti i linguaggi, perché possiamo dare loro un nome con una dichiarazione; ad esempio:

```
int succ (int x) {
    return x+1;
}
```

assegna il nome `succ` alla funzione che calcola il successore. Questi stessi valori non sono in genere esprimibili nei comuni linguaggi imperativi, perché non ci sono espressioni complesse che restituiscono una funzione come risultato della loro valutazione. Allo stesso modo, non sono in genere valori memorizzabili, perché non possiamo assegnare una funzione ad una variabile. La situazione è diversa in linguaggi di altri paradigmi, quali per esempio i linguaggi funzionali (Scheme, ML, Haskell ecc.), nei quali i valori funzionali sono sia denotabili, che esprimibili, che, in certi linguaggi, memorizzabili. I valori di tipo intero sono in genere sia denotabili (li possiamo associare ad una costante) che esprimibili, che memorizzabili.

### 10.2.1 Controlli statici e dinamici

Un linguaggio ha *tipizzazione statica* (o controllo statico dei tipi) se i controlli dei vincoli di tipizzazione possono essere condotti a tempo di compilazione, sul testo del programma; ha *tipizzazione dinamica* altrimenti (cioè se i controlli avvengono a tempo d'esecuzione).

Il controllo dinamico dei tipi prevede che ogni oggetto (valore) possieda un descrittore a run-time che ne specifica il tipo. La macchina astratta è responsabile di controllare che ogni operazione sia applicata solo ad operandi del tipo corretto. Spesso ciò viene ottenuto facendo generare al compilatore opportuno codice di controllo prima di ogni operazione. Non è difficile comprendere che la tipizzazione dinamica previene gli errori di tipo, ma è inefficiente in esecuzione, dato che le operazioni sono intercalate con i controlli di tipo. Inoltre, l'eventuale errore di tipo è rilevato solo durante l'esecuzione, quando il programma potrebbe già essere operativo presso il suo utente finale.

Nel controllo statico dei tipi, invece, i controlli sono fatti a tempo di compilazione. In tal modo i controlli sono anticipati e vengono così rilevati dal programmatore prima della consegna del programma all'utente. In un controllo completamente statico, inoltre, il mantenimento esplicito dei tipi durante l'esecuzione è del

tutto inutile, perché la correttezza è stata garantita staticamente *per ogni sequenza di esecuzione*. L'esecuzione è così più efficiente, visto che non sono necessari controlli a run-time. Vi sono, ovviamente, dei prezzi da pagare. In primo luogo la progettazione di un linguaggio con controllo statico è più complessa di quella di un linguaggio con controllo dinamico, specialmente se, insieme al controllo statico, si vuole garantire anche la sicurezza rispetto ai tipi<sup>3</sup>. In secondo luogo, la compilazione è più complessa e lenta: un prezzo che si paga volentieri, visto che la compilazione viene eseguita poche volte (rispetto al numero di volte che verrà eseguito il programma) e, soprattutto, perché i controlli di tipo abbreviano la fase di testing e di debugging.

Vi è infine un terzo prezzo da pagare, meno appariscente dei precedenti, ma intimamente collegato con la natura statica del controllo dei tipi: i controlli statici possono decretare come sbagliati programmi che, in realtà, non causano un errore di tipo durante l'esecuzione. A titolo di semplice esempio si consideri il seguente frammento nel nostro pseudolinguaggio:

```
int x;
if (0==1) x = "pippo";
else x = 3+4;
```

Il primo ramo del condizionale assegna alla variabile intera `x` un valore non compatibile con il suo tipo, ma l'esecuzione del frammento non causa nessun errore, perché la condizione non è mai verificata. Tuttavia ogni controllo statico dei tipi segnalerà tale frammento come scorretto, perché i tipi dei due rami del condizionale non sono entrambi corretti. Un controllo statico è dunque sempre più conservatore dei controlli dinamici. La motivazione di quest'affermazione va ricercata nelle considerazioni del Capitolo 5 sull'esistenza di problemi indecidibili. Oltre al problema della fermata, anche il problema di verificare se un programma provochi un errore di tipo durante l'esecuzione è indecidibile (si veda il riquadro di pag. 286). Ne segue che non esiste alcun controllore statico che possa rilevare esattamente tutti e soli i programmi che possono generare errori a tempo d'esecuzione. Il controllore statico adotta quindi una posizione prudenziale: escludere più programmi di quelli davvero necessari, in modo da poter garantire la correttezza.

Come abbiamo già visto più volte in altri contesti, il controllo statico e quello dinamico dei tipi costituiscono i due estremi di uno spettro di soluzioni in cui le due modalità coesistono. Quasi ogni linguaggio di alto livello combina controlli statici con controlli dinamici. Riservandoci di tornare ancora sull'argomento a tempo debito, facciamo soltanto un semplice esempio in Pascal, un linguaggio tradizionalmente classificato tra quelli con controllo statico dei tipi. Pascal permette la definizione di tipi intervallo (vedi Paragrafo 10.3.9): ad esempio `1..10` è il tipo degli interi compresi tra 1 e 10. Un'espressione di tipo intervallo dovrà essere controllata dinamicamente, per garantire che il suo valore sia effettivamente contenuto all'interno dell'intervallo. Più in generale, ogni linguaggio con vettori

<sup>3</sup>Non è un caso che LISP, un linguaggio con controllo dinamico, sia anche uno dei primi linguaggi di alto livello realizzati, mentre Java, che ha un esteso sistema di tipi con controllo statico ed è type safe, sia uno degli ultimi linguaggi commerciali progettati.

### Il verificarsi di un errore di tipo non è decidibile

Non è difficile dimostrare che è indecidibile il problema di verificare se un programma provochi un errore di tipo durante l'esecuzione. Possiamo infatti sfruttare quello che già sappiamo, cioè che è indecidibile il problema della fermata.

Consideriamo il seguente frammento, dove *P* è un generico programma:

```
int x;
P;
x = "pippo";
```

In quali condizioni questo frammento genera un errore di tipo durante l'esecuzione? Solo se *P* termina, nel qual caso si tenta di eseguire l'assegnamento  
*x* = "pippo";

che appunto viola il sistema di tipi. Se invece *P* non termina, il frammento non genera alcun errore, perché il controllo rimane sempre dentro *P* senza mai arrivare all'assegnamento critico. Dunque il frammento genera un errore di tipo se e solo se *P* termina. Se ora esistesse un metodo generale per decidere se un generico programma causa un errore di tipo durante l'esecuzione, potremmo applicare tale metodo al nostro frammento. Tale metodo, però, sarebbe anche un modo per decidere della terminazione del (generico) programma *P*, cosa che sappiamo essere impossibile.

che voglia controllare che l'indice di accesso ad un array sia compatibile con la dichiarazione dell'array stesso, non potrà che introdurre dei controlli dinamici.

## 10.3 Tipi scalari

Sono detti *tipi scalari* (o tipi semplici) tutti quei tipi i cui valori non sono costituiti da aggregazioni di altri valori. Presenteremo in questo paragrafo una veloce rassegna dei principali tipi scalari presenti nei più comuni linguaggi di programmazione, mentre tratteremo nel paragrafo seguente i tipi che risultano per aggregazione di altri valori. I dettagli (che non daremo) dipendono ovviamente dagli specifici linguaggi. Per fissare la notazione, supporremo di avere nel nostro pseudolinguaggio un modo per definire (dichiarare) nuovi tipi, nel modo seguente:

```
type nuovotipo = espressione;
```

che introduce il nome di tipo *nuovotipo*, con struttura data da *espressione*. Per sottolineare alcuni fenomeni tipici di un fissato linguaggio, talvolta faremo uso della sintassi specifica di quel linguaggio. In C, scriveremmo, con lo stesso significato:

```
typedef espressione nuovotipo;
```

### 10.3.1 Booleani

Il tipo dei valori logici, o booleani, è costituito da:

- *valori*: i due valori di verità, vero e falso;
- *operazioni*: un'opportuna selezione tra le principali operazioni logiche: congiunzione (and), disgiunzione (or), negazione (not), uguaglianza, or esclusivo ecc.

Laddove presente (C per esempio non possiede un tipo siffatto), i suoi valori sono memorizzabili, esprimibili e denotabili. Per motivi di indirizzabilità, la rappresentazione in memoria non consiste in un singolo bit, ma in un byte (che possono divenire di più per ragioni di allineamento).

### 10.3.2 Caratteri

Il tipo dei caratteri è costituito da:

- *valori*: un insieme di codici di caratteri, fissato al momento della definizione del linguaggio; i due insiemi più comuni sono ASCII e UNICODE;
- *operazioni*: fortemente dipendenti dal linguaggio; troviamo sempre l'uguaglianza, i confronti e qualche modo per passare da un carattere al successivo (nella fissata codifica) e/o al precedente.

I valori sono memorizzabili, esprimibili e denotabili. La rappresentazione in memoria consiste in un byte (ASCII) o due byte (UNICODE).

### 10.3.3 Interi

Il tipo dei numeri interi è costituito da:

- *valori*: un sottinsieme finito dei numeri interi, di norma fissato al momento della definizione del linguaggio (ma vi sono casi in cui è fissato dalla specifica macchina astratta, il che è causa di qualche problema di portabilità); per questioni di rappresentazione è di norma un intervallo della forma  $[-2^t, 2^t - 1]$ ;
- *operazioni*: i confronti e un'opportuna selezione tra le principali operazioni aritmetiche (somma, sottrazione, moltiplicazione, divisione intera, resto della divisione, esponenziazione ecc.);

I valori sono memorizzabili, esprimibili e denotabili. La rappresentazione in memoria consiste in un numero pari di byte (usualmente 2, o 4, o 8), in complemento a due. Alcuni linguaggi hanno un supporto built-in per interi di lunghezza arbitraria.

### 10.3.4 Reali

Il cosiddetto tipo dei reali (o dei numeri in virgola mobile, `float`) è costituito da:

- *valori*: un opportuno sottinsieme dei numeri razionali, di norma fissato al momento della definizione del linguaggio (ma vi sono casi in cui è fissato dalla specifica macchina astratta, il che è causa di gravi problemi di portabilità); la struttura (ampiezza, granularità ecc.) di tale sottinsieme dipende dalla rappresentazione adottata;
- *operazioni*: i confronti e un'opportuna selezione tra le principali operazioni numeriche (somma, sottrazione, moltiplicazione, divisione, esponenziazione, estrazione di radice quadrata ecc.);

I valori sono memorizzabili, esprimibili e denotabili. La rappresentazione in memoria consiste in quattro, otto od anche dieci byte, in virgola mobile secondo lo standard IEEE 754 (per linguaggi e architetture posteriori al 1985).

### 10.3.5 Virgola fissa

Il cosiddetto tipo dei reali in virgola fissa è costituito da:

- *valori*: un opportuno sottinsieme dei numeri razionali, di norma fissato al momento della definizione del linguaggio; la struttura (ampiezza, granularità ecc.) di tale sottinsieme dipende dalla rappresentazione adottata;
- *operazioni*: i confronti e un'opportuna selezione tra le principali operazioni numeriche (somma, sottrazione, moltiplicazione, divisione, esponenziazione, estrazione di radice quadrata ecc.);

I valori sono memorizzabili, esprimibili e denotabili. La rappresentazione in memoria consiste in quattro oppure otto byte; i numeri sono rappresentati in complemento a due, con un numero fissato di bit riservati per la parte decimale. I reali in virgola fissa permettono di rappresentare in modo compatto un ampio intervallo con pochi decimali di precisione.

### 10.3.6 Complessi

Il cosiddetto tipo dei complessi è costituito da:

- *valori*: un opportuno sottinsieme dei numeri complessi, di norma fissato al momento della definizione del linguaggio; la struttura (ampiezza, granularità ecc.) di tale sottinsieme dipende dalla rappresentazione adottata;
- *operazioni*: i confronti e un'opportuna selezione tra le principali operazioni numeriche (somma, sottrazione, moltiplicazione, divisione, esponenziazione, estrazione di radice quadrata ecc.);

I valori sono memorizzabili, esprimibili e denotabili. La rappresentazione in memoria consiste in una coppia di valori in virgola mobile.

### Vuoto o singoletto?

A prima vista si può essere confusi dall'affermazione che `void` ha un (solo) elemento invece che nessuno. Ragioniamo un attimo: siamo abituati a definire una funzione che "non ritorna niente" come

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

Se `void` fosse l'insieme vuoto, non potremmo scrivere alcuna funzione come `f`: non esistono funzioni con codominio vuoto, con l'unica eccezione della funzione ovunque divergente! È invece sensato supporre che in `void` vi sia un solo elemento, che è quello (implicitamente) restituito da `f`: siccome tale elemento è unico, non ci interessa (e non dobbiamo) neppure menzionare esplicitamente quale esso sia.

### 10.3.7 Void

In alcuni linguaggi esiste un tipo primitivo la cui semantica è quella di aver un solo valore; è talvolta indicato con `void` (anche se semanticamente sarebbe meglio chiamarlo `unit`, visto che non è l'insieme vuoto, ma un singoletto):

- *valori*: uno solo, che possiamo indicare con `()`;
- *operazioni*: nessuna.

A che serve un tipo del genere? Viene usato per indicare il tipo delle operazioni che modificano lo stato ma non restituiscono un valore: ad esempio, in alcuni linguaggi (ma non C o Java) un assegnamento ha tipo `void`.

### 10.3.8 Enumerazioni

In aggiunta ai tipi predefiniti, quali sono tutti quelli introdotti sin qui, troviamo nei linguaggi anche diversi modi per definire nuovi tipi. Le enumerazioni e gli intervalli sono tipi scalari definiti dall'utente.

Un tipo enumerazione consiste in un insieme finito di costanti, ciascuna caratterizzata dal proprio nome. Nel nostro pseudolinguaggio potremmo scrivere la seguente definizione:

```
type Nano = (Brontolo, Cucciolo, Dotto, Eolo, Gongolo, Mammolo,
             Pisolo);
```

che introduce un nuovo tipo di nome `Nano`, costituito da un insieme di 7 elementi, ciascuno contraddistinto dal proprio nome.

Le operazioni disponibili su un'enumerazione sono costituite dai confronti e da un meccanismo per passare da un valore al successivo e/o al precedente (si confronti con quanto abbiamo detto per il tipo dei caratteri: in Pascal, in effetti, il tipo `char` è una enumerazione predefinita).

Da un punto di vista pragmatico, le enumerazioni permettono la creazione di programmi molto leggibili, in quanto i nomi dei valori costituiscono una forma assai chiara di auto-documentazione del programma. Il controllo dei tipi, inoltre,

## Enumerazioni in C

In C la nostra definizione di Nano assumerebbe la forma

```
enum Nano {Brontolo, Cucciolo, Dotto, Eolo, Gongolo, Mammolo,
           Pisolo};
```

A parte le varianti notazionali, il punto essenziale è che, in C (ma non in C++), tale dichiarazione è sostanzialmente equivalente alle seguenti:

```
typedef int Nano;
const Nano Brontolo=0, Cucciolo=1, Dotto=2, Eolo=3, Gongolo=4,
          Mammolo=5, Pisolo=6;
```

Le regole di equivalenza tra tipi del linguaggio, in altre parole, permettono di usare un intero al posto di un Nano e viceversa: il controllo dei tipi non distingue tra i due tipi e, dunque, dall'uso di una enumerazione si ottiene solo una migliore documentazione e non un controllo più forte. Nei linguaggi che derivano da Pascal, invece, enumerazioni e interi sono tipi completamente distinti.

può essere sfruttato per verificare che una variabile di tipo enumerazione assuma solo i valori corretti.

Introdotti per la prima volta in Pascal, i tipi enumerazione sono presenti in molti altri linguaggi: il riquadro discute quelli di C.

Un valore di un'enumerazione è in genere rappresentato con un intero su un byte; i diversi valori hanno rappresentazione contigua, partendo da zero. Alcuni linguaggi (C e Ada, per esempio) permettono di scegliere i valori che corrispondono ai diversi elementi.

## 10.3.9 Intervalli

I valori di un tipo intervallo costituiscono un sottinsieme contiguo dei valori di un altro tipo scalare (il *tipo base* dell'intervallo). Due esempi in Pascal (che, come per le enumerazioni, ha introdotto per la prima volta gli intervalli):

```
type Tombola = 1..90;
AlcuniNani = Cucciolo..Eolo;
```

Nel primo caso il tipo Tombola è un intervallo di 90 elementi il cui tipo base è il tipo degli interi; l'intervallo AlcuniNani è costituito dai valori Cucciolo, Dotto, Eolo ed ha come tipo base Nano.

Come nel caso delle enumerazioni, il vantaggio di usare un tipo intervallo invece che il corrispondente tipo base è sia quello di una migliore documentazione sia quello della possibilità di un controllo dei tipi più stringente. Si osservi che la verifica che il valore di una certa espressione appartenga davvero all'intervallo deve necessariamente essere fatta in modo dinamico, anche in quei linguaggi in cui il sistema di tipi è progettato in modo da permettere un controllo statico di tutti gli altri vincoli.

Per quanto riguarda la rappresentazione, un compilatore potrebbe rappresentare un valore di tipo intervallo come un intero, su uno o due byte a seconda del numero di elementi dell'intervallo. In realtà le macchine astratte più diffuse rappresentano i valori di un intervallo nello stesso modo (e con lo stesso numero di byte) in cui è rappresentato il tipo base.

## 10.3.10 Tipi ordinali

I tipi dei booleani, dei caratteri, degli interi, le enumerazioni e gli intervalli sono esempi di *tipi ordinali* (o *tipi discreti*): sono dotati di una ben definita nozione di ordine totale e, soprattutto, possiedono una nozione di predecessore e successore per ogni elemento, esclusi gli estremi. I tipi ordinali costituiscono il dominio naturale su cui far variare gli indici di un vettore e sul quale definire delle iterazioni determinate (si veda quanto detto nel Paragrafo 8.3.3).

## 10.4 Tipi composti

I tipi non scalari sono detti *composti*, in quanto si ottengono combinando tra loro altri tipi mediante l'utilizzo di opportuni *costruttori*. I più importanti e diffusi tipi composti sono:

- *record* (o strutture), collezioni di valori eterogenei nel tipo;
- *array* (o vettori), collezioni di valori omogenei nel tipo;
- *insiemi*, sottinsiemi di un tipo base, in genere ordinale;
- *puntatori*, l-valori che permettono di accedere a dati di un altro tipo;
- *tipi ricorsivi*, tipi definiti per ricorsione, usando costanti e costruttori; casi particolari di tipi ricorsivi sono le liste, gli alberi ecc.

Li analizzeremo in quest'ordine nei paragrafi che seguono.

### 10.4.1 Record

Un record è una collezione costituita da un numero finito (e in genere ordinato) di elementi, detti *campi*, distinti dal loro nome; ciascun campo può essere di un tipo diverso dagli altri (i record sono una struttura dati *eterogenea*). Nella maggioranza dei linguaggi imperativi, ogni campo si comporta come una variabile del proprio tipo<sup>4</sup>. La terminologia non è, come al solito, univoca: in Pascal si chiamano record, in C (e C++, ALGOL 68 ecc.) si chiamano *strutture* (struct), in Java non esistono perché sussinti dalla nozione di classe.

Un semplice esempio nel nostro pseudolinguaggio, che si ispira nella notazione a C, potrebbe essere il seguente:

<sup>4</sup>In molti linguaggi non imperativi un record è invece un insieme finito di valori, ai quali si accede per nome. Tralasciando la questione dell'accesso per nome, un tipo record è in tal caso il prodotto cartesiano dei tipi dei suoi campi.

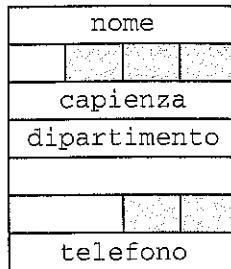


Figura 10.1 Possibile rappresentazione in memoria di un record di tipo Aula.

```
type Studente = struct{
    int matricola;
    float altezza;
};
```

Ogni record di tipo `Studente` è costituito da una coppia, di cui la prima componente è un intero, mentre la seconda è un reale.

La sola operazione permessa su un valore di tipo record è in genere la selezione di una componente, che si ottiene utilizzando il nome del campo. Ad esempio, se `s` è una variabile che fa riferimento ad uno `Studente`, possiamo assegnare dei valori ai suoi campi:

```
s.matricola = 12345;
s.altezza = 1.85;
```

In molti linguaggi i record possono essere annidati, cioè il campo di un record può essere a sua volta di tipo record:

```
type Aula = struct{
    char nome[5];
    int capienza;
    struct{
        char dipartimento[10];
        int telefono;
    } responsabile;
};
```

Il terzo campo (`responsabile`) di un record di tipo `Aula` è un record di due campi (il cui tipo, anonimo, è dunque a sua volta una `struct`: si osservi la convenzione di anteporre il tipo al nome che viene dichiarato). La selezione dei campi avviene con l'ovvia estensione della notazione: se a si riferisce ad un'Aula, con `a.responsabile.telefono` indichiamo il secondo campo del terzo campo (`responsabile`) di a.

L'uguaglianza tra record non è sempre definita (Ada la permette, ma Pascal, C, e C++ non la consentono); analogamente, non è sempre consentito l'assegnamento di record nella loro interezza. Laddove queste operazioni non siano con-

sentite, l'utente del linguaggio deve esplicitamente programmarle, per esempio confrontando (o assegnando) un campo alla volta.

L'ordine dei campi è in genere significativo ed è rispettato nella rappresentazione in memoria. I campi di un record sono memorizzati in locazioni contigue, anche se motivi di allineamento possono imporre di lasciare dei buchi tra un campo ed un altro. Ad esempio, in un'architettura a 32 bit un record di tipo `Aula` potrebbe essere rappresentato come descritto in Figura 10.1. Il campo `nome` è rappresentato su 5 byte. Il campo successivo, tuttavia, essendo un intero deve essere allineato alla parola e dunque tra `nome` e `capienza` sono lasciati vuoti 3 byte. Una situazione analoga si ripresenta coi campi `dipartimento` e `telefono`. Un record di tipo `Aula` viene così rappresentato su 7 parole di 4 byte, anche se soli 23 byte sono significativi<sup>5</sup>. Per ovviare a questi problemi alcuni linguaggi non assicurano che l'ordine dei campi venga mantenuto dalla macchina astratta, permettendo così al compilatore la riorganizzazione dei campi al fine di minimizzare i buchi dovuti all'allineamento. Il nostro record di tipo `Aula` potrebbe essere così rappresentato su 6 parole, con spreco di un solo byte. Si tratta di una soluzione ottimale se il linguaggio garantisce un livello di astrazione significativo tra l'utilizzatore e l'implementazione. In alcune applicazioni, al contrario, si vuole permettere agli utenti di manipolare l'implementazione dei tipi: linguaggi come C sono progettati proprio per permettere *anche* questo genere di operazioni. In tal caso la riorganizzazione dei campi non sarebbe una buona scelta progettuale.

## 10.4.2 Record varianti e unioni

Una forma particolare di record è quella in cui alcuni campi sono tra loro mutuamente esclusivi: parliamo in tal caso di *record varianti*. Si tratta di una possibilità che, con vari nomi e con sintassi e vincoli molto diversi, è offerta in molti linguaggi, ma che, come vedremo tra breve, è anche causa di molte complicazioni. A differenza del nostro solito modo di procedere (che usa uno pseudolinguaggio neutro), inizieremo con un esempio in Pascal, uno dei linguaggi in cui la nozione di record variante è presente con la maggior chiarezza (ma anche con tutti i suoi problemi...). In Pascal potremmo dichiarare un tipo `Stud` come segue (la corrispondente definizione in C, che discuteremo in seguito, è data in Figura 10.2):

```
type Stud = record
    nome : array [1..6] of char;
    matricola : integer;
    case fuoricorso : boolean of
        true: (ultimoanno : 2000..maxint);
        false:(inpari : boolean;
                anno : (primo,secondo,terzo)
            )
end;
```

<sup>5</sup>La presenza dei buchi è spesso il motivo per cui alcuni linguaggi non consentono l'uguaglianza tra record nella loro interezza: un confronto bit a bit rischia di distinguere due record che sono in realtà coincidenti eccetto che per dell'informazione irrilevante presente nei buchi.

```

struct Stud{
    char nome[6];
    int matricola;
    int fuoricorso;
    union{
        int ultimoanno;
        struct{
            int inpari;
            int anno;
        } stud_in_corso;
    } campivarianti;
};

```

Figura 10.2 Un record variante in C mediante l'uso di union.

I primi due campi del record sono un vettore di 6 caratteri (nome) e un intero (matricola). Il terzo campo, fuoricorso, preceduto dalla parola riservata case, è il tag, o discriminante del record variante. Un tag true indica che il record ha un ulteriore campo: ultimoanno di tipo intervallo. Un tag false indica invece che il record ha due ulteriori campi: inpari di tipo booleano e anno di tipo enumerazione. Le due possibilità che compaiono tra parentesi tonde sono le varianti del record. In generale, il tag può essere di un qualsiasi tipo ordinale e può dunque essere seguito da un numero di varianti pari alla cardinalità di quel tipo.

Da un punto di vista semantico, in un record variante solo una delle due varianti è significativa, mai entrambe. Quale delle due varianti sia significativa è indicato dal valore del tag; al tag e alle varianti si può accedere come ad un qualsiasi altro campo. Ad esempio, se s si riferisce ad uno Stud,

```

s.fuoricorso := true
assegna un valore al tag, mentre
s.ultimoanno := s.ultimoanno+1
incrementa di uno la prima variante.

```

Per quanto riguarda la rappresentazione in memoria, le varianti condividono la stessa zona di memoria, visto che esse non possono essere attive contemporaneamente: in effetti il risparmio di memoria fu uno dei motivi che portarono all'introduzione di questo tipo di dato, anche se oggi questo non è più una preoccupazione primaria. La Figura 10.3 illustra lo schema di allocazione per uno Stud e due possibili situazioni che si possono presentare.

**Unioni in C** In C non esiste il concetto primitivo di record variante, ma quello di tipo *unione*. Un'unione è analoga ad un record (*struct*) nella definizione e nella selezione dei campi, ma con la fondamentale differenza che solo uno dei

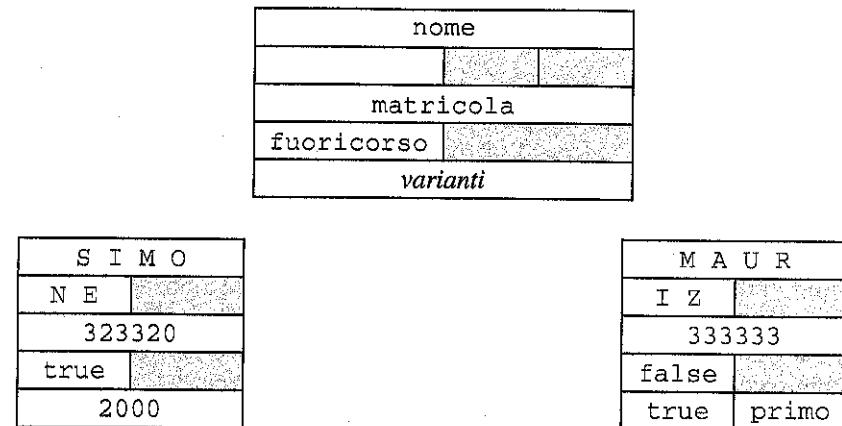


Figura 10.3 Possibile rappresentazione in memoria di un record variante.

campi di un'unione può essere attivo in un qualsiasi momento, visto che i campi (che possono essere di tipo diverso) condividono la rappresentazione in memoria. Si vede così subito che la Figura 10.2 corrisponde esattamente al record variante che abbiamo già discusso in Pascal<sup>6</sup>. L'unica differenza (non di poco conto, da un punto di vista pragmatico) risiede nella necessità, in C, di aggiungere un ulteriore livello di nomi: il quarto campo della struttura, di nome campivarianti, è di tipo unione e serve per poter accedere ai veri e propri varianti. Si osservi, inoltre, come la seconda variante (che è costituita da una coppia di informazioni) debba essere dichiarata esplicitamente come una struttura, con proprio nome (stud.in.corso). Per accedere al campo inpari, se s si riferisce ad uno Stud, occorre scrivere

```
s.campivarianti.stud_in_corso.inpari
```

che è certo più pesante del semplice s.inpari di Pascal.

Da un punto di vista linguistico, tuttavia, la differenza più rilevante tra i record varianti di Pascal e la loro simulazione in C mediante le unioni è che in C il tag è del tutto svincolato dall'unione: come si vede nella Figura 10.2, fuoricorso è un campo come tutti gli altri e solo la disciplina del programmatore lo mette in relazione al significato dei campi dell'unione che lo segue. Il seguente frammento, ad esempio, è del tutto legale:

```
s.fuoricorso = 0; // 0 sta per falso: C non ha booleani primitivi
```

<sup>6</sup>Osserviamo, da un punto di vista notazionale, che la struct più esterna (seguita subito dal nome Stud) è una definizione di tipo (il tipo Stud, appunto), mentre la struct più interna (che non ha nomi prima della parentesi graffa) è un'espressione di tipo anonima che serve a determinare il tipo del campo il cui nome (stud.in.corso) segue l'espressione di tipo.

```
s.campivarianti.ultimoanno = 2001;
if (s.fuoricorso) printf("%d", s.campivarianti.ultimoanno);
else print("%d", s.campivarianti.stud_in_corso.anno);
```

Nonostante sia legale sintatticamente, è evidente che pone qualche problema semantico: il "tag" è falso (ad indicare che è attiva la seconda componente dell'unione), ma si assegna un valore al primo campo dell'unione, salvo poi accedere (per la stampa) alla seconda componente.

**Varianti e sicurezza** La discussione che abbiamo appena fatto circa le unioni di C sembrerebbe suggerire che i record varianti di Pascal siano da preferire alla soluzione "record+unioni" di C. Non c'è dubbio che Pascal sia più elegante: fondendo nell'unico concetto di record variante i due meccanismi di record e unione, il linguaggio costruisce un meccanismo più chiaro, più compatto e, si vorrebbe dire, più sicuro. Peccato che quest'ultimo aggettivo non si possa utilizzare in questo contesto, perché, per quanto riguarda la sicurezza dei tipi, qui Pascal si comporta esattamente come C.

Il fatto è che, anche in Pascal, il linguaggio non riesce a garantire che vi sia un legame formale e controllato tra il valore del tag e la significatività di una delle varianti che lo seguono. Il punto cruciale è che il tag è accessibile mediante un assegnamento ordinario.

Si potrebbe pensare di imporre alla macchina astratta di controllare che ogni accesso ad una variante avvenga solo se il tag ha il valore corretto (segnalando un errore dinamico altrimenti). A parte il fatto che la definizione del linguaggio non richiede tale controllo, il problema è che questa soluzione sarebbe in grado di catturare (dinamicamente) alcuni errori semantici, ma non tutti. Supponiamo infatti di essere nel contesto delle dichiarazioni

```
type Tre = 1..3;
var tmp : record
  case quale : Tre of
    1 : (a : integer);
    2 : (b : boolean);
    3 : (c : char)
end
```

Questa ipotetica macchina astratta che controlla il tag prima dell'accesso sarebbe in grado di segnalare (dinamicamente) l'errore nel seguente programma (la situazione d'errore è analoga a quella che abbiamo discusso in C poco fa):

```
tmp.quale := 1;
tmp.a := 123;
writeln(tmp.c); (* errore dinamico *)
```

Ma il frammento che segue, invece, passerebbe indenne tutti i controlli, nonostante vi sia un accesso al campo c che non ha ancora ricevuto assegnamenti significativi:

```
tmp.quale := 1;
tmp.a := 123;
tmp.quale := 3;
writeln(tmp.c); (* nessun errore, ma c non è significativo! *)
```

### Unioni sicure: ALGOL 68

È possibile progettare record varianti sicuri, cioè che non minino alla radice il sistema di tipi di un linguaggio? La risposta è: certamente sì, a patto di volerne pagare il prezzo, sia linguistico che di efficienza. Discuteremo qui per sommi capi la soluzione di ALGOL 68, che precede di qualche anno il progetto di Pascal (Ada è un altro linguaggio con varianti sicure). ALGOL 68 permette la definizione di tipi unione e impone alla macchina astratta di seguire l'evoluzione del tipo di una qualsiasi variabile di tipo unione, senza tag esplicativi:

```
union (int, bool, char) tmp; # tmp è di tipo unione #
...
tmp := true; # ora tmp è un booleano #
...
tmp := 123; # ora tmp è un intero #
```

La macchina astratta mantiene per ogni variabile di tipo unione un tag nascosto, che viene settato implicitamente al momento di ogni assegnamento. Il punto cruciale è come possa essere usata un'unione: attraverso una "clausola di conformità" (un case) che specifica cosa fare *in tutti i casi possibili* per quella variabile. Ad esempio:

```
case tmp in
  (int a) : a := a+1,
  (bool b) : b := not b,
  (char c) : print(c)
esac
```

Una clausola di conformità è l'unico costrutto di tutto il linguaggio che genera controlli di tipo dinamici. Oltre all'aggravio di efficienza, è palese il peso linguistico di una soluzione del genere.

La situazione è perfino peggiore, perché il tag non è obbligatorio. La seguente definizione di record variante è perfettamente legale:

```
var tmp : record
  case Tre of
    1 : (a : integer);
    2 : (b : boolean);
    3 : (c : char)
  end
```

Le varianti sono ancora discriminate rispetto ai valori del tipo Tre, ma nel record non viene riservato alcuno spazio per memorizzare il tag. In questo caso anche quella limitata forma di controllo sugli accessi che abbiamo discusso poc' anzi diviene impossibile.

La conseguenza di tutto ciò è che la presenza di record varianti siffatti distrugge la sicurezza relativamente ai tipi di Pascal. Ci si potrebbe pertanto domandare se valga la pena avere un costrutto così "costoso" in termini di sicurezza in un linguaggio che pone i tipi a cardine del proprio progetto. La risposta, nella prospettiva odierna, è negativa: il risparmio di memoria (e concettuale) che i record

varianti assicurano non è giustificato dai problemi che essi causano al sistema di tipi<sup>7</sup>. I record varianti (o le unioni) non esistono in Modula-3 o in Java.

### 10.4.3 Array

Un array (o vettore) è una collezione finita di *elementi* dello stesso tipo, indicizzata su un intervallo di un tipo ordinale<sup>8</sup>; ogni elemento si comporta come una variabile del proprio tipo. L'intervallo ordinale degli indici è il *tipo indice*, o tipo degli indici, dell'array; il tipo degli elementi è il *tipo dei componenti*, o anche, con una certa imprecisione, il *tipo dell'array*. Siccome tutti gli elementi sono dello stesso tipo, gli array sono una struttura dati *omogenea*.

Gli array sono senza dubbio il tipo composto più diffuso nei linguaggi di programmazione, a partire da FORTRAN, il capostipite dei linguaggi di alto livello. La sintassi e le varie caratteristiche degli array, tuttavia, variano notevolmente da linguaggio a linguaggio. Gli ingredienti fondamentali della dichiarazione di un array sono costituiti dal suo nome, il suo tipo indice, il tipo dei suoi elementi. Iniziamo con uno degli esempi più semplici che possiamo trovare in C:

```
int V[10];
```

La parentesi quadra dopo il nome della variabile indicano che si tratta di un array, costituito da 10 elementi, ciascuno dei quali è una variabile di tipo intero (d'ora in poi diremo semplicemente: un array di 10 interi). Il tipo degli elementi è dunque int, mentre il tipo degli indici è costituito da un intervallo di 10 elementi; nella convenzione di molti linguaggi (C, C++, Java ecc.) tale intervallo parte da 0. In questo caso, dunque, il tipo degli indici è l'intervallo da 0 a 9. In generale, possiamo supporre che un linguaggio permetta di dichiarare il tipo indice come intervallo arbitrario in un tipo ordinale; per esempio:

```
int W[21..30];
type Nano = {Brontolo, Cucciolo, Dotto, Eolo, Gongolo, Mammolo,
             Pisolo};
float Z[Dotto..Mammolo];
```

Nella prima dichiarazione W è un array di 10 interi, con indici interi da 21 a 30; l'ultima riga dichiara Z, un array di 4 reali, con indici presi dal tipo Nano che variano da Dotto a Mammolo.

Tutti i linguaggi permettono la definizione di array *multidimensionali*, cioè array indicizzati su due o più tipi indice:

```
int V[1..10,1..10];
char C[Dotto..Mammolo,0..10,1..10];
```

<sup>7</sup> Ma si ricordi che Pascal è un linguaggio progettato agli inizi degli anni settanta, quando il problema dell'occupazione della memoria centrale durante l'esecuzione costituiva un collo di bottiglia non trascurabile.

<sup>8</sup> Da un punto di vista semantico, un array è una funzione che ha come dominio l'intervallo degli indici, e come codominio il tipo degli elementi degli array.

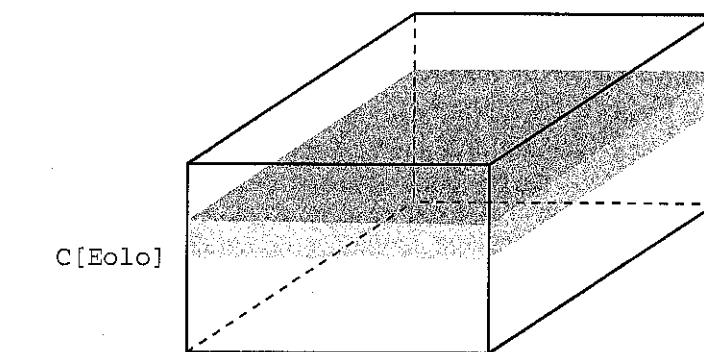


Figura 10.4 Slice su array.

L'array V è una matrice quadrata  $10 \times 10$  di interi, con indici di riga e colonna da 1 a 10; C è una matrice  $4 \times 11 \times 10$  di caratteri.

In alcuni linguaggi, un array multidimensionale si può ottenere dichiarando che il tipo degli elementi dell'array è a sua volta un array. Una possibile sintassi nel nostro pseudolinguaggio potrebbe essere, dichiarando gli stessi array di poc'anzi:

```
int V[1..10][1..10];
char C[Dotto..Mammolo][0..10][1..10];
```

Per qualche linguaggio le due modalità (array multidimensionali e array di array) sono equivalenti (l'una è un'abbreviazione per l'altra): è il caso di Pascal. In altri solo una delle due possibilità è ammessa: in C, un array multidimensionale è un array di array e deve essere dichiarato come tale. Altri linguaggi, infine, ammettono entrambe le modalità, ma permettendo sugli array di array alcune operazioni (lo *slicing*, che vedremo tra poco) che non sono contemplate per gli array multidimensionali.

**Operazioni sugli array** L'operazione più semplice permessa su un array è la *selezione di un elemento*, che si ottiene mediante un valore dell'indice. La notazione più comune è quella che fa uso di parentesi quadre: W[e] indica l'elemento di W corrispondente all'indice dato dal valore dell'espressione e. Per array multidimensionali possiamo trovare C[Eolo,1,2] oppure C[Eolo][1][2], a seconda se si enfatizzi l'aspetto multidimensionale o quello di array di array.

Alcuni linguaggi, inoltre, permettono operazioni su array nella loro globalità: assegnamento, uguaglianza, confronti ed anche operazioni aritmetiche (eseguite, in genere, elemento per elemento). In altri linguaggi queste operazioni globali sono solo un caso particolare di operazioni che permettono di selezionare intere porzioni di array, e di operare su di esse in modo globale. Una *slice* (fetta) di un array è una sua porzione, costituita da elementi contigui. Con le dichiarazioni di prima, V[3] potrebbe indicare la terza riga della matrice V, e C[Eolo] il piano

della matrice tridimensionale  $C$  che si ottiene selezionando la sua sola prima componente (con valore  $E_{0,0}$ ) (Figura 10.4). Altri linguaggi, infine, permettono la selezione di slice ancora più sofisticate: diagonali, cornici ecc.

**Controlli** La specifica del tipo indice di un array è parte integrante della sua definizione. Il controllo dei tipi del linguaggio, pertanto, dovrebbe verificare che ogni accesso ad un elemento del vettore avvenga davvero “entro i limiti” dell’array e che non si tenti di accedere ad elementi che non esistono. Ad eccezione di alcuni casi speciali, tale controllo può avvenire solo a tempo di esecuzione. Un linguaggio sicuro, pertanto, dovrà far sì che il compilatore generi opportuni controlli in corrispondenza di *ogni accesso*. Siccome tali controlli influiscono sull’efficienza del programma, alcuni linguaggi, che pur permettono di generare tali controlli, consentono di disattivarli (per esempio Pascal).

Osserviamo, per inciso, che si tratta di una questione di non poco conto per la sicurezza di un sistema, intesa non solo come “type safeness”, ma come “security” in senso proprio. Uno degli attacchi più comuni e più seri alla sicurezza di un sistema è quello che va sotto il nome di *buffer overflow*: un agente maligno manda messaggi sulla rete con lo scopo di farli leggere in un buffer dal destinatario. Se il destinatario non controlla che la lunghezza del messaggio non superi la capacità del buffer, il mittente maligno ha la capacità di scrivere in un’area di memoria che non è più quella allocata per il buffer. Se il buffer è allocato in un RdA ed il mittente riesce a scrivere nell’area dell’RdA riservata per il salvataggio dell’indirizzo di ritorno, ciò può causare un “ritorno” ad una qualsiasi istruzione, in particolare a codice maligno appositamente caricato (per esempio nel buffer stesso di cui si è sfruttato l’overflow). Nella quasi totalità dei casi, un attacco di buffer overflow può essere impedito da una macchina astratta che controlli che ogni accesso ad un array avvenga “entro i limiti”.

**Memorizzazione e calcolo degli indici** Un array è memorizzato di solito in una porzione contigua di memoria (si veda l’Esercizio 2 per una tecnica di memorizzazione non contigua). Per un array monodimensionale, l’allocazione segue l’ordine degli indici. Nel caso di array multidimensionali si seguono due tecniche alternative, dette memorizzazione in *ordine di riga* e in *ordine di colonna*. In ordine di riga, sono contigui due elementi che differiscono di uno nell’ultimo indice (eccetto nel caso degli elementi agli estremi di una riga, o di un piano ecc.). In ordine di colonna, sono contigui due elementi che differiscono di uno nel primo indice (eccetto nel caso degli estremi). L’ordine di riga è un po’ più diffuso di quello di colonna, anche perché memorizza in modo contiguo gli elementi di una riga e dunque rende più semplice la selezione di una slice di riga (che è l’operazione di slicing più comunemente fornita).

Le due modalità di memorizzazione non sono equivalenti, quanto ad efficienza, in presenza di cache. I programmi che manipolano grandi array multidimensionali sono spesso caratterizzati da cicli annidati che “spazzano” tali array. Se l’array non entra per intero in cache, è importante che si acceda agli elementi del vettore “lungo” la cache, in modo che il primo miss porti in cache gli elementi

che saranno acceduti immediatamente dopo. Se il ciclo opera per righe, anche la cache dovrebbe essere caricata per righe, ovvero conviene una memorizzazione in ordine di riga; se il ciclo opera per colonne, converrebbe una memorizzazione in ordine di colonna.

Una volta memorizzato in modo contiguo un array, il calcolo dell’indirizzo corrispondente ad un suo generico elemento non è difficile, anche se richiede un po’ di aritmetica. Consideriamo il generico array a  $n$  dimensioni, con elementi di tipo  $T$

$$T \ V[L_1..U_1] \dots [L_n..U_n];$$

Sia  $S_n$  il numero di unità indirizzabili (tipicamente byte) necessarie per memorizzare un singolo elemento di tipo  $T$ . A partire da essa, possiamo calcolare successivamente una serie di valori che esprimono la quantità di memoria necessaria a memorizzare slice sempre più grandi di  $V$ . Supponendo di lavorare in ordine di riga, si ha

$$\begin{aligned} S_{n-1} &= (U_n - L_n + 1)S_n, \\ &\dots \\ S_1 &= (U_2 - L_2 + 1)S_2. \end{aligned}$$

Ad esempio, per  $n = 3$ ,  $S_2$  è la quantità di memoria necessaria per una riga, mentre  $S_1$  esprime la memoria necessaria per un intero piano di  $V$ .

L’indirizzo dell’elemento  $V[i_1, \dots, i_n]$  si ottiene ora sommando all’indirizzo al quale inizia la memorizzazione di  $V$  la quantità

$$(i_1 - L_1)S_1 + \dots + (i_n - L_n)S_n.$$

Nel caso in cui il numero delle dimensioni (cioè  $n$ ) e i valori degli  $L_j$  e  $U_j$  siano tutti noti a tempo di compilazione (cioè la *forma* dell’array sia statica, vedi il paragrafo successivo) è conveniente riformulare tale espressione dell’indirizzo come

$$i_1S_1 + \dots + i_nS_n - (L_1S_1 + \dots + L_nS_n), \quad (10.1)$$

nella quale la seconda parte (tra parentesi), e *a fortiori* tutti gli  $S_j$ , sono costanti determinabili a tempo di compilazione. Sfruttando questa formulazione, un indirizzo si calcola (dinamicamente) con  $n$  moltiplicazioni e  $n$  addizioni. La sottrazione finale, poi, scompare qualora tutti gli  $L_j$  siano zero (il che spiega perché alcuni linguaggi usino zero come valore del primo limite degli indici).

**Forma di un array: dove viene allocato un array** La *forma*, o *shape*, di un array è costituita dal numero delle sue dimensioni e dall’intervallo su cui ciascuna di esse può variare. Un aspetto importante della definizione di un linguaggio è la decisione su quando debba essere fissata la forma di un array. In corrispondenza dei tre casi principali, abbiamo anche tre modalità diverse di allocazione in memoria di un array:

- **forma statica:** in tal caso tutto è deciso a tempo di compilazione e l’array può essere memorizzato nel RdA del blocco in cui compare la sua definizione (o

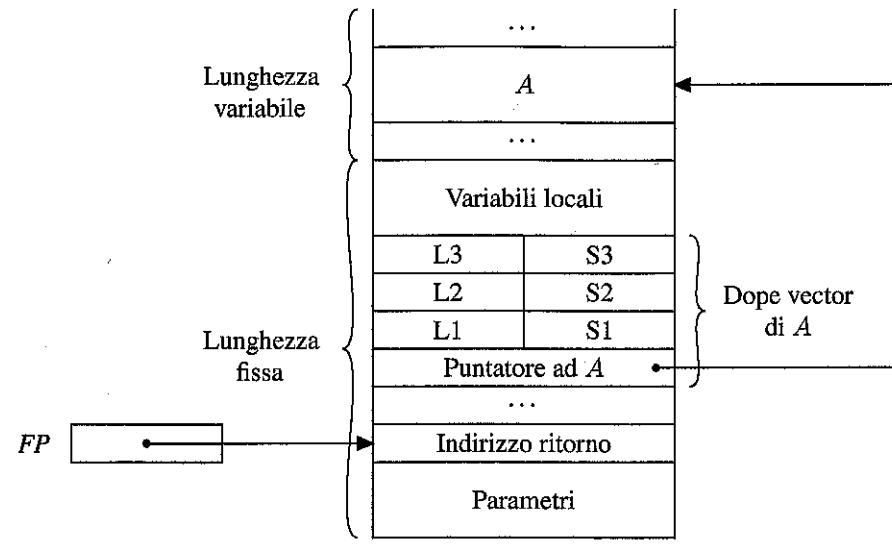


Figura 10.5 Struttura di un RdA con dope vector.

nella memoria statica allocata per i globali, se questo è il caso). Osserviamo che, in questo caso, la dimensione totale necessaria per un array è una costante nota a tempo di compilazione: è pertanto una costante anche l'offset tra l'indirizzo di inizio dell'array ed il punto fissato dell'RdA a cui si riferisce il puntatore all'RdA (si veda quanto detto nel Paragrafo 7.3.3). L'accesso ad un elemento dell'array si differenzia da un accesso ad una variabile ordinaria ("scalare") solo per la somma della quantità determinata con l'espressione (10.1).

- forma *fissata al momento dell'elaborazione della dichiarazione*: in questo caso a tempo di compilazione non si è in grado di definire la forma, ma questa è nota, e fissata, al momento in cui il controllo raggiunge la dichiarazione dell'array (ad esempio, l'intervallo dell'indice dipende dal valore di una variabile). Anche in questo caso l'array può essere allocato nell'RdA del blocco in cui compare la sua definizione, ma il compilatore non ha modo di conoscere l'offset tra l'inizio dell'array e il punto fissato dell'RdA cui si riferisce il frame pointer. Questa è una situazione spiacevole, perché potrebbe ripercuotersi anche su altre strutture dati completamente statiche. Per ovviare a questo problema, un RdA viene diviso in due parti: una parte per i dati di lunghezza fissa, un'altra per quelli di lunghezza variabile. A tutti i dati allocati nella parte a lunghezza fissa si accede per offset, nel modo usuale. Ai dati allocati nella parte a lunghezza variabile, invece, si accede per accesso indiretto attraverso un descrittore dei dati a lunghezza variabile. Il descrittore è contenuto nella parte a lunghezza fissa, di modo che ad esso si acceda per offset, e contiene, tra le altre cose, un puntatore all'inizio della struttura dati (nel nostro caso un array). La Figura 10.5

### Array in Java

In Java un array non viene creato al momento della dichiarazione di una variabile di tipo array: `int[] v;` introduce solo il nome `v`, che non è associato ad alcuna struttura dati (esattamente come una qualsiasi dichiarazione di una variabile di un tipo classe). La struttura dati viene creata, sullo heap, mediante l'operazione predefinita `v = new int[10];` crea sullo heap un nuovo array di 10 interi ed assegna un suo riferimento al nome `v`.

Un array in Java è un oggetto, nel senso tecnico di questo termine nel contesto di quel linguaggio, con tempo di vita illimitato.

riporta uno schema di allocazione in un RdA in questa situazione. Il descrittore di un array va sotto il nome di *dope vector* e lo descriveremo più in dettaglio tra un attimo.

- forma *dinamica*: in questo caso un array può cambiare forma dopo la sua creazione, per effetto dell'esecuzione. L'allocazione sulla pila non è più possibile, perché dovrebbe essere dinamicamente modificata l'ampiezza (e la struttura) di un RdA. Gli array dinamici devono pertanto essere allocati sullo heap, mentre un puntatore all'inizio dell'array rimane memorizzato nella parte di lunghezza fissa dell'RdA.

Per concludere l'argomento, osserviamo che oltre alla staticità o dinamicità della forma, per decidere sull'allocazione di un array occorre tener conto anche del suo tempo di vita. Un array allocato in un RdA ha il proprio tempo di vita limitato da quello del blocco in cui avviene la dichiarazione. In linguaggi che permettono la creazione di array con tempo di vita illimitato (per esempio Java), questi devono essere comunque allocati sullo heap.

**Dope vector** Il descrittore di un array di forma non nota staticamente (sia essa fissata a tempo d'esecuzione che completamente dinamica) prende il nome di *dope vector*. Un dope vector, usualmente allocato nella parte di lunghezza fissa di un RdA, contiene:

- un puntatore alla prima locazione nella quale è memorizzato l'array;
- tutte le informazioni dinamiche necessarie al calcolo della quantità (10.1): il numero di dimensioni (detto anche il *rango* dell'array), il valore del limite inferiore di ogni dimensione (i valori  $L_j$ ), l'occupazione di ogni dimensione (i valori  $S_j$ ).

Qualora qualcuna di queste quantità fosse staticamente nota, essa non viene ovviamente memorizzata. Il dope vector è inizializzato al momento della creazione dell'array (con gli opportuni calcoli dei valori  $S_j$ ). Per accedere ad un elemento dell'array, si accede (per offset mediante il frame pointer) al dope vector; si calcola l'espressione (10.1) e si somma (mediante un accesso indiretto) all'indirizzo d'inizio dell'array.

#### 10.4.4 Insiemi

Alcuni linguaggi di programmazione (in particolare Pascal e i suoi discendenti) permettono di definire tipi insieme, i cui valori sono costituiti dai sottinsiemi di un tipo base (o universo), usualmente ristretto ad essere un tipo ordinale. Ispirandoci per la sintassi (molto adattata) da Pascal:

```
set of char S;
set of Nano N;
```

abbiamo una variabile *S* che è un sottinsieme dei caratteri, ed una variabile *N* che conterrà un sottinsieme del tipo *Nano* che abbiamo definito a pag. 289. I linguaggi che permettono insiemi offrono opportuna sintassi per assegnare uno specifico sottinsieme ad una variabile, per esempio

```
N = (Eolo, Brontolo);
```

Le operazioni possibili sui valori di tipo insieme sono il test di appartenenza di un elemento ad un insieme e le usuali operazioni insiemistiche di unione, intersezione, differenza; non sempre è disponibile il complemento.

Un insieme è in genere rappresentato come un vettore di bit (il vettore *caratteristico* dell'insieme), di lunghezza pari alla cardinalità del tipo base: ad esempio, un sottinsieme di *char*, su un'implementazione che usi il codice ASCII a 7 bit, sarebbe rappresentato su 128 bit. Nel vettore caratteristico il *j*-esimo bit ad uno indica che l'elemento *j*-esimo del tipo base (nell'enumerazione standard) appartiene all'insieme; un bit a zero indica che il corrispondente elemento non appartiene all'insieme. Se questa rappresentazione consente l'esecuzione molto efficiente delle operazioni insiemistiche (come operazioni bit a bit sulla macchina fisica), è anche evidente come sia assolutamente inadatta per sottinsiemi di tipi base di cardinalità maggiore di qualche centinaia. Per ovviare a questo problema, e proprio per questo scopo, i linguaggi limitano spesso i tipi che possono essere usati come tipi base di un insieme. Oppure scelgono altre rappresentazioni (per esempio tabelle hash), che permettono di rappresentare in modo più compatto gli insiemi a danno dell'efficienza delle loro operazioni.

#### 10.4.5 Puntatori

Alcuni linguaggi permettono di manipolare direttamente l-valori: il tipo corrispondente è detto tipo dei puntatori. In genere, viene permessa la definizione di un tipo di puntatori per ogni tipo del linguaggio: se *T* è un tipo, è possibile definire il tipo dei "puntatori a (variabili di tipo) *T*". Si osservi che, in questo contesto, ci interessiamo di puntatori in quanto tipo esplicitamente presente nel linguaggio, in contrasto con l'ovvia presenza di puntatori nella macchina astratta.

In un linguaggio con modello delle variabili a riferimento, un tipo dei puntatori non è generalmente necessario (o opportuno): ogni variabile è sempre un riferimento, ossia è sempre considerata come un l-valore, anche se tale valore non può essere manipolato esplicitamente. Nei linguaggi con variabili modificabili, invece, i puntatori forniscono un modo di riferirsi ad un l-valore senza dereferenziarlo automaticamente. In tali linguaggi uno degli usi principali dei puntatori è

quello di costruire valori di tipi ricorsivi (come strutture concatenate), che non sono in genere forniti come primitivi dal linguaggio stesso.

Nel nostro pseudolinguaggio indicheremo con *T\** il tipo dei puntatori ad oggetti di tipo *T* e dunque dichiareremo un puntatore come<sup>9</sup>:

```
T* p;
```

I valori del tipo *T\** sono puntatori (da un punto di vista implementativo: indirizzi) a locazioni di memoria (cioè variabili modificabili) che contengono valori di tipo *T*. Non è detto che tali puntatori possano riferirsi a locazioni arbitrarie: alcuni linguaggi richiedono che i puntatori puntino solo ad oggetti allocati sullo heap (è il caso di Pascal, dei suoi discendenti, e delle prime versioni di Ada). Altri linguaggi, invece, ammettono che un puntatore possa puntare anche a locazioni sulla pila di sistema o nell'area globale (è il caso di C, C++ e le versioni posteriori di Ada).

Tra tutti i valori che un puntatore può assumere ne esiste di norma uno canonico (che appartiene al tipo *T\** per ogni *T*) che indica il puntatore nullo, cioè che non punta ad alcun valore. Lo indicheremo con *null*. Le operazionimesse sui valori di tipo puntatore sono di norma la loro creazione, il test di uguaglianza (in particolare l'uguaglianza con *null*), la loro dereferenziazione (cioè l'accesso all'oggetto puntato).

Il modo più comune per creare un valore di tipo puntatore è quello di usare un costrutto predefinito o una funzione di libreria che contemporaneamente alloca sullo heap un oggetto del tipo opportuno e restituisce un riferimento a tale oggetto. È questo il ruolo di *malloc* in C:

```
int* p;
p = (int *) malloc (sizeof (int));
```

o di *new* in Pascal. Per la gestione dello heap in presenza di allocazione esplicita, rimandiamo a quanto abbiamo già detto nel Paragrafo 7.4.

In alcuni linguaggi è possibile creare puntatori anche applicando opportuni operatori che restituiscono l'indirizzo di memorizzazione di un oggetto allocato in memoria. Ancora una volta è C a guidarci negli esempi, con l'operatore *&*:

```
float r = 3.1415;
float* q;
q = &r;
```

Il puntatore *q* punta ora alla locazione che contiene la variabile *r*. Si osservi come ciò costituisca anche un esempio di un puntatore che punta ad una locazione sulla pila e non sullo heap.

<sup>9</sup>Il programmatore madrelingua C scriverebbe l'esempio con una diversa distribuzione degli spazi: *T \*p*, leggendo che *\*p* è un puntatore a *T*. In effetti in C *\** è un modificatore della variabile e non del tipo (il che si vede bene qualora si vogliano introdurre due puntatori con la stessa dichiarazione: C scrive *int \*p, \*q;*, mentre noi scriveremmo *int\* p, q;* per uniformità con gli altri tipi). Sebbene l'uso di C sembri più semplice (soprattutto se visto insieme all'operatore di dereferenziazione), è semanticamente più corretto (e più uniforme rispetto ad altri linguaggi), vedere *\** come un modificatore del tipo e non della variabile.

La dereferenziazione di un puntatore viene di solito indicata con un opportuno operatore, per esempio lo `*` prefisso di C. Continuando gli esempi di poc' anzi:

```
*p = 33;
r = *q + 1;
```

nel quale la prima linea assegna il valore 33 all'oggetto puntato da `p` (33 è sullo heap), mentre la seconda linea assegna a `r` il valore 4.1415 (sulla pila). Si osservi che, per effetto collaterale, il secondo assegnamento modifica anche il valore puntato da `q` (ma non `q` stesso!). Si osservi, infine, come la distinzione l-valore/r-valore di una variabile in un assegnamento rimane valida anche per un puntatore dereferenziato: quando `*p` è a sinistra dell'assegnamento, indica lo l-valore che si ottiene dereferenziando `p` (cioè l'assegnamento avviene all'indirizzo "contenuto nel puntatore"), mentre quando `*p` è a destra dell'assegnamento indica lo r-valore di quella stessa locazione.

Da un punto di vista pragmatico, i puntatori svolgono un ruolo chiave nella definizione di strutture dati *ricorsive*, quali le liste, gli alberi ecc. Alcuni linguaggi permettono direttamente la definizione di tipi ricorsivi, come vedremo nel Paragrafo 10.4.6. Nei linguaggi con puntatori, strutture dati ricorsive possono essere definite in modo naturale. Ad esempio, una lista di interi<sup>10</sup> può essere definita come

```
type int_list = elemento*;
type elemento = struct {int val;
                        int_list next;};
```

Si osservi che, affinché una tale definizione sia legale in un ipotetico linguaggio, devono essere risolti i problemi relativi all'uso di un nome prima della sua definizione, cui abbiamo accennato nell'Approfondimento 6.1.

**Aritmetica dei puntatori** In C e nei suoi discendenti, oltre alle operazioni che abbiamo appena discusso, sui puntatori è possibile effettuare anche alcune operazioni "aritmetiche": è possibile incrementare un puntatore, sottrarre tra loro due puntatori (così da ottenere una costante di offset), sommare una quantità arbitraria ad un puntatore. Questo complesso di operazioni va sotto il nome di aritmetica dei puntatori. La semantica di queste operazioni, sebbene indicate con gli operatori aritmetici standard, deve essere intesa con riferimento al tipo dei puntatori su cui operano. A titolo d'esempio, consideriamo il seguente frammento:

```
int* p;
int* c;
p = (int *) malloc(sizeof(int));
c = (char *) malloc(sizeof(char));
```

<sup>10</sup>Come abbiamo visto nell'Approfondimento 6.1, una lista è una struttura dati a dimensione variabile, costituita da una successione ordinata, eventualmente vuota, di elementi presi da qualche altro tipo, in cui è possibile aggiungere o togliere degli elementi e dove si può accedere direttamente solo al primo elemento.

## Array e puntatori in C

In C array e puntatori sono "interoperabili". La dichiarazione di un array introduce un nome che può essere usato anche come un puntatore all'array stesso; inversamente, se un puntatore ad interi si riferisce ad un array, può essere usato anche come il nome di un array:

```
int V[10];
int* W;
W = V;           // W punta all'inizio dell'array V
V[1] = 5;
W[1] = 5;
*(W+1) = 5;
*(V+1) = 5;
```

I quattro ultimi assegnamenti di questo esempio sono tutti equivalenti: assegnano 5 al secondo elemento dell'unico array puntato da `v` e `w` (si osservi come l'aritmetica dei puntatori sia molto naturale in questo contesto). Nel caso di array multidimensionali (memorizzati in ordine di riga), l'aritmetica dei puntatori permette varie combinazioni: nella portata della dichiarazione `int z[10][10]`, le seguenti espressioni sono tutte equivalenti: `z[i][j]`, `(*(z+i))[j]`, `*(z[i]+j)`, `*(*(z+i)+j)`.

L'interoperabilità tra array e puntatori è essenziale al momento del passaggio dei parametri. Si ricordi che C ha il solo passaggio per valore. Ma quando passa un vettore, passa sempre (per valore) un puntatore, mai l'array stesso. Il parametro formale può essere dichiarato sia come un array (`int A[]`) che come un puntatore (`int *A`). Nel caso di array multidimensionali, occorre specificare nel formale il numero di elementi delle dimensioni diverse dalla prima (per esempio `A[] [10]` oppure `(*A) [10]`): l'informazione è necessaria per generare staticamente il codice corretto per accedere agli elementi del vettore.

C permette, infine, la memorizzazione di array multidimensionali come array di puntatori ad array; per questa organizzazione a righe di puntatori (*row-pointer*) si veda l'Esercizio 2.

```
p = p+1;
c = c+1;
```

Se le due `malloc` restituiscono gli indirizzi *i* per `p` e *j* per `c`, al termine del frammento il valore dei due puntatori *non* è *i*+1 e *j*+1, bensì *i*+`sizeof(int)` e *j*+`sizeof(char)`. E così analogamente per tutte le altre operazioni di incremento e decremento.

Dovrebbe essere evidente che l'aritmetica dei puntatori distrugge qualsiasi ambizione alla type safeness di un linguaggio: non c'è alcuna garanzia che, in un generico momento dell'esecuzione di un programma, una variabile dichiarata come puntatore ad un oggetto di tipo `T`, punti ancora ad un'area di memoria in cui è memorizzato un valore di tipo `T`.

**Deallocazione** Abbiamo visto che uno dei metodi più comuni per creare un valore di tipo puntatore è quello di sfruttare opportune funzioni del linguaggio che contemporaneamente allocano un oggetto sullo heap e restituiscono un puntatore ad esso. La deallocazione della memoria può essere sia esplicita che implicita.

Nel caso di deallocazione implicita, il linguaggio non fornisce al programmatore alcuno strumento per deallocare memoria; il programmatore continua a chiedere allocazioni, fin quando vi è disponibilità sullo heap. Quando la memoria sullo heap è finita, tuttavia, non è detto che la computazione debba abortire. Infatti, alcuni valori precedentemente allocati potrebbero non avere più alcun cammino d'accesso per il loro uso. Nel seguente esempio

```
int* p = (int *) malloc(sizeof(int));
*p = 5;
p = null;
```

l'ultima istruzione, assegnando `null` a `p`, distrugge l'unico puntatore che permette di raggiungere l'area precedentemente allocata. È possibile dotare la macchina astratta di un meccanismo di recupero di tali porzioni di memoria; queste tecniche, che vanno sotto il nome di *garbage collection* (raccolta della spazzatura), sono un argomento molto studiato nell'ambito delle implementazioni dei linguaggi di programmazione che affronteremo sinteticamente nel Paragrafo 10.11.

Nel caso di deallocazione esplicita, il linguaggio mette a disposizione un meccanismo con il quale il programmatore rilascia la memoria riferita da un puntatore. In C, ad esempio, se `p` punta ad un oggetto sullo heap precedentemente creato con `malloc`, chiamando su `p` la funzione `free` viene deallocato l'oggetto puntato da `p` (che viene restituito alla lista libera secondo le tecniche già viste nel Paragrafo 7.4), ed è buona pratica assegnare contemporaneamente il valore `null` a `p`. È un errore semantico dal risultato impredicibile invocare `free` su un puntatore che non si riferisce ad un oggetto allocato con `malloc` (il che può accadere sia perché si è usata l'aritmetica dei puntatori, sia perché il puntatore era stato ottenuto con l'operatore `&`)<sup>11</sup>. Come abbiamo già osservato nel Capitolo 6, il problema più rilevante che si presenta con la deallocazione esplicita è che si possono generare in tal modo dei *dangling reference* (riferimenti pendenti), cioè puntatori con valore diverso da `null` che puntano ad informazioni non più significative. L'esempio più semplice è probabilmente il seguente

```
int* p;
int* q;
p = (int *) malloc(sizeof(int));
*p = 5;
q = p;
free(p);
p = null;
//
```

<sup>11</sup>Il punto è che `malloc` non alloca solo lo spazio necessario all'oggetto richiesto, ma anche un descrittore di tale dato, che contiene la dimensione del blocco allocato ed eventualmente altre informazioni. Accedendo a tali dati che `free` (che prende come parametro solo un puntatore) è in grado di determinare cosa deve essere deallocated.



Figura 10.6 Dangling reference.

```
... // serie di comandi senza modifica di p o q
...
// stampa (*p); // errore rilevabile dalla macchina astratta
stamp(a); // errore non rilevabile
```

La Figura 10.6 illustra la situazione della memoria e dei puntatori subito dopo la linea 5 e dopo la linea 7: il puntatore `q` ha un valore diverso da `null`, ma punta ad un'area di memoria che potrebbe essere ormai allocata per altri dati. Possiamo senz'altro supporre che la macchina astratta controlli e rilevi come errore ogni dereferenziazione di un puntatore `null` (come quello alla linea 11)<sup>12</sup>, mentre la dereferenziazione di `q` della linea 12 non verrà segnalata e può essere causa di errori tanto devastanti quanto difficili da localizzare.

In linguaggi che permettono ai puntatori di riferirsi ad oggetti sulla pila, i *dangling reference* si possono generare anche senza deallocazione esplicita. È sufficiente memorizzare in un ambiente non locale l'indirizzo di una variabile locale ad una funzione:

```
int* p;
void foo() {
    int n;
    p = &n;
}
foo();
... // qui p è un dangling reference
}
```

Se un linguaggio permette il verificarsi di *dangling reference*, è ovvio che non può essere type safe. Qualora il linguaggio non permetta puntatori a dati sulla pila, le *dangling reference* possono essere evitate impedendo la deallocazione esplicita, vuoi a livello di progetto del linguaggio (e dunque spostandosi su un linguaggio con deallocazione implicita dei puntatori e *garbage collection*), vuoi non deallocando niente anche quando il programmatore lo richiede: è il caso di alcune delle prime implementazioni di Pascal, dove la funzione di deallocazione `dispose` era implementata come una funzione con corpo vuoto.

<sup>12</sup>Ciò può avvenire spesso senza aggravio per l'efficienza, sfruttando i meccanismi di protezione degli indirizzi della macchina fisica sottostante.

```
{2, {33, {1, {4, {3, {1, {21, null}}}}}}}}
```

**Figura 10.7** Un valore di tipo int\_list.

Se tuttavia un linguaggio mantiene la deallocazione esplicita, o se ammette puntatori verso la pila, il Paragrafo 10.10 discuterà alcune tecniche (per la gestione dinamica dei puntatori) mediante le quali si possono disinnescare i dangling reference e dunque ripristinare la type safeness.

#### 10.4.6 Tipi ricorsivi

Un tipo ricorsivo è un tipo composto nel quale un valore del tipo può contenere un (riferimento a un) valore dello stesso tipo. In molti linguaggi li possiamo assimilare a (ed in effetti sono) record nei quali un campo può essere dello stesso tipo del record stesso. L'esempio più semplice è forse quello di una lista di interi, che presentiamo nel nostro pseudolinguaggio adattando la notazione da Java:

```
type int_list = {int val;
                 int_list next;};
```

Per terminare la ricorsione, il linguaggio può fornire un valore speciale, per esempio `null`, che appartiene a qualsiasi tipo (ricorsivo) e che possiamo immaginare come l'assenza di valori. Un valore di tipo `int_list` è dunque una coppia: il primo elemento è un intero, il secondo è a sua volta un valore di tipo `int_list`, e così via, fino a quando uno di questi valori `int_list` non sia `null` (si veda la Figura 10.7 per un esempio).

La Figura 10.8 riporta la definizione di un tipo ricorsivo per alberi binari di caratteri, insieme all'esempio di un valore e alla sua usuale rappresentazione grafica.

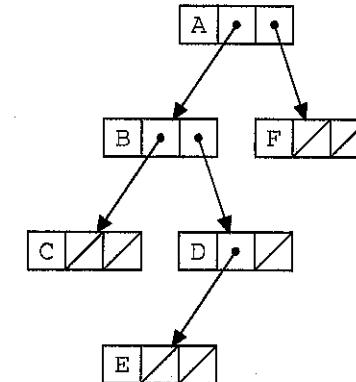
Come si vede da questi semplicissimi esempi la possibilità di definire tipi ricorsivi rende il sistema di tipi di un linguaggio flessibile e potente (si veda per esempio l'Esercizio 4). Le operazionimesse sui valori di tipo ricorsivo sono la selezione di una componente e il test di uguaglianza sul valore comune `null`.

Se il linguaggio ammette variabili modificabili (il che avviene in ogni linguaggio imperativo), è possibile costruire anche valori ciclici, mentre in un linguaggio puramente funzionale i valori di un tipo ricorsivo sono sempre strutture ad albero.

I tipi ricorsivi sono rappresentati di solito come strutture dati sullo heap. Un valore di un tipo ricorsivo corrisponde ad una struttura concatenata: ogni elemento della struttura è costituito da un record, nel quale il riferimento ricorsivo è implementato come un indirizzo al prossimo record (quello del passo ricorsivo precedente). La rappresentazione sullo heap dell'albero binario di Figura 10.8 corrisponde esattamente alla sua rappresentazione grafica...

```
type char_tree = {char val;
                  char_tree left;
                  char_tree right;};

{A, {B, {C, null, null}, {D, {E, null, null}, null}}, {F, null, null}}
```



**Figura 10.8** Alberi binari su char.

La creazione di valori di un tipo ricorsivo (così da poterli legare ad un nome) è un'operazione che varia sensibilmente da linguaggio a linguaggio. In molti linguaggi di tipo funzionale i valori di un tipo ricorsivo sono esprimibili: esistono esplicativi costrutti sintattici che permettono di designare un valore; un esempio potrebbe essere costituito dall'espressione della Figura 10.7, o da quella della Figura 10.8 (si veda anche il riquadro su ML). Nei linguaggi imperativi, invece, i valori di tipo ricorsivo vengono costruiti mediante allocazione esplicita dei loro componenti sullo heap. Per continuare la nostra analogia con Java, abbiamo un costruttore predefinito `new` che permette di allocare sullo heap un'istanza del tipo:

```
int_list l = new int_list();
```

La parte a destra dell'uguale crea sullo heap un record della forma specificata da `int_list` e assegna al nome `l` un riferimento a questo record. A questo punto possono essere inizializzati i suoi campi, per esempio

```
l.val = 2;
l.next = null;
```

Rimane da discutere della gestione dello heap in presenza di valori di tipo ricorsivo. Questi sono infatti dinamicamente allocati (in modo esplicito, come in Java, o in modo implicito, come in ML), ma i linguaggi che ammettono tipi ricorsivi primitivi non permettono in genere la deallocazione esplicita dei valori così creati. Come nel caso della deallocazione implicita che abbiamo discusso nel contesto dei puntatori, bisogna dotare la macchina astratta di meccanismi di

## Tipi ricorsivi in ML

ML è un importante linguaggio funzionale: dotato di un esteso sistema di tipi, è un linguaggio type safe che presenta molte interessanti caratteristiche. Il linguaggio permette la definizione esplicita di tipi ricorsivi mediante *costruttori*, introdotti nella definizione del tipo. A titolo d'esempio, la nostra definizione di lista di interi potrebbe essere scritta come (tralasciando il fatto che ML ha le liste come tipo predefinito):

```
datatype int_list = Null | Cons of int * int_list;
```

In questa definizione Null e Cons sono costruttori, cioè simboli di funzione a cui non corrisponde alcun codice, ma che servono solo a costruire sintatticamente termini del tipo voluto (in questo caso int\_list). La barra verticale deve essere letta come "oppure": un valore di tipo int\_list è la costante Null, oppure è una coppia (contraddistinta dal costruttore Cons) composta da un int ed una int\_list. Dunque, Cons(2, Null) è un semplice valore di tipo int\_list, mentre la Figura 10.9 riporta l'espressione ML che corrisponde al valore della Figura 10.7.

Il linguaggio permette la definizione di tipi ricorsivi malfondati. Ad esempio la seguente definizione introduce il tipo vuoto, del quale non è possibile costruire valori:

```
datatype vuoto = Void of vuoto;
```

Non vi è infatti un costruttore base che permetta di costruire un valore di vuoto senza presupporne un altro.

garbage collection, per recuperare le porzioni di memoria allocate ma senza più cammini d'accesso.

### 10.4.7 Funzioni

Tutti i linguaggi di alto livello permettono di definire funzioni (o procedure), ma pochi tra i linguaggi convenzionali permettono di denotare il tipo delle funzioni (cioè dargli un nome nel linguaggio). Se f è una funzione definita come

```
T f(S s){...}
```

possiamo indicare il suo tipo come  $S \rightarrow T$  e, più in generale, una funzione con intestazione

```
T f(S1 s1, ..., Sn sn){...}.
```

ha tipo  $S_1 \times \dots \times S_n \rightarrow T$ .

I valori di un tipo funzione sono dunque denotabili in tutti i linguaggi, ma raramente esprimibili (o memorizzabili). Oltre alla definizione, l'operazione principale ammessa su un valore di tipo funzione è l'*applicazione*, cioè l'invocazione di una funzione su alcuni argomenti (parametri attuali).

```
CONS (2, CONS (33, CONS (1, CONS (4, CONS (3, CONS (1, CONS (21, Null)))))))
```

Figura 10.9 Un valore di tipo int\_list in ML.

In alcuni linguaggi, tuttavia, il tipo delle funzioni ha la stessa dignità degli altri tipi: è possibile passare funzioni come argomento ad un'altra funzione, e definire funzioni che restituiscono funzioni come risultato. Tali linguaggi di *ordine superiore* appartengono soprattutto al paradigma funzionale (giusto per nominarne alcuni: ML, Haskell, Scheme). In questi linguaggi, una funzione non è più solo "un pezzo di codice dotato di ambiente locale", ma deve essere gestita come un qualsiasi altro dato: vedremo nel Capitolo 13 lo schema di una possibile macchina astratta in grado di gestire questa situazione.

## 10.5 Equivalenza

Dopo aver analizzato i principali tipi che troviamo in un linguaggio di programmazione, è venuto il momento di discutere delle regole che governano la correttezza di un programma relativamente alla tipizzazione. La prima classe di queste regole si preoccupa di definire quando due tipi, formalmente diversi, sono da considerarsi intercambiabili, cioè non distinguibili all'interno di un programma. Le regole definiscono tra i tipi una relazione di equivalenza: se due tipi sono equivalenti, ogni espressione o valore di un tipo è anche un'espressione o valore dell'altro e viceversa. Discuteremo nel prossimo paragrafo le regole di compatibilità, che specificano quando un valore di un tipo possa essere usato in un contesto nel quale sarebbe richiesto un valore di un altro tipo (ma non vale il viceversa).

Rammentiamo che un tipo, pur con le dovute differenze tra i linguaggi, viene definito con una definizione della forma seguente

```
type nuovotipo = espressione;
```

I vari linguaggi interpretano una definizione siffatta in due modi diversi, che danno luogo a due distinte regole di equivalenza tra tipi:

- la definizione di tipo può essere *opaca*, nel qual caso dà luogo all'equivalenza *per nome*;
- la definizione di tipo può essere *trasparente*, nel qual caso dà luogo all'equivalenza *strutturele*.

### 10.5.1 Equivalenza per nome

Se un linguaggio adotta definizioni di tipo opache, ogni nuova definizione introduce un nuovo tipo, diverso da tutti i precedenti. In questo caso dunque si ha equivalenza per nome, come definita qui sotto.

### Relazioni di preordine e di equivalenza

Una relazione binaria  $\star$  su un insieme  $D$  è un sottinsieme del prodotto cartesiano di  $D$  per se stesso:  $\star \subseteq D \times D$ . Scriviamo  $c \star d$  per  $(c, d) \in \star$ . Una relazione è *riflessiva*, quando, per ogni  $d \in D$ , vale  $d \star d$ ; è *simmetrica*, se per ogni  $c, d \in D$ , se  $c \star d$  allora vale anche  $d \star c$ ; è *transitiva*, se per ogni  $c, d, e \in D$ , se  $c \star d$  e  $d \star e$  allora anche  $c \star e$ ; è *antisimmetrica*, se per ogni  $c, d \in D$ , se  $c \star d$  e  $d \star c$  allora  $c$  coincide con  $d$ .

Se una relazione è riflessiva e transitiva, si dice che è un *preordine*. Se un preordine è anche simmetrico, allora è un'equivalenza. Se invece un preordine è anche antisimmetrico, allora è un *ordine parziale*.

**Definizione 10.2 (Equivalenza per nome)** Due tipi sono equivalenti per nome solo se hanno lo stesso nome (cioè un tipo è equivalente solo a se stesso).

Per fare un esempio, le seguenti definizioni di tipo

```
type T1 = 1..10;
type T2 = 1..10;
type T3 = int;
type T4 = int;
```

introducono quattro tipi distinti e non vi sono equivalenze tra essi.

Talvolta l'equivalenza per nome è troppo vincolante: alcuni linguaggi (Pascal per esempio) adottano un'equivalenza per nome *debole* (o lasca): una semplice ridenominazione di un tipo non genera un nuovo tipo, ma solo un alias per lo stesso tipo. Nell'esempio di sopra, in equivalenza per nome debole i tipi T3 e T4 sono equivalenti, mentre rimangono distinti i tipi T1 e T2.

Nell'equivalenza per nome, ogni tipo ha un'unica definizione in un unico punto del programma, il che semplifica la manutenzione. A dispetto della sua semplicità, da un punto di vista pragmatico l'equivalenza per nome è la scelta che più rispetta l'intenzione del progettista. Se il programmatore ha introdotto due nomi distinti per lo stesso tipo, avrà avuto le sue motivazioni, che il linguaggio rispetta mantenendo i tipi distinti.

Si osservi, tuttavia, che l'equivalenza per nome è definita in riferimento ad uno specifico, fissato programma e non ha senso chiedersi "in generale" se due tipi siano equivalenti.

### 10.5.2 Equivalenza strutturale

Una definizione di tipo è trasparente quando il nome del tipo non è che un'abbreviazione del tipo che viene definito. In un linguaggio con dichiarazioni trasparenti, due tipi sono equivalenti se hanno la stessa struttura, cioè se, sostituendo tutti i nomi con le relative definizioni, si ottengono tipi identici. Se un linguaggio adotta definizioni di tipo trasparenti, l'equivalenza tra tipi che si ottiene si dice equivalenza strutturale.

Possiamo dare una definizione più precisa dell'equivalenza strutturale.

### Equivalenza per nome in Pascal

Un aspetto non sempre gradevole dell'equivalenza per nome è che due espressioni di tipo "anonime" corrispondono a due tipi distinti (perché non avendo nomi, non hanno neppure lo stesso nome!). Nelle due dichiarazioni seguenti in Pascal:

```
var V : array [1..10] of integer;
W : array [1..10] of integer;
```

v e w non hanno lo stesso tipo. La situazione è particolarmente pesante quando si dichiara un parametro formale di una procedura mediante un'espressione di tipo anonimo:

```
procedure f (Z : array [1..10] of integer);
```

In questo esempio né v né w possono essere passati come parametri attuali a f, perché il parametro formale z è di un (terzo) tipo distinto dai due precedenti.

**Definizione 10.3 (Equivalenza strutturale)** L'equivalenza strutturale fra tipi è la (minima) relazione d'equivalenza che soddisfa le seguenti tre proprietà:

- un nome di tipo è equivalente a se stesso;
- se un tipo T è introdotto con una definizione type T = espressione, T è equivalente a espressione;
- se due tipi sono costruiti applicando lo stesso costruttore di tipo a tipi equivalenti, allora essi sono equivalenti.

A titolo d'esempio, si considerino le seguenti definizioni:

```
type T1 = int;
type T2 = char;
type T3 = struct{
    T1 a;
    T2 b;
}
type T4 = struct{
    int a;
    char b;
}
```

T3 e T4 sono strutturalmente equivalenti. Alcuni aspetti della definizione di equivalenza strutturale che abbiamo dato sono volutamente vaghi o imprecisi. Ad esempio, non è chiaro se i seguenti tre tipi siano equivalenti:

```
type S = struct{
    int a;
    int b;
}
type T = struct{
    int n;
    int m;
}
```

```
type U = struct{
    int m;
    int n;
}
```

dal momento che risultano dall'applicazione degli stessi costruttori, ma si sono scelti nomi (o ordine dei campi) diversi. Un'ulteriore questione delicata riguarda l'equivalenza strutturale di tipi ricorsivi:

```
type R1 = struct{
    int a;
    R2 p;
}
type R2 = struct{
    int a;
    R1 p;
}
```

Intuitivamente possiamo pensare che R1 e R2 siano equivalenti, ma il controllore dei tipi non riesce a risolvere la mutua ricorsione coinvolta in queste definizioni (per dimostrare l'equivalenza tra R1 e R2 occorre un argomento matematico abbastanza sofisticato).

L'equivalenza strutturale permette di parlare di tipi equivalenti in generale, e non con riferimento ad uno specifico programma. In particolare, due tipi equivalenti possono sempre essere sostituiti l'uno al posto dell'altro in un qualsiasi contesto senza alterare il significato del programma in cui avviene la sostituzione (questa proprietà generale della sostituibilità viene spesso indicata come "trasparenza referenziale").

Non sarà una sorpresa per il lettore sapere che i linguaggi esistenti adottano quasi sempre qualche forma di combinazione o variante delle due regole di equivalenza che abbiamo definito. Abbiamo già detto che Pascal adotta l'equivalenza per nome (debole); Java adotta l'equivalenza per nome, eccetto che per gli array, per i quali adotta un'equivalenza strutturale; C adotta equivalenza strutturale per gli array e i tipi definiti con `typedef`, ma non quando coinvolgono record (`struct`) e unioni, per i quali adotta sempre l'equivalenza per nome; C++ adotta equivalenza per nome (salvo per ciò che eredita da C...); ML adotta equivalenza strutturale, eccetto per i tipi definiti con `datatype`; Modula-2 eredita da Pascal l'equivalenza per nome, ma Modula-3 usa equivalenza strutturale; e così via.

## 10.6 Compatibilità e conversione

La relazione di compatibilità tra tipi, più debole dell'equivalenza, permette di usare un tipo in un contesto nel quale sarebbe richiesto un altro tipo.

**Definizione 10.4 (Compatibilità)** *Diciamo che il tipo T è compatibile con il tipo S, se un valore di tipo T è ammesso in un qualsiasi contesto in cui sarebbe richiesto un valore di tipo S.*

In molti linguaggi è la compatibilità (e non l'equivalenza) a regolare la correttezza di un assegnamento (il tipo del membro destro deve essere compatibile col tipo del membro sinistro), quella del passaggio dei parametri (il tipo del parametro attuale deve essere compatibile con quello del parametro formale) ecc. È chiaro che due tipi equivalenti sono anche compatibili l'uno con l'altro, ma in generale la relazione di compatibilità non è simmetrica: l'esempio canonico è la compatibilità esistente in molti linguaggi (ma non tutti: ML e Java ad esempio non l'ammettono) tra `int` e `float` (ma non viceversa). La relazione di compatibilità è dunque un preordine (cioè una relazione riflessiva e transitiva) non simmetrico, ma quasi mai un ordine (cioè non è simmetrico, ma neppure antisimmetrico): in un linguaggio che ammetta una qualche forma di equivalenza strutturale, due tipi strutturalmente equivalenti sono compatibili, ma non uguali.

La relazione di compatibilità, più ancora di quella di equivalenza, varia moltissimo da linguaggio a linguaggio e non è facile indicare delle linee comuni. Senza pretesa di essere completi, elenchiamo alcune delle possibili nozioni che possiamo incontrare, in ordine crescente di generalità (cioè di "generosità" della relazione di compatibilità); un tipo T può essere compatibile con S quando:

1. T e S sono equivalenti: è la nozione più restrittiva di compatibilità.
2. i valori di T sono un sottinsieme dei valori di S: è il caso di un tipo intervallo, contenuto nel suo tipo base (e quindi compatibile con esso).
3. tutte le operazioni sui valori di S sono possibili anche sui valori di T. L'esempio più semplice (che però non corrisponde alla compatibilità di nessuno dei principali linguaggi) è quello di due tipi record: si supponga di aver dichiarato

```
type S = struct{
    int a;
}
type T = struct{
    int a;
    char b;
}
```

L'unica operazione possibile sui valori di tipo S è la selezione del campo di nome a, che ha senso anche sui valori di tipo T. Si osservi che non c'è relazione di inclusione tra i valori di T e quelli di S, ma c'è un modo canonico per ottenere un valore di S a partire da un valore di T: prendi la prima componente e scorda la seconda. Questa interpretazione della compatibilità dà luogo a quella particolare nozione di compatibilità costituita dalla nozione di *sottotipo* per i linguaggi orientati agli oggetti, che discuteremo più avanti nel Paragrafo 12.2.4.

4. i valori di T corrispondono *in modo canonico* ad alcuni valori di S: oltre alla situazione del punto precedente, è il caso, già citato, di `int` compatibile con `float` (si osservi che questo caso non rientra nel secondo punto, perché nell'*implementazione* i valori dei due tipi sono distinti).
5. i valori di T possono essere fatti corrispondere ad alcuni valori di S: fatta cadere la richiesta di canonicità della corrispondenza, ogni tipo può essere reso compatibile con un altro definendo un modo (convenzionale) per trasformare un valore di T in uno di S. Con questa nozione amplissima di compatibilità, `float`

può essere compatibile con `int`, definendo in modo arbitrario la procedura di conversione (per esempio arrotondamento, troncamento ecc.)

Per gestire in un modo unificato questa congerie di interpretazioni diverse, si introduce la nozione di *conversione di tipo*, che viene declinata in due modi distinti:

- *conversione implicita* (detta anche *coercione*, *coercion*, o conversione forzata), quando è la macchina astratta ad inserire tale conversione, senza che ve ne sia traccia a livello linguistico;
- *conversione esplicita*, o *cast*, quando la conversione è indicata nel testo del programma.

**Coercizioni** In presenza di compatibilità tra il tipo `T` ed il tipo `S`, il linguaggio permette la presenza di un valore di `T` laddove sarebbe richiesto un valore di `S`. In corrispondenza di ogni situazione del genere, il compilatore e/o la macchina astratta inseriscono una conversione implicita di tipo da `T` a `S`, che chiamiamo coercione di tipo (*coercion*).

Da un punto di vista sintattico, la coercione non ha altro significato che quello di annotare la presenza di una situazione di compatibilità. Da un punto di vista implementativo, invece, una coercione può corrispondere a cose distinte, a seconda della nozione di compatibilità adottata.

1. `T` è compatibile con `S` e condividono la rappresentazione in memoria (almeno sui valori di tipo `T`): in tal caso la coercione rimane al livello sintattico e non genera alcun codice.
2. `T` è compatibile con `S` perché esiste un modo canonico di trasformare i valori di `T` in `S`: in questo caso la coercione viene eseguita dalla macchina astratta, che applica appunto la conversione canonica. Ad esempio, nel caso di `int` e `float`, la macchina astratta inserisce codice per trasformare (a run-time) la rappresentazione in complemento a due in quella in virgola mobile.
3. la compatibilità di `T` con `S` è stabilita in virtù di una corrispondenza arbitraria tra i valori di `T` e quelli di `S`: anche in questo caso la macchina astratta inserisce il codice che corrisponde alla trasformazione. Rientrano in questo caso anche tutte le situazioni nelle quali `T` ed `S` hanno la stessa rappresentazione, ma `T` è un sovrainsieme di `S`: ad esempio `T` è `int` e `S` è un intervallo di interi<sup>13</sup>(si noti che siamo nella situazione simmetrica al caso “canonico” trattato al punto 1, in cui  $T \subseteq S$ ). La coercione in quest’ultimo caso non trasforma la rappresentazione, ma controlla dinamicamente (almeno se il linguaggio vuole essere type safe) che il valore di `T` appartenga a `S`.

Linguaggi con forti controlli di tipo tendono ad avere poche coercizioni (cioè poche compatibilità) dell’ultima specie che abbiamo menzionato; al contrario,

<sup>13</sup>Quasi tutti i linguaggi che ammettono intervalli ammettono anche questo tipo di compatibilità, per ovvi motivi di flessibilità di programmazione.

### Cast che non modificano la rappresentazione

In certe situazioni, specialmente nella programmazione di sistema, è utile poter cambiare il tipo di un oggetto senza cambiarne la rappresentazione in memoria. Questo tipo di conversioni sono ovviamente vietate in ogni linguaggio type safe, ma sono invece possibili in molti linguaggi (sono dette spesso conversioni “unchecked” (ADA), o “non-converting type cast”).

Un esempio in C attraverso l’operatore “indirizzo-di” (`&`) e quello di dereferenziazione (`*`):

```
int a = 233;
float b = *(float*) &a;
```

Per prima cosa si prende l’indirizzo della variabile intera `a`, lo si converte esplicitamente in un puntatore ad un `float` mediante il cast (`float*`), infine si de-referenzia questo puntatore e lo si assegna alla variabile reale `b`. Il tutto è solo annotazione per il compilatore, che non si traduce in nessuna azione della macchina astratta: i bit che in complemento a due rappresentavano 233, ora sono interpretati come una rappresentazione IEEE 754 di un numero in virgola mobile (a patto che interi e `float` siano rappresentati con lo stesso numero di byte...).

È chiaro che una conversione che non modifica la rappresentazione è sempre un’operazione estremamente delicata, nonché potenziale fonte di errori di difficile eliminazione.

un linguaggio come C, il cui sistema di tipi è stato progettato per poter essere aggirato, permette innumerevoli coercizioni (da caratteri a interi, da reali lunghi a corti, da interi lunghi a reali corti ecc.).

**Conversioni esplicite** Le conversioni esplicite (o *cast*, dal nome con cui sono indicate in C e altri linguaggi) sono annotazioni *nel linguaggio* che specificano che un valore di un tipo deve essere convertito in un altro tipo. Anche in questo caso tale conversione può essere solo un’indicazione sintattica o può corrispondere a codice eseguito dalla macchina astratta, secondo la casistica che abbiamo già fatto per le coercizioni. Nel nostro pseudolinguaggio, prendendo la notazione dalla famiglia di C, indicheremo un cast con le parentesi:

```
S s = (S) t;
```

dove stiamo assegnando ad una variabile di tipo `S` il valore `t`, dopo averlo convertito nel tipo `S`. Non ogni conversione esplicita è consentita, ma solo quelle per le quali il linguaggio conosce come implementare la conversione. È evidente che si può sempre inserire un cast laddove esiste una compatibilità: la conversione esplicita è inutile sintatticamente, ma può essere consigliata per motivi di documentazione. I linguaggi con poche compatibilità rendono disponibili, in genere, molte conversioni esplicite, con le quali il programmatore annota le situazioni dove è necessario un cambio di tipo.

In linea generale, i linguaggi moderni tendono a favorire i cast rispetto alle coercizioni: sono più espressivi da un punto di vista della documentazione, non dipendono dal contesto sintattico in cui compaiono, e, soprattutto, si comportano meglio in presenza di overloading e polimorfismo, due questioni che tratteremo tra poco.

## 10.7 Polimorfismo

Un sistema di tipi nel quale ogni oggetto del linguaggio (valore, funzione ecc.) ha un unico tipo, si dice *monomorfo*<sup>14</sup>. Qui ci interessa la nozione più generale, data dalla seguente definizione.

**Definizione 10.5** *Un sistema di tipi nel quale uno stesso oggetto può avere più di un tipo è detto polimorfo*<sup>15</sup>. Per analogia, diremo che un oggetto è polimorfo quando il sistema di tipi gli assegna più di un tipo.

Anche nei linguaggi più convenzionali vi sono alcune limitate (o in qualche caso apparenti) forme di polimorfismo: il nome + ha in molti linguaggi sia il tipo `int × int → int` che il tipo `float × float → float`; il valore `null` ha tipo `T*` per ogni tipo `T`; la funzione `length` (che restituisce il numero di elementi di un array) ha tipo `T[] → int` per ogni tipo `T`; e l'elenco potrebbe continuare. Nei linguaggi convenzionali, tuttavia, non è in genere permesso all'utente di definire oggetti polimorfi. Prendiamo un linguaggio con un sistema di tipi abbastanza rigido (per esempio Pascal o Java, ma in qualche misura anche C) e supponiamo di voler scrivere una funzione che ordini un vettore di interi. La funzione risultante potrebbe avere l'intestazione

```
void int_sort(int A[])
```

Se adesso abbiamo la necessità di ordinare un vettore di caratteri, dovremo definire un'altra funzione

```
void char_sort (char C[])
```

nella quale tutto è identico alla precedente, eccetto le annotazioni di tipo. Un linguaggio polimorfo ci permetterebbe di definire un'unica funzione

```
void sort(<T> A[])
```

dove `<T>` indica un tipo generico, che verrà specificato a tempo debito.

In questo paragrafo analizzeremo il fenomeno del polimorfismo, cominciando col distinguere tra due diverse forme:

- *polimorfismo ad hoc*, detto anche *overloading*;

<sup>14</sup>È una parola che deriva dal greco classico e significa "che ha una sola (*mono-*) forma (*-morphos*)".

<sup>15</sup>"Che ha tante (*poly-*) forme".

- *polimorfismo universale*, che distinguiamo ulteriormente in
  - *parametrico*;
  - *di sottotipo*, o *di inclusione*.

### 10.7.1 Overloading

L'overloading (sovraffollamento), come l'altro nome di polimorfismo *ad hoc* suggerisce, è in realtà polimorfismo solo in apparenza. Un nome è sovraccaricato (overloaded) quando ad esso corrispondono più oggetti, e viene usata l'informazione fornita dal contesto per decidere quale oggetto è denotato da una specifica istanza di quel nome. Gli esempi più comuni di questo fenomeno sono l'uso del nome `+` per indicare sia la somma intera che la somma reale (e talvolta anche la concatenazione tra caratteri) e la possibilità di definire più funzioni (o costruttori) con lo stesso nome, ma distinte dal numero o dal tipo dei parametri.

Nel caso di overloading dunque, ad uno stesso nome corrispondono più oggetti diversi (in numero finito): se si tratta di un nome di funzione, ad esso vengono associati codici distinti. La situazione di ambiguità viene risolta staticamente, utilizzando l'informazione di tipo presente nel contesto. Da un punto di vista concettuale, possiamo immaginare una sorta di pre-analisi del programma, che risolve i casi di overloading e sostituisce ogni simbolo sovraccaricato con un nome non ambiguo che denota univocamente un solo oggetto. L'overloading è dunque una sorta di abbreviazione sintattica che scompare non appena siano introdotte ulteriori informazioni.

Non si confonda l'overloading con le coercizioni: si tratta di due meccanismi tra loro del tutto indipendenti, che rispondono a problemi distinti. D'altra parte, in presenza di coercizioni, può non essere del tutto evidente come un caso di overloading debba essere risolto. Si considerino, per esempio, le seguenti espressioni:

```
1 + 2
1.0 + 2.0
1 + 2.0
1.0 + 2
```

Possiamo interpretare queste quattro espressioni in più modi diversi: `+` è sovraccaricato con quattro diversi significati; `+` è sovraccaricato con due significati (intero e reale) e sono inserite coercizioni nei due ultimi casi; `+` denota solo la somma reale e negli altri tre casi sono introdotte coercizioni. Sarà la definizione del linguaggio a stabilire quale sia l'interpretazione corretta.

### 10.7.2 Polimorfismo universale parametrico

Iniziamo con una definizione, al solito imprecisa ma sufficiente per i nostri scopi.

**Definizione 10.6** *Un valore esibisce polimorfismo universale parametrico (o polimorfismo parametrico, in breve) quando ha un'infinità di tipi diversi, che si ottengono per istanziazione da un unico schema di tipo generale.*

### Tipi polimorfi e quantificatori universali

Invece di scrivere un tipo polimorfo con le parentesi angolate, una notazione più uniforme (e più suggestiva, nonché matematicamente più accurata) è quella che fa uso di quantificatori universali: invece di  $\langle T \rangle [] \rightarrow void$  si scriverà  $\forall T. T [] \rightarrow void$ .

Si tratta di una notazione che si presta bene a descrivere tutte le varianti di polimorfismo che sono state proposte e che useremo nel Paragrafo 10.7.3 (e poi ancora in 12.4.1) per trattare del polimorfismo di sottotipo.

Una funzione polimorfa universale è dunque costituita da un unico codice, che lavora uniformemente su tutte le istanze del suo tipo generale (in sostanza perché le informazioni di tipo non sono utilizzate nell'algoritmo che la funzione implementa).

Tra gli esempi che abbiamo discusso all'inizio di questo paragrafo, ne troviamo due che ricadono in questa categoria: il valore `null` che appartiene *ad ogni* tipo  $T^*$  (cioè ad ogni tipo che si ottiene sostituendo a  $T$  un tipo vero e proprio); la funzione `void sort(<T> A[])` che ordina un array di un *qualunque* tipo.

Prima di fare ulteriori esempi, elaboriamo sulla notazione. Seguendo quanto suggerito nel Paragrafo 10.4.7, possiamo scrivere il tipo polimorfo di `sort` come  $\langle T \rangle [] \rightarrow void$ . In questa notazione, usiamo le parentesi angolate per indicare che  $\langle T \rangle$  non è un tipo vero e proprio, ma è una sorta di parametro: sostituendo ad esso *un qualsiasi* tipo “concreto” si ottiene uno specifico tipo per `sort`. Tutti i modi possibili di sostituire un tipo al posto di  $\langle T \rangle$  corrispondono agli infiniti modi di applicare tale funzione: `int [] → void`, `char [] → void` ecc. Con questa notazione, il tipo di `null` sarebbe più correttamente indicato con  $\langle T \rangle ^*$ .

Facciamo un altro esempio: una funzione che scambia due variabili di tipo qualsiasi. Come nel caso di `sort`, in un linguaggio senza polimorfismo avremo bisogno di una funzione `swap` per ogni possibile tipo delle variabili. Usando il polimorfismo universale e supponendo di avere a disposizione il passaggio per riferimento, potremo scrivere

```
void swap (reference <T> x, reference <T> y) {
    <T> tmp = x;
    x = y;
    y = tmp;
}
```

Un oggetto polimorfo può essere *istanziato* su uno specifico tipo; l'istanziazione può avvenire in molti modi diversi, a seconda dello specifico meccanismo di polimorfismo fornito da un certo linguaggio. La modalità più semplice è quella in cui l'istanziazione avviene automaticamente, per effetto del compilatore o della macchina astratta:

```
int* k = null;
char v,w;
int i,j;
...
```

### Template in C++

Il polimorfismo parametrico si ottiene in C++ con l'uso dei *template*: schemi di programma in cui compaiono parametri (di tipo o di classe). La funzione `swap` potrebbe assumere la forma seguente

```
template<typename T>
void swap (T& x, T& y) {
    T tmp = x;
    x = y;
    y = tmp;
}
```

dove l'operatore & postfisso al nome del tipo di un parametro formale indica che quel parametro è passato per riferimento, una modalità che C++ ha come primitiva, a differenza di C. L'istanziazione di un template è automatica: `swap(x, y)` viene istanziata sul tipo comune delle variabili `x` e `y`.

```
swap(v,w);
swap(i,j);
```

Senza bisogno di ulteriori annotazioni il controllore dei tipi istanzia alla linea 1 il tipo di `null` come `int*` sfruttando le informazioni di contesto (l'assegnamento viene fatto su una variabile di tipo `int*`); alla linea 5, `swap` viene istanziata sui caratteri; alla linea 6 sugli interi.

Il polimorfismo parametrico è molto generale e flessibile ed è presente nei linguaggi di programmazione in due forme notizionalmente distinte, che sono dette polimorfismo parametrico *esplicito* e polimorfismo parametrico *implicito*.

**Polimorfismo esplicito** Il polimorfismo esplicito è quello che abbiamo discusso sin qui: nel programma sono presenti esplicite annotazioni (i nostri  $\langle T \rangle$ ) che indicano quali tipi devono essere considerati parametri. È la forma di polimorfismo esistente in C++ (mediante la nozione di “template”) e Java (a partire dalla versione J2SE 5.0 del settembre 2004, mediante la nozione di “generics”).

**Polimorfismo implicito** Altri linguaggi (in particolare il linguaggio funzionale ML) adottano il polimorfismo parametrico implicito, nel quale il programma può non riportare alcuna indicazione di tipo, ed è invece il controllore dei tipi (anzi, più propriamente, il modulo che si occupa dell'inferenza dei tipi, Paragrafo 10.8) a cercare di ottenere, per ogni oggetto del linguaggio, il suo tipo più generale, dal quale tutti gli altri tipi si possano ottenere per istanziazione dei parametri di tipo. L'esempio più semplice di funzione polimorfa è quello dell'identità che, usando una notazione il più possibile vicina a quella che abbiamo usato sinora, può essere scritta come:

```
fun Ide(x) {return x;}
```

Usiamo la parola `fun` per indicare che stiamo procedendo alla definizione di una funzione; per il resto è tutto analogo alle definizioni che abbiamo visto, tranne che

non vi sono indicazioni di tipo, né per i parametri, né per il risultato. L'inferenza dei tipi assegnerà alla funzione `Ide` il tipo  $\langle T \rangle \rightarrow \langle T \rangle$ : se il parametro attuale è di tipo  $X$ , anche il risultato è di tipo  $X$ , per qualsiasi  $X$ . Anche l'applicazione di una funzione polimorfa avviene ovviamente senza indicazione di tipo: `Ide(3)` sarà correttamente tipizzata come un valore di tipo `int`, mentre `Ide(true)` otterrà tipo `bool`.

Concludiamo con un ulteriore esempio di polimorfismo parametrico implicito combinato con l'ordine superiore:

```
fun Comp(f,g,x){return f(g(x));}
```

`Comp` applica all'argomento  $x$  la composizione delle due funzioni passate come primi parametri. Qual è il tipo polimorfo che l'inferenza assegnerà a `Comp`? Si tratta di quello più generale possibile:

```
(\S\rightarrow\T) \times (\R\rightarrow\S) \times \R \rightarrow \T
```

Il tipo ottenuto dall'inferenza è forse più generale di quello che alcuni lettori potrebbero aver pensato...

### 10.7.3 Polimorfismo universale di sottotipo

Il polimorfismo di sottotipo, presente tipicamente nei linguaggi orientati agli oggetti, è una forma più limitata di polimorfismo universale rispetto al polimorfismo parametrico.

Anche in questo caso, infatti, uno stesso oggetto ha un'infinità di tipi diversi, che si ottengono per istanziazione da uno schema di tipo più generale. Ed anche in questo caso, trattandosi comunque sempre di polimorfismo "vero", cioè universale, c'è (almeno concettualmente) un solo algoritmo *uniforme nel tipo* (cioè che non dipende da qualche particolare struttura del tipo), che può venire istanziato sull'infinità di tipi possibili.

Tuttavia, nel caso del polimorfismo di sottotipo, non tutte le possibili istanziazioni dello schema di tipo più generale sono ammissibili, ma dobbiamo limitarci a quelle definite da una qualche nozione di compatibilità "strutture" tra tipi, e cioè dalla nozione di sottotipo.

Per essere più precisi, assumiamo che fra i tipi del linguaggio sia definita una relazione di sottotipo che indichiamo con il simbolo " $<:$ ", leggendo  $C <: D$  come " $C$  è un sottotipo di  $D$ ". Per ora ci accontentiamo di una nozione astratta, ma nel Paragrafo 12.2.4 sostanzieremo questo concetto in termini di relazioni fra classi nei linguaggi orientati agli oggetti.

**Definizione 10.7** Un valore esibisce polimorfismo di sottotipo (*o limitato*) quando ha un'infinità di tipi diversi, che si ottengono per istanziazione da uno schema di tipo generale, sostituendo ad un opportuno parametro i sottotipi di un tipo assegnato.

Per esprimere il polimorfismo di sottotipo è utile sfruttare la notazione con quantificatore universale che abbiamo introdotto nel riquadro di pag. 322. Una funzione polimorfa di tipo

### Polimorfismo implicito in ML

La maggior parte dei compilatori per ML sono sistemi interattivi: l'utente inserisce un'espressione (o una definizione) per volta. Il sistema in primo luogo verifica la correttezza dei tipi dell'espressione e deriva per essa il tipo più generale. Se i tipi sono corretti, l'espressione viene valutata e viene mostrato il risultato (nel caso di una definizione la sua valutazione coincide con l'estensione dell'ambiente con la nuova associazione). Possiamo definire la funzione identità con

```
- fun Ide(x) = x;
val Ide = fn : 'a -> 'a
```

In questo esempio, la prima linea è ciò che è stato inserito dall'utente (- è il prompt), mentre la linea seguente è la risposta del sistema: viene inserito nell'ambiente il nome `Ide` legato ad un valore funzionale di tipo `'a -> 'a`. Gli identificatori preceduti da ' `sono in ML variabili di tipo, cioè parametri che possono essere istanziati. Possiamo chiedere la valutazione di Ide:`

```
- Ide(4);
val it = 4 : int
- Ide(true);
val it = true : bool
```

Adesso le altre funzioni:

```
- fun Comp(f,g,x) = f(g(x));
val Comp = fn : ('a->'b)*('c->'a)*'c -> 'b
- fun swap(x,y) = let tmp = !x in
      (x:=!y; y=tmp);
val swap = fn : ('a ref)*('a ref) -> unit
```

Nel caso di `swap`, l'inferenza di tipo individua che i due argomenti devono essere non generici valori, ma *variabili* ("riferimenti") di un tipo qualsiasi (`'a ref`); `!` è l'operatore di dereferenziazione esplicita; `unit` è il tipo singoletto usato come tipo delle espressioni che hanno effetti collaterali.

```
- val v = ref 0;
val v = ref 0 : int ref
- val w = ref 2;
val w = ref 0 : int ref
- swap(v,w);
val it = () : unit
- !w;
val it = 0 : int
```

Nell'esempio `v` e `w` vengono inizializzate, rispettivamente, ad un riferimento a 0 e 2 e l'inferenza correttamente deduce che devono essere variabili di tipo `int`. Dopo lo scambio, la dereferenziazione di `w` dà ovviamente 0.

```
VT<:D.T-> void
```

può essere applicata a tutti i valori di un qualsiasi sottotipo di *D*. La situazione di polimorfismo dunque non è generale ma è *limitata ai sottotipi di D*.

#### 10.7.4 Cenni sull'implementazione

Non possiamo in questo testo trattare in dettaglio di come il polimorfismo universale possa essere implementato. Ci limiteremo a portare due esempi paradigmatici dei problemi che si possono incontrare.

Un primo modo di gestire il polimorfismo è quello di risolverlo in modo statico, a tempo di linking<sup>16</sup> (è questo il caso di C++). Quando una funzione polimorfa è chiamata con due istanze diverse (il che può avvenire anche da parte di moduli compilati separatamente), il suo codice viene istanziato in due modi distinti, per tenere conto delle diverse istanze con le quali è usata. Nel solito esempio di *swap*, c'è una variabile locale *tmp* che deve essere allocata (nel RdA di *swap* al momento della sua chiamata). Ma lo spazio da allocare per *tmp* dipende dal suo tipo, che non è noto al momento in cui il template viene originariamente compilato. A tempo di linking, pertanto, vengono identificate le funzioni polimorfe, il loro codice viene opportunamente modificato (istanziato) per tener conto del tipo con cui sono chiamate e il codice risultante viene collegato tutto assieme. Si osservi che esistono nel codice eseguibile più copie dello stesso template, una per ogni istanza. D'altra parte l'esecuzione di un programma che usa i template è efficiente quanto quella di un programma che non li usa, dato che i template non esistono più a tempo d'esecuzione.

Nel caso di altri linguaggi con polimorfismo (e ML è uno di questi), viene generata un'unica versione del codice di una funzione polimorfa, ed è proprio quell'unico codice che viene eseguito quando serve una sua istanza. Come fare però per le informazioni che dipendono dal tipo (nel caso di *swap*, con le dimensioni da allocare per *tmp*)? Occorre cambiare alla radice la rappresentazione dei dati: invece di allocare direttamente il dato, nell'RdA si mette un puntatore al dato stesso, che comprende anche un suo descrittore (dimensione, struttura ecc.). Nel caso di *swap*, quando si deve memorizzare un valore in *tmp*, accedendo tramite l'opportuno puntatore alla variabile *x*, si identificano (nel descrittore) le sue dimensioni; a questo punto si alloca sullo heap la memoria necessaria per *tmp* ed un puntatore ad essa viene messo nel RdA. La flessibilità, l'uniformità e la concisione del codice (non ci sono istanze ripetute della stessa funzione) sono pagate con un po' d'efficienza: occorre sempre un accesso indiretto per riferirsi ad un dato. Nel caso del polimorfismo di sottotipo, vedremo ulteriori dettagli implementativi per il caso dei linguaggi orientati agli oggetti nel Paragrafo 12.4.

<sup>16</sup>Qualche autore ritiene che in un caso del genere si dovrebbe parlare di *generici* invece che di polimorfismo. Noi pensiamo, invece, che il fenomeno del polimorfismo sia un fenomeno sintattico di un linguaggio, che può essere ottenuto con svariate implementazioni diverse.

## 10.8 Controllo e inferenza di tipo

Abbiamo già osservato più volte come sia responsabilità del *controllore dei tipi* (*type checker*) di un linguaggio il verificare che un programma rispetti le regole imposte dal sistema di tipi (in particolare la compatibilità). Nel caso di un linguaggio con controlli statici, il controllore è un modulo del compilatore; nel caso di controlli dinamici, il controllore è un modulo del supporto a tempo d'esecuzione. Per ottenere il suo scopo il controllore deve determinare il tipo delle espressioni presenti nel programma (ad esempio, in un assegnamento, deve determinare il tipo dell'espressione che ne costituisce il membro destro, in modo da verificare la sua compatibilità con il tipo della variabile a membro sinistro), sfruttando le informazioni di tipo che il programmatore ha esplicitamente inserito in alcuni punti critici (per esempio le dichiarazioni dei nomi locali, le dichiarazioni dei parametri, le conversioni esplicite ecc.) e quelle implicite nella definizione del programma (per esempio i tipi delle costanti predefinite, quelli delle costanti numeriche ecc.).

Per determinare il tipo delle espressioni complesse, il controllore esegue una semplice visita dell'albero sintattico (si ricordi l'inizio del Paragrafo 8.1.3) del programma: a partire dalle foglie (variabili e costanti di cui si conosce il tipo), risale l'albero verso la radice, calcolando il tipo delle espressioni composte a partire dalle informazioni fornite dal programmatore e da quelle che derivano dal sistema di tipi (ad esempio, il sistema di tipi potrebbe stabilire che *+* è un operatore che, applicato a due espressioni di tipo *int*, permette di ottenere un'espressione anch'essa di tipo *int*, mentre *=*, se applicato a due espressioni dello stesso tipo scalare, dà un'espressione di tipo *bool*). In molti casi l'informazione fornita dal programmatore può risultare ridondante:

```
int f(int n){return n+1;}
```

A partire dall'informazione che *n* ha tipo *int*, si inferisce in modo semplice che il valore restituito dalla funzione è un *int*. La specifica esplicita da parte del programmatore del tipo del risultato è una specifica ridondante che il linguaggio richiede al fine di rilevare eventuali errori logici.

Al posto del semplice controllo dei tipi che abbiamo appena delineato, alcuni linguaggi adottano un procedimento più sofisticato, che possiamo chiamare *inferenza di tipo* (cioè derivazione, deduzione di tipo). Il capostipite di questa famiglia di linguaggi è ML, che abbiamo più volte citato in precedenza, e il cui sistema di tipi è sofisticato e raffinato: alcune idee in esso presenti hanno ispirato il progetto di molti linguaggi diversi, anche non funzionali. Per introdurre in modo semplice il concetto di inferenza, riprendiamo la definizione di *f* che abbiamo appena dato: a ben guardare anche la specifica che il parametro formale *n* deve essere *int* è ridondante: la costante *1* è un intero, e *+* prende due interi e restituisce un intero, dunque *n* deve essere un intero. A partire da una dichiarazione della forma

```
fun f(n){return n+1;}
```

è certo possibile derivare automaticamente che *f* ha tipo *int->int*. L'inferenza di tipo è appunto questo processo di attribuzione di un tipo ad un'espressione nella quale non appaiono esplicite dichiarazioni di tipo per i suoi componenti.

Per condurre in porto questa derivazione si lavora ancora sull'albero sintattico, a partire dalle foglie, ma può capitare che in corrispondenza di qualche espressione atomica (*n* in questo caso) non sia possibile determinare subito un tipo specifico. L'algoritmo di inferenza assegna in tal caso una *variabile di tipo*, che possiamo indicare con '*a*', secondo l'uso di ML. Risalendo nell'albero e sfruttando le informazioni di tipo presenti nel contesto, vengono collezionati alcuni vincoli sulle variabili di tipo. Nel nostro caso, dal tipo di 1 e da quello di + (che risulta nella tabella dei simboli predefiniti insieme ai vari tipi delle sue versioni overloaded), si deriva il vincolo '*a* = int'.

Questo tipo di inferenza è molto più generale e potente del semplice controllo dei tipi usato nei linguaggi quali Pascal, C o Java: è in grado di scoprire il tipo più generale di una funzione, cioè di esplicitare tutto il polimorfismo implicito in un'espressione. Essa procede secondo l'algoritmo seguente.

1. Assegna un tipo ad ogni nodo dell'albero sintattico: per i nomi predefiniti e le costanti, usa il tipo indicato nella tabella dei simboli; per gli identificatori "nuovi" e per ogni espressione composta (i nodi interni dell'albero) usa una variabile di tipo (una nuova variabile per ogni espressione o nome).
2. Risali l'albero sintattico, generando un vincolo (di egualanza) tra tipi in corrispondenza ad ogni nodo interno; ad esempio, se applichiamo il simbolo di funzione *f* al quale avevamo precedentemente assegnato il tipo '*a*' all'argomento *v* di tipo '*b*', verrà generato il vincolo '*a* = '*b*->*c*', ad indicare che *f* deve essere davvero una funzione e che il suo argomento deve essere dello stesso tipo di *v* ('*c*' è una nuova variabile di tipo).
3. Risoli i vincoli così raccolti, usando l'algoritmo di *unificazione*, un potente (ma concettualmente semplice) strumento di manipolazione simbolica che discuteremo nel contesto dei linguaggi logici, nel Paragrafo 14.3.

Vi sono casi in cui la risoluzione dei vincoli non è in grado di rimuovere tutte le variabili. Se applichiamo l'algoritmo di inferenza a

```
fun g(n) {return n;}
```

otterremo come tipo '*a* -> '*a*'. Sappiamo già che questo non è un errore, ma una caratteristica positiva: un'espressione il cui tipo più generale contiene una variabile di tipo è un'espressione polimorfa.

## 10.9 Sicurezza: un bilancio

Abbiamo iniziato la nostra analisi dei tipi in un linguaggio di programmazione dalla nozione di sicuro relativamente ai tipi; questa stessa nozione ci ha guidato nella nostra disamina delle varie caratteristiche di un sistema di tipi. È venuto il momento di fare un breve bilancio di quello che abbiamo compreso, classificando i linguaggi di programmazione relativamente alla sicurezza. Possiamo distinguere tra:

1. linguaggi non sicuri;

2. linguaggi localmente non sicuri;
3. linguaggi sicuri.

Nella categoria dei linguaggi non sicuri troviamo tutti quelli il cui sistema di tipi è nella sostanza un suggerimento metodologico per il programmatore, nel senso che il linguaggio permette di aggirare o rilassare i controlli di tipo. Ogni linguaggio che permette di accedere alla rappresentazione di un tipo di dato appartiene a questa categoria, così come ogni linguaggio che permetta di accedere al valore di un puntatore (aritmetica dei puntatori). Sono dunque non sicuri C, C++ e i linguaggi della stessa famiglia.

Sono localmente non sicuri quei linguaggi nei quali il sistema di tipi è ben regolato e i tipi controllati, ma che contengono alcuni, limitati, costrutti che, qualora usati, permettono di scrivere programmi non sicuri. Fanno parte di questa categoria ALGOL, Pascal, Ada e molti loro discendenti, a patto che la macchina astratta controlli davvero i tipi anche per quel che riguarda i controlli dinamici, come i limiti dei tipi intervallo (che, lo ricordiamo, sono usati come tipo indice degli array). La locale non sicurezza discende dalla presenza di unioni (varianti non controllate) e dalla deallocazione esplicita di memoria. Di questi due costrutti, è il secondo ad avere il maggior impatto pratico (è molto più frequente incontrare programmi che deallocano memoria che programmi che fanno uso significativo di record varianti), ma è il primo ad essere in via di principio radicalmente più pericoloso. Come vedremo nel prossimo paragrafo, è possibile dotare la macchina astratta di opportuni meccanismi che consentono di controllare e impedire il fenomeno dei dangling reference, anche se, per motivi di efficienza, non sono quasi mai utilizzati, a meno di particolari richieste di correttezza. Abbiamo già visto diversi esempi di violazione del sistema di tipi che sono possibili sfruttando i record varianti. Abbiamo lasciato per ultimo il più radicale, nel quale i varianti sono usati per accedere e manipolare il valore di un puntatore. Scriviamo in Pascal<sup>17</sup>:

```
var v : record
  case bool of
    true : (i:integer);
    false : (p:^integer)
  end
```

dove ^integer è la notazione Pascal per il tipo dei puntatori a interi. È ora possibile assegnare un puntatore alla variante p, manipolarla come un intero usando la variante i e poi usarla di nuovo come un puntatore.

Infine, abbiamo i linguaggi sicuri, per i quali un teorema garantisce che l'esecuzione di un programma tipizzato non può mai generare un errore non rilevato "indotto dalla violazione di un tipo". Stanno in questa categoria sia linguaggi con controllo dinamico dei tipi, quali LISP e Scheme, sia linguaggi con controllo statico come ML, sia linguaggi con controllo statico ma con importanti controlli a tempo d'esecuzione, come Java.

<sup>17</sup>In Pascal i puntatori sono creati solo per mezzo di una richiesta di allocazione sullo heap (funzione new), possono essere assegnati, ma non c'è alcun mezzo primitivo per accedere al valore di un puntatore.

## 10.10 Evitare i dangling reference

Affrontiamo in questo paragrafo il problema di quali meccanismi possano essere inclusi in una macchina astratta per impedire dinamicamente la dereferenziazione di un dangling reference, problema che abbiamo discusso nel Paragrafo 10.4.5. Presenteremo per prima una soluzione radicale, che funziona nel caso generale di puntatori sia verso lo heap, sia verso la pila. Considereremo in seguito un meccanismo un po' più leggero, che però funziona solo nel caso di puntatori verso lo heap (e sotto certe assunzioni di probabilità).

### 10.10.1 Tombstone

Mediane le *tombstone* (*pietre tombali*) una macchina astratta è in grado di segnalare ogni tentativo di dereferenziare un dangling reference. Il meccanismo è concettualmente semplice: ogni volta che viene allocato un oggetto sullo heap cui si accede per puntatore, la macchina astratta alloca anche un'ulteriore parola di memoria (la tombstone, appunto). In modo analogo, una tombstone viene allocata anche tutte le volte che viene creato un puntatore che si riferisce alla pila (cioè, con una certa approssimazione, tutte le volte che viene usato l'operatore “indirizzo-di” &). La tombstone viene inizializzata con l'indirizzo dell'oggetto allocato, mentre il puntatore riceve l'indirizzo della tombstone. Quando si dereferenzia un puntatore, la macchina astratta inserisce un secondo livello di indirizzamento, per accedere prima alla tombstone e poi da questa all'oggetto puntato. Quando un puntatore viene assegnato ad un altro, è il contenuto del puntatore (e non della tombstone) ad essere modificato: la Figura 10.10 descrive graficamente questo funzionamento.

Alla deallocazione di un oggetto (o quando un indirizzo sulla pila diviene non più valido perché parte di un RdA che viene tolto dalla pila<sup>18</sup>), la tombstone relativa viene invalidata, memorizzandovi un valore particolare che segnala che i dati a cui si riferiva il puntatore sono “morti” (da cui il nome di questa tecnica). Scegliendo tale valore in modo opportuno si può fare in modo che ogni tentativo di accedere all'indirizzo contenuto in una tombstone invalidata sia catturato dal meccanismo di protezione degli indirizzi della macchina fisica sottostante.

Le tombstone sono allocate in un'area particolare della memoria della macchina astratta (il “cimitero”), che può essere gestita in modo più efficiente dello heap, visto che tutte le tombstone hanno la stessa dimensione.

Per quanto semplice, il meccanismo delle tombstone presenta un conto salato da pagare, sia in termini di efficienza che in termini di spazio. Per quanto riguarda il tempo, dobbiamo considerare il tempo necessario alla creazione della tombstone, quello del suo controllo (che come abbiamo detto può essere trascurato se

<sup>18</sup>Quest'operazione può non essere del tutto ovvia. Si consideri il caso di una variabile passata per riferimento ad una funzione; all'interno della funzione viene creato un puntatore a tale variabile, con relativa tombstone. La tombstone deve essere invalidata non quando la funzione termina, ma quando termina il tempo di vita della variabile che costituiva il parametro attuale.

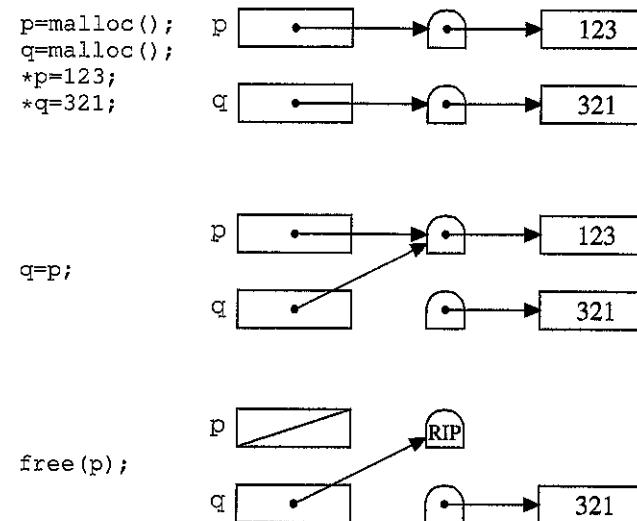


Figura 10.10 Tombstone.

è possibile usare un meccanismo di protezione di basso livello) e, soprattutto, il doppio accesso indiretto. Anche per quanto riguarda lo spazio le tombstone sono costose: richiedono una parola di memoria per ogni oggetto allocato sullo heap e per ogni creazione di puntatore verso la pila. Se le allocazioni sono tante e relative a oggetti di piccole dimensioni, la percentuale di memoria richiesta dalle tombstone può essere significativa. Inoltre, le tombstone invalidate rimangono per sempre allocate, con conseguente possibilità di esaurire lo spazio disponibile nel cimitero pur in presenza di molto spazio libero sullo heap. Per ovviare a quest'ultimo inconveniente, è possibile riusare le tombstone che non servono più (cioè quelle alle quali non punta nessun puntatore) mediante un piccolo garbage collector che usa la tecnica del contatore dei riferimenti (Paragrafo 10.11.1), ma questo aumenta ancora di più il costo in termini di tempo del meccanismo.

### 10.10.2 Lucchetti e chiavi

Un'alternativa alle tombstone è la tecnica nota come “*lucchetti e chiavi*” (*locks and keys*), che consente di risolvere il problema dei dangling reference verso lo heap con un meccanismo che non soffre del problema dell'accumulo delle tombstone.

Ogni volta che viene creato un oggetto sullo heap, viene associato all'oggetto anche un “lucchetto”, cioè una parola di memoria nella quale viene memorizzato un valore casuale (o anche un valore serialmente incrementato ad ogni nuova allocazione, purché siano evitati valori comuni come 0, 1, il codice di caratte-

ri frequenti ecc.). Un puntatore è costituito in quest'approccio da una coppia: l'indirizzo vero e proprio e una “chiave”, cioè una parola di memoria che viene inizializzata al valore del lucchetto corrispondente all'oggetto puntato. Quando un puntatore viene assegnato ad un altro, viene assegnata tutta la coppia; tutte le volte che si dereferenzia un puntatore, la macchina astratta controlla che la “chiave apra il lucchetto”, cioè che l'informazione contenuta nella chiave coincida con quella del lucchetto; in caso contrario viene segnalato un errore. Nel momento in cui un oggetto viene deallocated, il suo lucchetto viene annullato, memorizzandovi qualche valore canonico (per esempio zero), così che tutte le chiavi che prima lo aprivano, ora causino un errore (vedi la Figura 10.11). Può capitare, ovviamente, che l'area di memoria precedentemente usata come lucchetto venga riutilizzata (per un altro lucchetto o per altri scopi), ma è statisticamente assai improbabile che un eventuale errore non venga rilevato perché un ex-lucchetto si trova per caso ad avere lo stesso valore che aveva prima del suo annullamento.

Anche lucchetti e chiavi hanno un costo non trascurabile: in termini di spazio costano addirittura più delle tombstone, visto che è necessaria una parola aggiuntiva per ogni puntatore. D'altra parte, sia i lucchetti che le chiavi sono deallocated insieme all'oggetto o al puntatore di cui fanno parte. Da un punto di vista di efficienza, occorre tener presente sia il costo della creazione, sia, soprattutto, il maggior costo necessario all'assegnamento di un puntatore e quello necessario al controllo se la chiave apra il lucchetto, che avviene ad ogni dereferenziazione.

## 10.11 Garbage collection

Nei linguaggi senza deallocazione esplicita della memoria sullo heap, si rende necessario dotare la macchina astratta di un meccanismo con il quale si possa automaticamente recuperare la memoria allocata sullo heap e non più utilizzata: si tratta del *garbage collector* (spazzino), introdotto per la prima volta in LISP (intorno al 1960) e da allora presente in molti linguaggi, dapprima soprattutto funzionali, poi anche imperativi: Java ha un esteso ed efficiente garbage collector.

Da un punto di vista logico, il funzionamento di un garbage collector si compone di due fasi:

1. distinguere gli oggetti vivi da quelli non più utilizzati (*garbage detection*);
2. recuperare gli oggetti non più utilizzati, così che il programma li possa riutilizzare.

In pratica queste due fasi non sono sempre temporalmente separate e, soprattutto, le tecniche per il recupero degli oggetti dipendono in modo essenziale da quelle usate per la determinazione degli oggetti non più in uso. Vedremo, poi, che la nozione di “non più in uso” di un garbage collector è spesso un'approssimazione conservativa: per motivi di efficienza<sup>19</sup> non tutti gli oggetti che non saranno più usati sono in realtà scoperti come tali dal garbage collector.

<sup>19</sup>Nonché problemi di decidibilità...

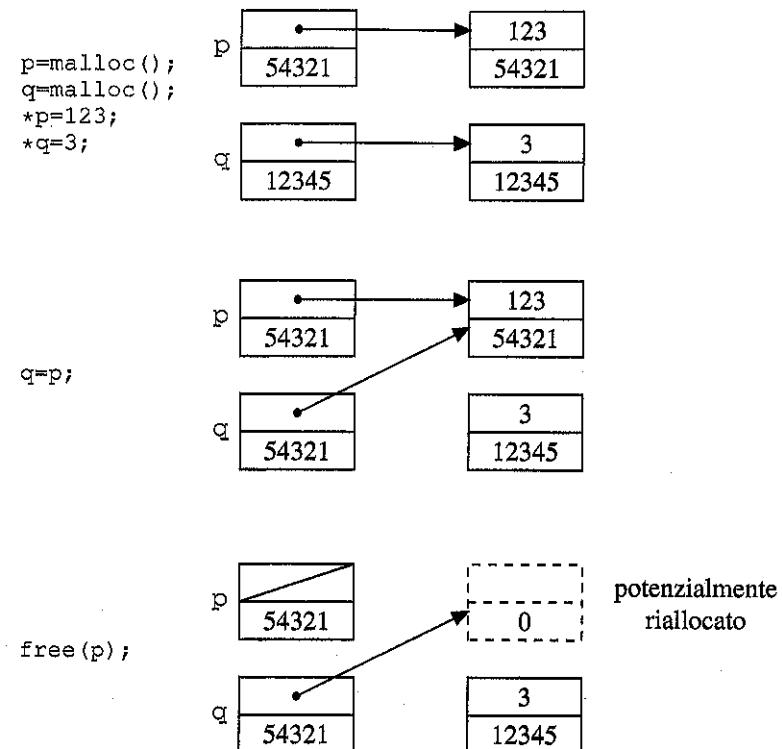


Figura 10.11 Lucchetti e chiavi.

Lo scopo didattico di questo testo non ci consente di descrivere un garbage collector per un linguaggio reale, né di fornire una panoramica esaurente delle diverse tecniche esistenti (si vedano i riferimenti bibliografici per delle rassegne esaustive). Ci limiteremo a presentare per sommi capi alcune delle tecniche più comuni, di cui i garbage collector reali sono in genere varianti o contaminazioni. Esulano in particolare dai nostri scopi i collector oggi più interessanti, basati su qualche forma di incrementalità del recupero della memoria.

Possiamo classificare i garbage collector classici a seconda di come determinano gli oggetti non più in uso: avremo così i collector basati su *contatori dei riferimenti*, su *marcatura* e su *copia*. Li presenteremo nelle sezioni che seguono, discutendo anche delle tecniche di recupero che vi sono associate. Discuteremo di queste tecniche parlando di puntatori e oggetti, ma l'argomento si applica in egual modo a linguaggi senza puntatori e con modello delle variabili a riferimento.

Vedremo, infine, che tutte le tecniche che tratteremo hanno la necessità di riconoscere, all'interno di un oggetto allocato sullo heap, se e quali campi corrispondano a puntatori. Se gli oggetti sono creati come istanze di tipi definiti staticamente (e dunque è staticamente nota la posizione dei puntatori all'interno

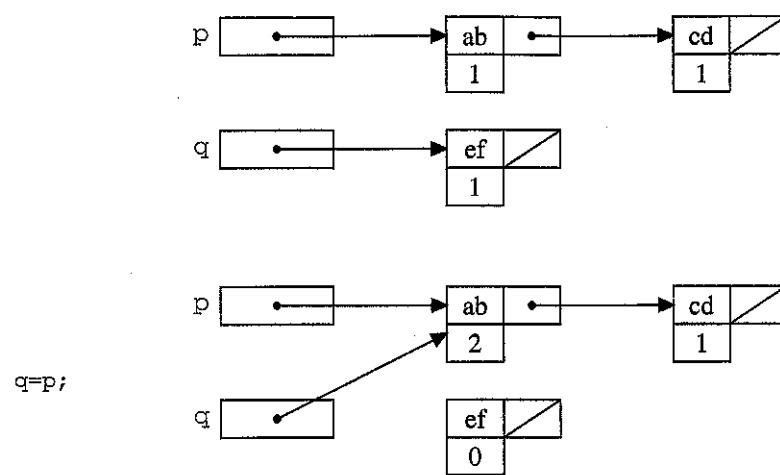


Figura 10.12 Contatori dei riferimenti.

dell'istanza), il compilatore può generare un descrittore per ogni tipo, nel quale sono indicati gli offset ai quali si trovano i puntatori. Ad ogni oggetto sullo heap è associato il tipo di cui è istanza (per esempio mediante un puntatore al descrittore del tipo<sup>20</sup>); al momento in cui un oggetto deve essere deallocated, il garbage collector accede al descrittore del tipo e, mediante questo, ai puntatori presenti all'interno dell'oggetto. Tecniche simili sono usate per riconoscere quali parole di un RDA corrispondono a puntatori. Se i tipi sono noti solo dinamicamente, i descrittori devono essere completamente dinamici e allocati anch'essi insieme all'oggetto.

### 10.11.1 Contatori dei riferimenti

La risposta più semplice che possiamo dare alla domanda relativa a quando un oggetto non è più utilizzato è: quando non vi sono puntatori verso di esso. La tecnica dei contatori dei riferimenti si basa su questa definizione e costituisce probabilmente il modo più elementare per realizzare un garbage collector.

Al momento della creazione sullo heap di un oggetto, viene allocato insieme ad esso anche un intero (il *contatore dei riferimenti*, o *reference count*, di quell'oggetto), inaccessibile al programmatore. La macchina astratta si incarica di far sì che, per ogni oggetto, a run-time tale contatore contenga il numero dei puntatori attivi a quell'oggetto.

<sup>20</sup>Se i puntatori hanno un tipo rigido ("puntatori a oggetti di tipo T"), è sufficiente questa informazione sul puntatore per ottenere quella sull'oggetto puntato e, transitivamente, quella su tutti gli oggetti di una struttura concatenata.

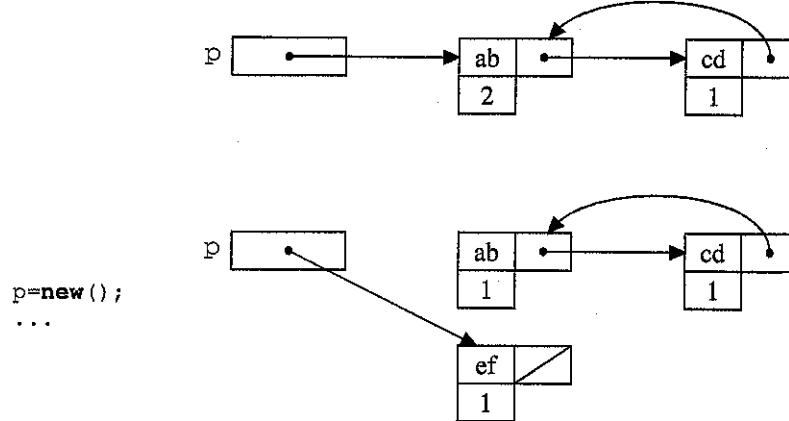


Figura 10.13 Strutture circolari con contatori dei riferimenti.

A questo scopo, al momento della creazione di un oggetto (e dunque dell'assegnamento del suo indirizzo ad un puntatore), il contatore è inizializzato a 1. In corrispondenza di un assegnamento tra puntatori

`p = q;`

il contatore dell'oggetto puntato da `q` viene incrementato di uno, mentre il contatore dell'oggetto puntato da `p` viene decrementato di uno. Quando si esce da un ambiente locale, sono decrementati di uno tutti i contatori degli oggetti puntati da puntatori locali a quell'ambiente. La Figura 10.12 illustra schematicamente il funzionamento di questa tecnica.

Quando un contatore, per effetto di questi aggiornamenti, raggiunge il valore 0, l'oggetto relativo può essere deallocated e restituito alla lista libera. Tale oggetto, tuttavia, potrebbe a sua volta avere dei puntatori al suo interno: la macchina astratta pertanto, quando un contatore assume il valore 0, prima di restituirlo alla lista libera segue tutti i puntatori in esso presenti, decrementando di uno i contatori degli oggetti puntati e recuperando ricorsivamente tutti gli oggetti il cui contatore raggiunge 0.

Dal punto di vista della divisione astratta in due fasi di un garbage collector, i calcoli e i controlli sui contatori implementano la fase di garbage detection, mentre la fase di recupero avviene quando il contatore è zero.

Un chiaro vantaggio di questa tecnica è la sua *incrementalità*: controllo e recupero sono mescolati con il funzionamento normale del programma. Con pochi aggiustamenti, è possibile adottare questa tecnica in sistemi real-time, cioè nei quali si debba dare una limitazione superiore assoluta al tempo di risposta del sistema.

Il difetto maggiore, almeno in via di principio, di questa tecnica sta nella sua incapacità di deallocare tutte le strutture circolari. La Figura 10.13 mostra

una situazione nella quale una struttura circolare non ha più cammini di accesso, ma non viene recuperata perché, ovviamente, i suoi contatori non sono zero. Si osservi che il problema non risiede tanto nell'algoritmo, quanto nella definizione stessa di cosa sia un oggetto non più utilizzato: è evidente che tutti gli oggetti della struttura circolare non sono più utilizzabili, ma non ricadono nella definizione di "non aver puntatori verso di essi".

I contatori dei riferimenti, a dispetto della loro semplicità, sono anche abbastanza *inefficienti*, in quanto hanno un costo proporzionale al lavoro complessivo compiuto dal programma (e non alla dimensione dello heap o alla percentuale di esso in uso o non in uso). Si pensi soprattutto all'aggiornamento dei contatori nel caso di parametri di tipo puntatore, passati a funzioni con breve tempo di vita: i contatori sono incrementati per essere riportati al loro valore originale dopo brevissimo tempo.

### 10.11.2 Mark and sweep

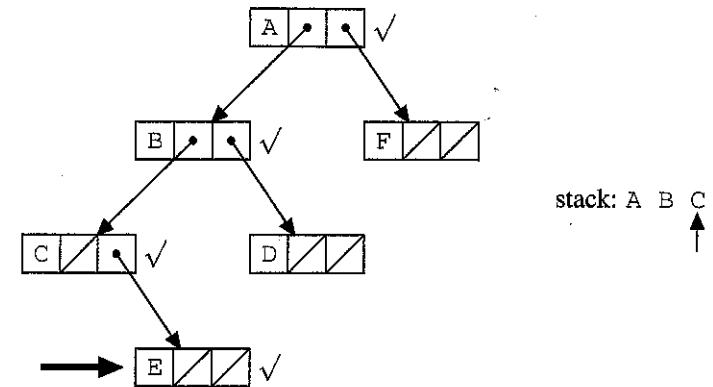
La tecnica di *mark and sweep* prende il proprio nome dalle modalità con le quali vengono realizzate le due fasi astratte che abbiamo menzionato all'inizio:

- *mark*: per riconoscere cosa è inutilizzato, si attraversano una prima volta tutti gli oggetti dello heap, marcando ciascuno di essi come "inutilizzato"; partendo poi dai puntatori attivi presenti sulla pila (l'*insieme radice*, o *root set*), si attraversano ricorsivamente tutte le strutture dati presenti sullo heap (in genere attraverso una visita in ampiezza o in profondità), marcando come "in uso" ogni oggetto che viene attraversato.
- *sweep*: lo heap viene spazzato (*swept*): tutti i blocchi marcati come "in uso" sono lasciati immutati, mentre quelli "inutilizzati" sono restituiti alla lista libera.

Si osservi che per poter implementare entrambe le fasi è necessario poter riconoscere i blocchi allocati nello heap: saranno pertanto necessari dei descrittori che diano la dimensione (e la struttura) di ogni blocco allocato.

A differenza dei garbage collector a contatori dei riferimenti, un collector mark and sweep non è incrementale: verrà invocato dalla macchina astratta quando la memoria libera disponibile sullo heap è prossima ad esaurirsi. L'utente del programma può in tal caso sperimentare un significativo degrado della performance di risposta del sistema, in attesa che il garbage collector abbia terminato il proprio lavoro.

La tecnica di mark and sweep soffre di tre difetti principali. In primo luogo, così come del resto accade per il reference count, è asintoticamente causa di frammentazione esterna (si veda il Paragrafo 7.4.2): oggetti vivi e non più in uso sono arbitrariamente intercalati sullo heap, e ciò può rendere difficile allocare un oggetto di grandi dimensioni anche se molti blocchi piccoli sarebbero disponibili. Il secondo problema è di efficienza: richiede tempo proporzionale alla dimensione totale dello heap, indipendentemente dalla percentuale di spazio usato o non usato. Il terzo problema ha a che vedere con la località dei riferimenti: gli oggetti "in uso" rimangono al loro posto, ma è possibile che oggetti ad essi contigui



**Figura 10.14** Necessità di una pila per una visita in profondità.

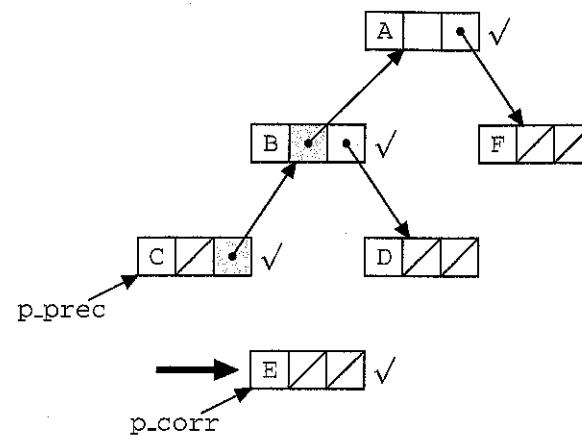
siano stati recuperati e che al loro posto siano stati allocati nuovi oggetti. A regime troveremo, come contigui, oggetti di età tra loro molto diverse, il che in genere diminuisce drasticamente la località dei riferimenti, con l'usuale degrado di performance in presenza di gerarchie di memoria.

### 10.11.3 Intermezzo: rovesciare i puntatori

Senza qualche precauzione, ogni tecnica di marcatura corre il rischio di essere completamente inutilizzabile in un garbage collector. Il collector, infatti, entra in azione quando la memoria sta esaurendosi, mentre la fase di marcatura comporta la visita ricorsiva di un grafo, che necessita in modo essenziale di una pila per memorizzare i punti di ritorno<sup>21</sup>. Per marcare un grafo in queste condizioni, pertanto, occorre usare astutamente lo spazio già presente per i puntatori, secondo una tecnica che va sotto il nome di *rovesciamento dei puntatori* (pointer reversal).

Come descritto nella Figura 10.14, per visitare una struttura concatenata, occorre marcare un nodo e visitare ricorsivamente le sottostrutture i cui puntatori siano parte del nodo. In questo schema ricorsivo, bisogna memorizzare su una pila l'indirizzo del blocco appena visitato, così da poter tornare indietro quando si raggiunga il termine di una sottostruttura. In figura, si sta eseguendo una visita in profondità e si è raggiunto il nodo E (i blocchi marcati sono indicati con "✓"): la pila contiene i nodi A, B e C. Dopo aver visitato E, si toglie C dalla pila, e si seguono eventuali altri puntatori che escono da questo nodo; siccome non ve ne sono, si toglie B dalla pila e si segue il puntatore restante, mettendo di nuovo B

<sup>21</sup>In molte macchine astratte pila e heap sono realizzati in un'unica area di memoria, con pila e heap che crescono in direzioni opposte a partire dai due estremi. In tal caso, quando non c'è spazio sullo heap, non c'è spazio neppure sullo stack.



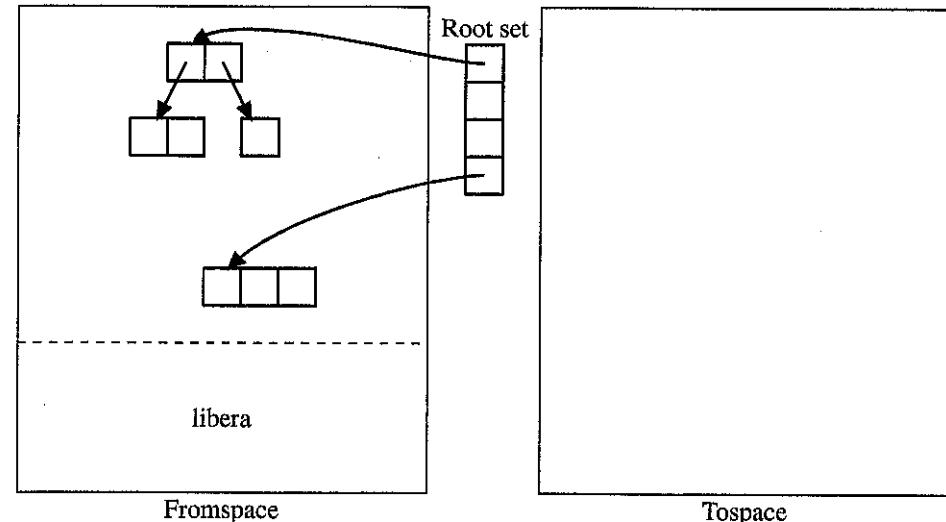
**Figura 10.15** Rovesciamento dei puntatori.

in pila e visitando D. Mediante il rovesciamento dei puntatori, questa pila viene memorizzata al posto dei puntatori che costituiscono la struttura, come descritto nella Figura 10.15. Quando si arriva al termine della sottostruttura (e si dovrebbe eseguire un pop dalla pila), si ripristina il puntatore al suo valore originale, cosicché al termine della visita la struttura è esattamente nella stessa situazione iniziale (a parte la marcatura, ovviamente). In figura, stiamo visitando il nodo E: si osservi come la pila della Figura 10.14 sia rappresentata all'interno della struttura stessa (i puntatori con campo grigio). Sono necessari solo due puntatori (`p_prec` e `p_corr`) per gestire la visita. Dopo aver visitato E, usando `p_prec` si risale nella struttura di un passo; usando il puntatore invertito si risale a B, ripristinando il valore corretto del puntatore in C usando `p_corr`.

#### 10.11.4 Mark and compact

Per ovviare alla frammentazione causata dalla tecnica di mark and sweep, possiamo modificare la fase di sweep e convertirla in una fase di compattamento: gli oggetti vivi sono spostati in modo da renderli contigui e lasciare tutta la memoria libera in un unico blocco contiguo. La compattazione si può eseguire scandendo linearmente lo heap e “traslando” ogni blocco vivo che si incontra, in modo da renderlo contiguo al blocco vivo precedente. Alla fine, tutti i blocchi liberi saranno contigui, così come tutti i blocchi non più in uso.

Si tratta di una tecnica che, come mark and sweep, necessita di diversi passaggi sullo heap (e dunque proporzionale alla dimensione di questo). Il compattamento, da solo, necessita di due o tre passaggi: il primo per calcolare la nuova posizione che ogni blocco vivo andrà ad assumere, un secondo per aggiornare i puntatori interni agli oggetti, un terzo per spostare davvero gli oggetti. Si tratta



**Figura 10.16** Stop and copy prima della chiamata al garbage collector.

pertanto di una tecnica sostanzialmente più costosa di mark and sweep se vi sono molti oggetti vivi che devono essere spostati.

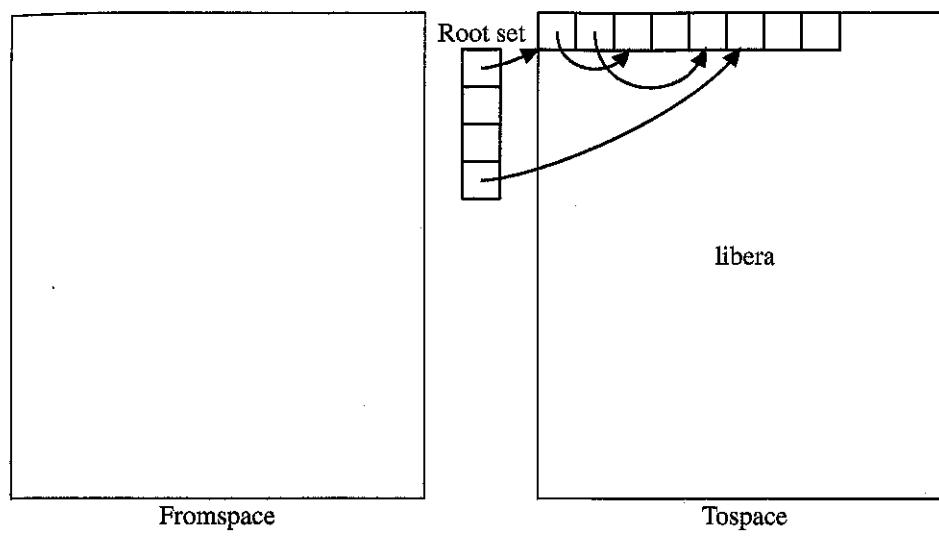
D'altra parte, la compattazione ha ottime ripercussioni sulla frammentazione e sulla località e permette la gestione della lista libera come un unico blocco, da cui si ritagliano i nuovi oggetti da allocare con una semplice sottrazione da un puntatore.

#### 10.11.5 Copia

Nei garbage collector basati su copia non c'è una vera fase di marcatura del “garbage”, quanto soprattutto la copia e compattazione dei blocchi vivi. La mancanza di un'esplicita fase di mark e la notevole differenza nella gestione dello spazio rende il suo costo sostanzialmente diverso da quello degli algoritmi basati su marcatura.

Nel più semplice garbage collector basato su copia (detto *stop and copy*) lo heap è diviso in due parti di uguali dimensioni (i due *semispazi*). Durante l'esecuzione normale, solo uno dei due semispazi è in uso: la memoria è allocata ad un'estremità del semispazio, mentre la memoria libera consiste di un unico blocco contiguo che si assottiglia ad ogni nuova allocazione, si veda la Figura 10.16. L'allocazione è estremamente efficiente e non c'è frammentazione.

Quando la memoria del semispazio è esaurita, viene invocato il garbage collector. Questi, a partire dai puntatori presenti sulla pila (il *root set*) inizia una visita delle strutture concatenate presenti nel semispazio corrente (il *fromspace*), copiandole una dopo l'altra nell'altro semispazio (il *tospace*), compattandole ad



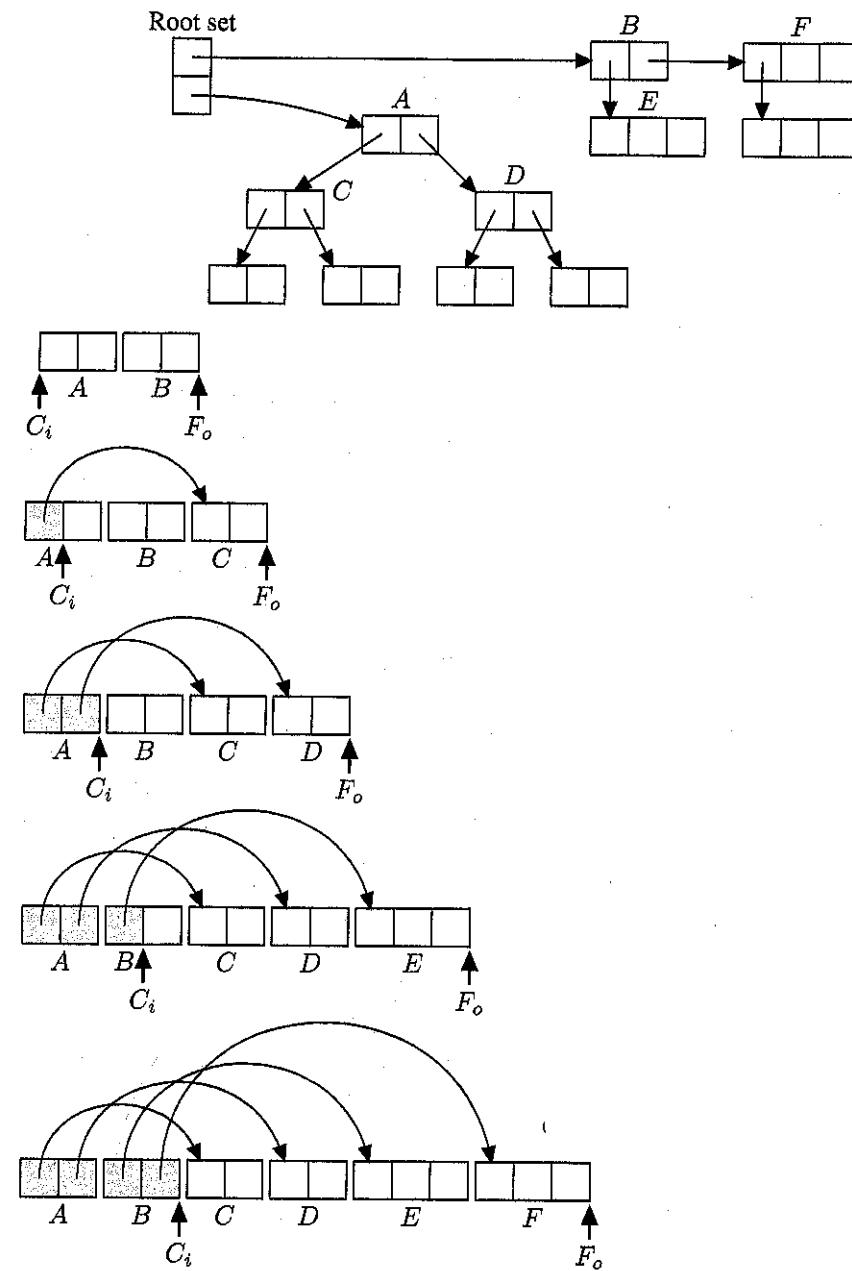
**Figura 10.17** Stop and copy dopo l'esecuzione del garbage collector.

un'estremità di questo, vedi la Figura 10.17. Al termine di questo processo, il ruolo dei due semispazi viene invertito e l'esecuzione ritorna al programma utente.

La visita e la copia della parte viva può essere eseguita in modo efficiente usando una semplice tecnica nota come algoritmo di Cheney, Figura 10.18. Inizialmente si copiano nel tospace tutti gli oggetti immediatamente raggiungibili a partire dal root set. Questo primo insieme di oggetti copiati in modo contiguo nel tospace è gestito come una coda: si prende in considerazione il primo di tali oggetti, aggiungendo in fondo alla coda (cioè copiando nel tospace) gli oggetti puntati dai puntatori presenti nell'oggetto e contemporaneamente modificando questi puntatori. In questo modo abbiamo copiato nel tospace tutti i figli del primo oggetto. Continuiamo ad elaborare la coda fino a quando questa non si sia svuotata. A quel punto avremo nel tospace una copia degli oggetti vivi del fromspace<sup>22</sup>.

Un garbage collector stop and copy può essere reso arbitrariamente efficiente, a patto di avere abbastanza memoria per i due semispazi. Infatti, il tempo richiesto da un collector stop and copy è proporzionale alla quantità di oggetti vivi presenti sullo heap. Non è irragionevole supporre che tale quantità sia approssimativamente costante in un qualsiasi momento dell'esecuzione di un programma. Se aumentiamo la memoria per i due semispazi, diminuiremo la frequenza con la quale è chiamato il collector e, dunque, il costo totale di garbage collection.

<sup>22</sup>Qualche precauzione deve essere presa per evitare di copiare più volte gli oggetti raggiungibili con più di un puntatore.



**Figura 10.18** Algoritmo di Cheney.

## 10.12 Sommario del capitolo

Il capitolo ha trattato un aspetto cruciale della definizione di un linguaggio di programmazione, quello che riguarda l'organizzazione dei dati in strutture astratte chiamate tipi di dato. Ricordiamo i seguenti aspetti principali:

- la *definizione di tipo*, come insieme di valori ed operazioni, e il ruolo dei tipi nel progetto, realizzazione ed esecuzione di un programma;
- i *sistemi di tipi*, come l'insieme dei costrutti e dei meccanismi che regolano e definiscono l'uso dei tipi in un linguaggio di programmazione;
- la distinzione tra controlli di tipo *dinamici* e controlli *statici*;
- il concetto di sistema di tipi *type safe*, cioè sicuro rispetto ai tipi;
- i principali *tipi scalari*, tra i quali troviamo i tipi *discreti*;
- i principali *tipi composti*, tra i quali abbiamo discusso in dettaglio i *record*, i *record varianti* e le *unioni*, gli *array*, i *puntatori*; per ciascuno di essi abbiamo presentato anche le principali tecniche di memorizzazione;
- il concetto di *equivalenza tra tipi*, distinguendo tra equivalenza per nome e equivalenza strutturale;
- il concetto di *compatibilità* e quelli, collegati, di coercizione e conversione;
- il concetto di *overloading*, quando uno stesso nome denota più oggetti e la disambiguazione avviene staticamente;
- il concetto di *polimorfismo universale*, quando uno stesso nome denota un oggetto che appartiene a più tipi diversi, distinguendo ulteriormente in polimorfismo parametrico e polimorfismo di sottotipo;
- l'*inferenza di tipo*, cioè i meccanismi che permettono di derivare il tipo di un'espressione complessa a partire dal tipo dei suoi componenti elementari;
- le tecniche per i controlli a tempo d'esecuzione per i *dangling reference*: tombstone e lucchetti e chiavi;
- le tecniche di *garbage collection*, cioè di recupero automatico della memoria, presentando per sommi capi i collector basati su contatori dei riferimenti, mark and sweep, mark and compact, copia.

I tipi sono il cuore di un linguaggio di programmazione: molti linguaggi si assomigliano quanto a costrutti per il controllo, ma quello che li differenzia sono i sistemi di tipi. Non è possibile comprendere gli aspetti essenziali di altri paradigmi, tra cui quello orientato agli oggetti, senza una competenza approfondita delle questioni presentate in questo capitolo.

## 10.13 Nota bibliografica

Ampie trattazioni dei tipi di dato nel contesto dei linguaggi di programmazione sono [87] e, per quanto un po' datata, [27]. Articoli di rassegna, che introducono il formalismo matematico necessario nella ricerca del settore sono [20] e [71]. Una trattazione più ampia degli stessi argomenti è in [79]. Sui sistemi di tipo, l'overloading e il polimorfismo, una bella e chiara rassegna è [21], che abbiamo largamente seguito nella nostra presentazione.

Le tombstone originano da [60] (vedi anche, dello stesso autore, [61]). Fisher e Leblanc [38] hanno proposto i locks and keys, insieme a tecniche affinché una macchina astratta di Pascal possa render sicuri i record varianti.

La definizione ufficiale di ALGOL 68, di ardua lettura, è in [101]; un'introduzione più accessibile è [75]. La definizione di ML è in [70]; [19] è un'esposizione introduttiva dell'inferenza per tipi polimorfi.

Sulle tecniche di garbage collection c'è amplissima letteratura. Una descrizione dettagliata di un algoritmo mark and sweep si trova su diversi manuali di algoritmi, per esempio [47]; [105] è una buona rassegna delle tecniche classiche. Per un testo interamente dedicato all'argomento, con pseudocodice e bibliografia praticamente completa (alla data di pubblicazione) si veda [49].

## 10.14 Esercizi

1. Si consideri la dichiarazione di array multidimensionale

```
int A[10][10][10].
```

Sappiamo che: un intero è memorizzato su 4 byte; l'array è memorizzato in ordine di riga, con indirizzi di memoria crescenti (cioè se un elemento è all'indirizzo  $i$ , il successivo è a  $i + 4$  ecc.); l'elemento  $A[0][0][0]$  è memorizzato all'indirizzo 0. Si dica a quale indirizzo è memorizzato l'elemento  $A[2][2][5]$ .

2. Al posto dell'allocazione contigua di array multidimensionali che abbiamo discusso nel Paragrafo 10.4.3, alcuni linguaggi permettono (C) o adottano (Java) una diversa organizzazione, detta organizzazione a righe di puntatori (*row-pointer*). Trattiamo il caso di un array bidimensionale. Invece di memorizzare le righe una dopo l'altra, ogni riga è memorizzata separatamente, in una porzione di memoria qualsiasi (per esempio sullo heap). In corrispondenza al nome del vettore è allocato un vettore di puntatori, ciascuno dei quali punta ad una riga dell'array vero e proprio. (i) Si dia la formula per l'accesso al generico elemento  $A[i][j]$  in questa organizzazione; (ii) Si generalizzi questa tecnica di memorizzazione ad array di rango maggiore di due; (iii) Si discutano vantaggi e svantaggi di questa organizzazione nel caso generale; (iv) Si discutano vantaggi e svantaggi di questa organizzazione per la memorizzazione di array bidimensionali di caratteri (cioè array di stringhe).
3. Si considerino le seguenti dichiarazioni (Pascal):

```
type stringa = packed array [1..16] of char;
type punt_stringa = ^stringa;
type persona = record
    nome : stringa;
    case studente: Boolean of
        true: (matricola: integer);
        false: (codicefiscale: punt_stringa)
    end;
```

e si supponga che la variabile C contenga un puntatore alla stringa LINGUAGGI. Si descriva la rappresentazione in memoria del record persona dopo ognuna delle seguenti istruzioni:

```
var pippo : persona;
pippo.studente := true;
pippo.matricola := 223344;
pippo.studente := true;
pippo.codicefiscale := C;
```

4. Si mostri che il tipo degli interi può essere definito come tipo ricorsivo a partire dal solo valore null. Si scriva poi una funzione che verifica se due interi così definiti sono uguali.
5. Sono date le seguenti definizioni di tipo in un linguaggio di programmazione che usa equivalenza strutturale tra tipi:

```
type T1 = struct{
    int a;
    bool b;
};

type T2 = struct{
    int a;
    bool b;
};

type T3 = struct{
    T2 u;
    T1 v;
};

type T4 = struct{
    T1 u;
    T2 v;
};
```

Nello scope delle dichiarazioni T3 a; T4 b; si dica se è ammesso l'assegnamento a = b; giustificando la risposta.

6. Quale tipo viene assegnato a ciascuna delle seguenti funzioni usando l'inferenza per tipi polimorfi?

```
fun G(f,x){return f(f(x));}
fun H(t,x,y){if (t(x)) return x;
             else return y;}
fun K(x,y){return x;}
```

7. Nella tecnica delle tombstone, occorre invalidare le tombstone quando un oggetto non è più significativo. La cosa è semplice se l'oggetto è sullo heap e viene esplicitamente deallocated; è più delicata quando la tombstone è associata ad un indirizzo sulla pila. In tal caso, infatti, occorre che la macchina astratta sia in grado di determinare tutte le tombstone eventualmente associate ad un RdA. Si progetti una possibile organizzazione che renda tale operazione ragionevolmente efficiente (si ricordi che le tombstone non sono allocate nel RdA, ma nel cimitero).
8. Si consideri il seguente frammento in uno pseudolinguaggio con modello delle variabili a riferimento e che adotta la tecnica locks and keys (C è una classe di cui non ci interessa la struttura):

```
C foo = new C(); // oggetto OG1
C bar = new C(); // oggetto OG2
C fie = foo;
bar = fie;
```

Si diano possibili valori di tutte le chiavi e tutti i lucchetti coinvolti, dopo l'esecuzione del frammento.

9. Nelle stesse ipotesi dell'Esercizio 8, si consideri il seguente frammento di codice, dove free(p) indica la deallocazione esplicita dell'oggetto riferito dal puntatore p:

```
class C { int n; C next;}
C foo = new C(); // oggetto OG1
C bar = new C(); // oggetto OG2
foo.next = bar;
bar.next = foo;
free(bar);
```

Per tutti i puntatori coinvolti, si diano possibili valori per le chiavi; per ogni oggetto coinvolto, si diano possibili valori per i lucchetti, al termine dell'esecuzione del frammento. Dopo aver eseguito il frammento, si esegue anche il codice foo.n = 1; foo.next.n = 0;. Si dica qual è un possibile risultato dell'esecuzione.

10. Si consideri il seguente frammento in uno pseudolinguaggio con modello delle variabili a riferimento e garbage collector mediante contatori dei riferimenti; se OGG è un generico oggetto nello heap, indichiamo con OGG.cont il suo contatore (nascosto).

```
class C { int n; C next;}
C foo(){
    C p = new C(); // oggetto OGG1
    p.next = new C(); // oggetto OGG2
    C q = new C(); // oggetto OGG3
    q.next = p.next;
    return p.next;
}
C r = foo();
```

Si dica quali sono i valori dei contatori dei riferimenti dei tre oggetti dopo l'esecuzione della linea 6 e della linea 9.

11. Nelle stesse ipotesi dell'Esercizio 10, si dica quali sono i valori dei contatori dei riferimenti dei due oggetti dopo l'esecuzione del seguente frammento. Quale dei due può essere restituito alla lista libera?

```
C foo = new C(); // oggetto OG1
C bar = new C(); // oggetto OG2
C fie = foo;
bar = fie;
```

12. Nelle stesse ipotesi dell'Esercizio 10, si dica quali sono i valori dei contatori dei riferimenti dei tre oggetti dopo l'esecuzione del seguente frammento. Quali di essi possono essere restituiti alla lista libera?

```
class C { int n; C next;}
```

```
C foo = new C(); // oggetto OG1
bar = new C(); // oggetto OG2
foo.next = bar;
bar = new C(); // oggetto OG3
foo = bar;
```

13. Usando il linguaggio di programmazione preferito, si definiscano le opportune strutture dati per un albero binario; si scriva poi del codice dettagliato che esegua una visita in ordine anticipato di un albero senza usare la pila di sistema e usando invece la tecnica di rovesciamento dei puntatori.

## Astrarre sui dati

La macchina fisica ha dati di un solo tipo: le stringhe di bit. I tipi di un linguaggio di alto livello impongono un'organizzazione su tale universo indistinto, dotando ogni valore di una sorta di "rivestimento": ogni valore è racchiuso in una capsula (il suo tipo) che fornisce le operazioni che possono manipolarlo. Il sistema di tipi di un linguaggio stabilisce quanto questa capsula di rivestimento è trasparente: nei linguaggi sicuri rispetto ai tipi, la capsula è del tutto opaca, nel senso che non permette l'accesso alla rappresentazione (o, meglio: ogni accesso può avvenire soltanto attraverso e per la mediazione della capsula stessa).

Il capitolo precedente ha presentato in dettaglio molti dei tipi predefiniti e dei principali meccanismi di definizione di nuovi tipi. Questi ultimi, tuttavia, sono abbastanza limitati: aggregazioni finite omogenee (array) e disomogenee (record), tipi ricorsivi e puntatori. Le operazioni possibili su tali tipi composti sono predefinite dal linguaggio e il programmatore può solo sfruttarle per i propri scopi. A ben vedere, coi meccanismi che abbiamo discusso nel Capitolo 10 il programmatore non ha vera possibilità di definire un *nuovo tipo*, inteso, secondo la nostra definizione, come una collezione di valori (omogenei ed effettivamente presentati) dotata di un insieme di operazioni. L'utente di un linguaggio può solo sfruttare le capsule esistenti ed ha solo modi molto limitati di definirne di nuove: vi sono pochi meccanismi di *astrazione sui dati*.

Presenteremo in questo capitolo alcuni dei principali modi con cui un linguaggio può mettere a disposizione dei meccanismi più sofisticati per definire astrazioni sui dati. Tra questi, discuteremo i cosiddetti tipi di dato astratti, che in varie forme sono possibili in molti linguaggi diversi. Accenneremo poi ai moduli, un concetto sostanzialmente simile, ma che si applica soprattutto alla programmazione in grande. Questi meccanismi di astrazione costituiscono anche un'introduzione ad alcuni temi che ritroveremo nella programmazione orientata agli oggetti. I concetti chiave che ci seguiranno in questo capitolo saranno la separazione tra *interfaccia* e *implementazione*, e la nozione collegata di *occultamento dell'informazione*.

```

type Int_Stack = struct{
    int P[100]; // la pila vera e propria
    int top; // primo elemento libero
}
Int_Stack crea_pila(){
    Int_Stack s = new Int_Stack();
    s.top = 0;
    return s;
}
Int_Stack push(Int_Stack s, int k){
    if (s.top == 100) errore;
    s.P[s.top] = k;
    s.top = s.top + 1;
    return s;
}
int top(Int_Stack s){
    return s.P[s.top];
}
Int_Stack pop(Int_Stack s){
    if (s.top == 0) errore;
    s.top = s.top - 1;
    return s;
}
bool empty(Int_Stack s){
    return (s.top == 0);
}

```

Figura 11.1 Pile di interi.

## 11.1 Tipi di dato astratti

L'introduzione di nuovi tipi con i meccanismi discussi nel capitolo precedente non permette all'utente di un linguaggio la definizione di tipi con lo stesso grado di astrazione goduto dai tipi predefiniti nel linguaggio.

A titolo d'esempio, la Figura 11.1 riporta una possibile definizione del tipo di dato "pila di interi" nel nostro pseudolinguaggio, assumendo un modello delle variabili a riferimento. Quando si definisce un tipo del genere, si vuole probabilmente intendere che una pila di interi è una struttura dati manipolabile con le operazioni di creazione, inserimento, accesso all'elemento sulla cima, eliminazione dell'elemento in cima. Tuttavia il linguaggio non garantisce che questi siano i *soli* modi con cui una pila può essere manipolata. Se anche adottiamo un'equivalenza tra tipi per nome stretta (in modo che una pila sia introdotta solo attraverso una dichiarazione di tipo `Int_Stack`) nulla impedisce che si possa accedere direttamente alla sua rappresentazione come array:

```

int second_from_top() (Int_Stack c){
    return c.P[s.top - 1];
}

```

Da un punto di vista generale, dunque, mentre il linguaggio fornisce delle astrazioni sui dati (i tipi predefiniti) che nascondono l'implementazione, lo stesso non è possibile al programmatore. Per ovviare a questo problema, alcuni linguaggi di programmazione permettono di definire delle astrazioni sui dati che si comportano come i tipi predefiniti rispetto all'(in-)accessibilità della rappresentazione. Questo meccanismo, che chiamiamo *tipo di dato astratto* (o *ADT*), è caratterizzato dai seguenti aspetti principali:

1. un nome per il tipo;
2. un'implementazione (o rappresentazione) per tale tipo;
3. un insieme di nomi di operazioni per la manipolazione dei valori di quel tipo, con i loro tipi;
4. per ogni operazione, un'implementazione che usi la rappresentazione fornita al punto 2;
5. una capsula di sicurezza che separi i nomi del tipo e delle operazioni dalle loro realizzazioni.

Una possibile notazione per l'ADT delle pile di interi nel nostro pseudolinguaggio potrebbe essere quella riportata nella Figura 11.2.

Una definizione del genere deve essere interpretata nel modo seguente: la prima linea introduce il nome del tipo di dato astratto; la linea 2 fornisce la rappresentazione (o *tipo concreto*) per il tipo astratto `Int_Stack`; le linee da 7 a 12 (introdotte da `signature`) definiscono i nomi e i tipi delle operazioni che possono manipolare un `Int_Stack`; le linee restanti (introdotte da `operations`) forniscono le implementazioni delle operazioni. Il punto saliente di questa definizione è che all'interno della dichiarazione il tipo `Int_Stack` è un sinonimo della sua rappresentazione concreta (e dunque le operazioni manipolano una pila come un record che contiene un array e due campi interi), mentre fuori di essa (dunque nella parte restante di programma), non c'è più alcuna relazione tra un `Int_Stack` e il suo tipo concreto: i soli modi possibili per manipolare un `Int_Stack` sono quelli dati dalle operazioni. La funzione `second_from_top()` che prima avevamo definito, ora è impossibile, perché il controllore dei tipi non permette di applicare un selettore di campo ad un `Int_Stack` (in quanto *non* è un record al di fuori della definizione).

Un ADT è una capsula opaca: sulla superficie esterna, visibile a chiunque, troviamo il nome del tipo, il nome e il tipo delle operazioni; al suo interno, invisibili da fuori, vi sono le implementazioni del tipo e delle operazioni. L'accesso all'interno è sempre mediato dalla capsula, che garantisce la consistenza delle informazioni che racchiude. Questa superficie esterna della capsula si chiama *segnatura* (*signature*<sup>1</sup>) o *interfaccia* dell'ADT; al suo interno sta l'*implementazione*.

<sup>1</sup>C'è chi traduce "signature" con "firma". Si tratta di una traduzione poco felice: in questo contesto l'inglese usa signature nell'accezione di "insieme di segni che contraddistinguono un documento", che in italiano si dice appunto "segnatura".

```

abstype Int_Stack{
    type Int_Stack = struct(
        int P[100];
        int n;
        int top;
    )
    signature
        Int_Stack crea_pila();
        Int_Stack push(Int_Stack s, int k);
        int top(Int_Stack s);
        Int_Stack pop(Int_Stack s);
        bool empty(Int_Stack s);
    operations
        Int_Stack crea_pila(){
            Int_Stack s = new Int_Stack();
            s.n = 0;
            s.top = 0;
            return s;
        }
        Int_Stack push(Int_Stack s, int k){
            if (s.n == 100) errore;
            s.n = s.n + 1;
            s.P[s.top] = k;
            s.top = s.top + 1;
            return s;
        }
        int top(Int_Stack s){
            return s.P[s.top];
        }
        Int_Stack pop(Int_Stack s){
            if (s.n == 0) errore;
            s.n = s.n - 1;
            s.top = s.top - 1;
            return s;
        }
        bool empty(Int_Stack s){
            return (s.n == 0);
        }
}

```

Figura 11.2 ADT per pile di interi.

```

abstype Int_Var{
    type Int_Var = Int_Stack;
    signature
        Int_Var crea_var();
        int deref(Int_Var v);
        Int_Var assign(Int_Var v, int n);
    operations
        Int_Var crea_var(){
            return push(crea_pila(), 0);
        }
        int deref(Int_Var v){
            return top(v);
        }
        Int_Var assign(Int_Var v, int n){
            return push(pop(v), n);
        }
}

```

Figura 11.3 Un ADT per una variabile intera, implementato con una pila.

I tipi di dato astratti (nei linguaggi che li permettono, per esempio ML e CLU<sup>2</sup>) si comportano come i tipi predefiniti: è possibile dichiarare variabili di un tipo astratto, e usare un tipo astratto per definirne un altro, come fatto, a titolo d'esempio, nella Figura 11.3, che implementa (in modo assai inefficiente) una variabile di tipo intero sfruttando una pila. Si osservi che all'interno dell'implementazione di una `Int_Var`, non si vede l'implementazione di `Int_Stack`, che è stata incapsulata una volta per tutte al momento della sua definizione.

## 11.2 Nascondere l'informazione

La distinzione tra interfaccia ed implementazione è di grande importanza per le tecniche di sviluppo del software, perché permette di separare l'uso di un componente dalla sua definizione. Abbiamo già visto questa distinzione quando abbiamo trattato dell'astrazione sul controllo: una funzione “astrae” (cioè: nasconde) il codice che costituisce il suo corpo (l’implementazione), mentre mostra la sua interfaccia, costituita dal suo nome e dal numero e tipi dei suoi parametri (cioè la sua segnatura). L’astrazione sui dati generalizza quella forma un po’ primitiva di astrazione: non viene nascosto solo “come” una certa operazione è realizzata, ma anche le modalità di rappresentazione dei dati, in modo tale che il linguaggio (col suo sistema di tipi) possa garantire che l’astrazione non venga violata. Que-

<sup>2</sup>Nei linguaggi orientati agli oggetti vedremo che è possibile ottenere lo stesso scopo con un meccanismo simile, ma più flessibile.

### Costruttori, trasformatori e osservatori

Nella definizione di un tipo di dato astratto  $T$ , le operazioni sono concettualmente suddivise in tre categorie distinte:

- *costruttori*: operazioni che costruiscono un nuovo valore di tipo  $T$ , eventualmente usando valori di altri tipi noti;
- *trasformatori, o operatori*: operazioni che calcolano valori di tipo  $T$ , usando eventualmente altri valori (dell'ADT o di altri tipi); una proprietà fondamentale di un trasformatore  $t$  di tipo  $S_1 \times \dots \times S_k \rightarrow T$  è che per ogni valore degli argomenti, deve accadere che  $t(s_1, \dots, s_k)$  è un valore costruibile a partire dai soli costruttori;
- *osservatori*: operazioni che calcolano un valore di un tipo noto diverso da  $T$ , usando uno o più valori di tipo  $T$ .

Un ADT senza costruttori è del tutto inutile: non c'è modo per costruirne un valore. In generale, un ADT deve avere almeno un'operazione per ciascuna categoria. Non è sempre facile dimostrare che un'operazione è davvero un trasformatore (cioè che ogni suo valore è in realtà un valore ottenibile con una sequenza di costruttori).

Nell'esempio delle pile di interi, `crea_pila` e `push` sono costruttori, `pop` è un trasformatore, `top` e `empty` sono osservatori.

sto fenomeno viene indicato come *occultamento dell'informazione (information hiding)*. Una delle conseguenze più interessanti dell'occultamento dell'informazione è che, a determinate condizioni, è possibile sostituire l'implementazione di un ADT con un'altra, mantenendo immutata l'interfaccia. Nell'esempio della pila, potremmo scegliere di rappresentare il tipo concreto come una lista concatenata allocata sullo heap (non ci preoccupiamo della deallocazione): diamo quest'ultima definizione nella Figura 11.4. Sotto certe ipotesi, sostituendo la prima definizione di `Int_Stack` con quella della Figura 11.4 non si sperimenta nessun effetto *osservabile* nei programmi che usano il tipo di dato astratto. Queste ipotesi riguardano quello che i clienti dell'interfaccia si aspettano dalle operazioni. Chiamiamo *specifica* la descrizione della semantica delle operazioni di un ADT, espressa non in termini del tipo concreto, ma per mezzo di relazioni generali astratte. Una possibile specifica per il nostro ADT `Int_Stack` potrebbe essere:

- `crea_pila` crea una pila vuota;
- `push` inserisce un elemento nella pila;
- `top` restituisce l'elemento in cima alla pila, senza modificare la pila stessa; la pila deve essere non vuota;
- `pop` elimina l'elemento in cima alla pila; la pila deve essere non vuota;
- `empty` è vero se e solo se la pila è vuota.

Ogni cliente che usa `Int_Stack` sfruttando solo questa specifica, non vedrà alcuna differenza tra le due definizioni. Questa proprietà viene detta *principio di*

```

abstype Int_Stack{
    type Int_Stack = struct{
        int info;
        Int_stack next;
    }
    signature
        Int_Stack crea_pila();
        Int_Stack push(Int_Stack s, int k);
        int top(Int_Stack s);
        Int_Stack pop(Int_Stack s);
        bool empty(Int_Stack s);
    operations
        Int_Stack crea_pila(){
            return null;
        }
        Int_Stack push(Int_Stack s, int k){
            Int_Stack tmp = new Int_Stack(); // nuovo elemento
            tmp.info = k;
            tmp.next = s; // concatenalo
            return tmp;
        }
        int top(Int_Stack s){
            return s.info;
        }
        Int_Stack pop(Int_Stack s){
            return s.next;
        }
        bool empty(Int_Stack s){
            return (s == null);
        }
}

```

**Figura 11.4** Un'altra definizione per l'ADT `Int_Stack`.

*indipendenza dalla rappresentazione.*

La specifica di un tipo di dato astratto può essere data in molti modi diversi, che vanno dal linguaggio naturale (come noi abbiamo fatto), a schemi semi-formali, a linguaggi completamente formalizzati che possono essere manipolati da dimostratori di teoremi. Una specifica è una sorta di contratto tra l'ADT e i suoi clienti: l'ADT garantisce che l'implementazione delle operazioni (ignota ai clienti) *soddisfa* (in inglese si usa spesso il verbo *to match*) la specifica, cioè tutte le proprietà enunciate nella specifica sono soddisfatte nell'implementazione. Quando questo accade si dice anche che l'implementazione è *corretta* rispetto alla specifica.

### 11.2.1 Indipendenza dalla rappresentazione

Possiamo enunciare la proprietà di indipendenza dalla rappresentazione come segue:

Due implementazioni corrette di una stessa specifica di un ADT sono osservabilmente indistinguibili da parte dei clienti di quel tipo.

Se un tipo gode dell'indipendenza dalla rappresentazione, è possibile modificare la sua implementazione con una equivalente (e forse più efficiente) senza che ciò provochi nei clienti la comparsa di (nuovi) errori.

Dovrebbe essere evidente che una parte considerevole (e per niente ovvia) della proprietà di indipendenza risiede nella garanzia che entrambe le implementazioni siano corrette rispetto alla stessa specifica. Questo può non essere facile da mostrare, soprattutto se la specifica è informale. Vi è però una versione debole della proprietà di indipendenza dalla rappresentazione che riguarda la sola correttezza rispetto alla segnatura: in un linguaggio type safe con tipi di dato astratti, la sostituzione di un ADT con un altro con la stessa segnatura (ma diversa implementazione) non causa errori di tipo. Nelle ipotesi fatte (type safeness e ADT), questa proprietà è un teorema, garantito dall'inferenza dei tipi. Linguaggi come ML e CLU godono di questa forma di indipendenza dalla rappresentazione.

## 11.3 Moduli

I tipi di dato astratti sono meccanismi della “programmazione in piccolo”: sono pensati per incapsulare *un* tipo con le relative operazioni. È molto più comune, tuttavia, che un'astrazione sia composta da più tipi (o strutture dati) tra loro correlate, delle quali si vuole dare ai clienti una visione limitata (cioè astratta: non vengono mostrate tutte le operazioni e si ha occultamento dell'implementazione). I meccanismi linguistici che realizzano questo tipo di incapsulamento sono in genere chiamati *moduli* o *package* e appartengono a quella parte di un linguaggio di programmazione che si preoccupa della “programmazione in grande”, cioè della realizzazione di un sistema complesso mediante composizione e assemblaggio di componenti più semplici. Il meccanismo dei moduli permette di partizionare staticamente un programma in parti distinte, ciascuna delle quali dotata sia di dati (tipi, variabili ecc.) che di operazioni (funzioni, codice ecc.): un modulo raggruppa più dichiarazioni (di dati e/o funzioni) e insieme definisce delle regole di visibilità per quelle dichiarazioni, mediante le quali si può realizzare una forma di incapsulamento e di occultamento dell'informazione.

Da un punto di vista dei principi, non c'è grande differenza con gli ADT, se non nella possibilità di definire più tipi contemporaneamente (secondo le definizioni che abbiamo dato, un ADT è un caso particolare di un modulo). Da un punto di vista pragmatico, invece, il meccanismo dei moduli fornisce molta flessibilità in più, sia nella definizione del grado di permeabilità della capsula (talvolta è possibile indicare in modo individuale quali operazioni siano o meno visibili

```

module Buffer imports Counter{
    public
        type Buf;
        void insert(reference Buf f, int n);
        int get(Buf b);
        Count c; // quante volte ho usato il buffer
    private imports Queue{
        type Buf = Queue;
        void insert(reference Buf b, int n){
            enqueue(b,n);
            inc(c);
        }
        int get(Buf b){
            return dequeue(b);
            inc(c);
        }
        init_counter(c); // parte di inizializzazione del modulo
    }
    module Counter{
        public
            type Count;
            void init_counter(reference Count c);
            int get(Count c);
            void inc(reference Count c);
        private
            type Count = int;
            void init_counter(reference Count c){
                c=0;
            }
            int get(Count c){
                return c;
            }
            void inc(reference Count c){
                c = c+1;
            }
    }
    module Queue{
        public
            type Queue;
            enqueue(reference Queue q, int n);
            int dequeue(reference Queue q);
            ...
        private
            void bookkeep(reference Queue q){
                ...
            }
            ...
    }
}

```

Figura 11.5 Moduli.

all'esterno, o selezionare il grado di visibilità), sia nella possibilità di definire moduli generici, cioè polimorfi (si ricordi il Paragrafo 10.7). Infine, i costrutti relativi ai moduli sono spesso congiunti con un meccanismo di compilazione separata dei moduli stessi<sup>3</sup>.

Pur nell'ampissima varietà dei linguaggi esistenti, possiamo presentare le caratteristiche linguistiche salienti di un modulo discutendo l'esempio della Figura 11.5, espresso come al solito in un comodo pseudolinguaggio. Innanzitutto, un modulo è suddiviso, come un ADT, in una parte pubblica visibile da tutti i clienti del modulo stesso, ed una parte privata, invisibile dall'esterno. La parte privata di un modulo può contenere dichiarazioni non menzionate affatto nella sua parte pubblica (per esempio la funzione bookkeep dentro Queue). Un modulo può menzionare alcune delle proprie strutture dati nella parte pubblica, cosicché chiunque può usarle o modificarle (la variabile c di Buffer). Un modulo cliente può usare la parte pubblica di un altro modulo *importandolo* (clausola imports); nel nostro esempio Buffer ha una clausola di importazione aggiuntiva nella parte privata.

Non ci dilunghiamo oltre in questa discussione, sia perché dovremmo scendere nel dettaglio dei meccanismi di uno specifico linguaggio, sia perché su molti argomenti torneremo nel contesto della programmazione orientata agli oggetti. Osserviamo solo, per concludere, che il meccanismo dei moduli è spesso congiunto a qualche forma di polimorfismo parametrico, risolto a tempo di collegamento. Nel nostro esempio, avremmo potuto definire buffer di tipo T, invece che un buffer di interi, rendendo "generica" la definizione e istanziandola opportunamente al momento dell'uso. La Figura 11.6 riporta la versione generica dell'esempio del buffer. Si osservi come il buffer e la coda siano entrambi generici; quando la parte privata di Buffer importa Queue, specifica che deve essere istanziata sullo stesso tipo (non ancora specificato) di Buffer.

## 11.4 Sommario del capitolo

Questo breve capitolo ha fornito una prima introduzione ai fenomeni di astrazione sui dati, che ruotano attorno ai concetti chiave di *interfaccia*, *implementazione*, *incapsulamento*, *occultamento dell'informazione*.

Da un punto di vista linguistico abbiamo presentato:

- il meccanismo dei *tipi di dato astratti*;
- i meccanismi per nascondere l'informazione e la loro conseguenza, cioè il *principio di indipendenza dalla rappresentazione*;
- il meccanismo dei *moduli*, che applica alla programmazione in grande i concetti di encapsulamento.

<sup>3</sup>È bene tenere a mente, tuttavia, che moduli e compilazione separata sono due aspetti indipendenti di un linguaggio; in Java, ad esempio, non è il modulo (package, nella terminologia Java) l'unità di compilazione, bensì la classe, e un package può essere composto da molti file diversi.

```
module Buffer<T> imports Counter{
    public
        type Buf;
        void insert(reference Buf f, <T> n);
        <T> get(Buf b);
        Count c; // quante volte ho usato il buffer
    private imports Queue<T>{
        type Buf = Queue;
        void insert(reference Buf b, <T> n){
            enqueue(b, n);
            inc(c);
        }
        <T> get(Buf b){
            return dequeue(b);
            inc(c);
        }
    }
    module Counter{
        public
            type Count;
            void init_counter(reference Count c);
            int get(Count c);
            void inc(reference Count c);
        private
            type Count = int;
            void init_counter(reference Count c){
                c=0;
            }
            int get(Count c){
                return c;
            }
            void inc(reference Count c){
                c = c+1;
            }
            init_counter(c); // parte di inizializzazione del modulo
    }
    module Queue<S>{
        public
            type Queue;
            enqueue(reference Queue q, <S> n);
            <S> dequeue(reference Queue q);
            ...
        private
            void bookkeep(reference Queue q){
                ...
            }
            ...
    }
}
```

Figura 11.6 Moduli generici.

Tutti questi concetti sono trattati più approfonditamente nei testi di ingegneria del software. Nell'economia di questo libro sono però strumentali alla comprensione del paradigma orientato agli oggetti, che presenteremo nel prossimo capitolo.

## 11.5 Nota bibliografica

Il concetto di modulo compare probabilmente per la prima volta nel linguaggio Simula [14, 74], il primo linguaggio orientato agli oggetti (vedi il Capitolo 16). Lo sviluppo dei moduli nei linguaggi di programmazione è legato ai lavori di Wirth [108], che porteranno al progetto di Modula e di Oberon [110].

La nozione di occultamento dell'informazione (information hiding) fa la sua prima comparsa in letteratura in un classico articolo di Parnas [76]. I tipi di dato astratti originano in questo stesso contesto, come un meccanismo diverso dai moduli per garantire l'astrazione. Dei linguaggi che lo introducono, quello che ha avuto più influenza sui successori è certamente CLU [58], sul quale è basato anche il testo [57].

## 11.6 Esercizi

- Si consideri la seguente definizione di un ADT nel nostro pseudolinguaggio:

```
abstype PocoUtile{
    type PocoUtile = int;
    signature
        PocoUtile prox(PocoUtile x);
        int get(PocoUtile x);
    operations
        PocoUtile prox(PocoUtile x) {
            return x+1;
        }
        int get(PocoUtile x) {
            return x;
        }
}
```

Per qual motivo si tratta di un tipo di dato poco utile?

# Il paradigma orientato agli oggetti

Presenteremo in questo capitolo gli aspetti salienti dei linguaggi orientati agli oggetti, un paradigma che vede i suoi precursori in Simula (fine anni '60) e Smalltalk (anni '70) e che ha ottenuto enorme successo, anche commerciale, nei due decenni seguenti (C++ e Java sono solo i due linguaggi più noti tra i tanti che sono tutti oggi in uso). "Orientato agli oggetti" è ormai una locuzione abusata, che troviamo applicata non solo ai linguaggi di programmazione, ma anche ai metodi di sviluppo del software, alle basi di dati, ai sistemi operativi ecc. Il nostro tentativo sarà quello di presentare gli aspetti *linguistici* che riguardano gli oggetti e il loro uso, facendo qualche sporadico accenno alle tecniche di progetto orientate agli oggetti, per le quali, tuttavia, non possiamo che rimandare alla (sterminata) bibliografia.

Anche dopo aver così ristretto il campo, ci sono molti modi di avvicinarsi ai concetti che ci interessano. Noi seguiranno quello che crediamo il più semplice: presenteremo gli oggetti come un modo per ottenere astrazione sui dati, in modo flessibile ed estendibile. Inizieremo perciò il nostro studio mostrando alcune limitazioni delle tecniche che abbiamo introdotto nel Capitolo 11: questi limiti ci suggeriranno alcuni concetti che, di fatto, costituiscono i cardini di un linguaggio orientato agli oggetti. Dopo aver approfondito questi aspetti caratteristici, studieremo alcune estensioni delle soluzioni linguistiche oggi disponibili nei linguaggi commerciali, con riferimento in particolare alla nozione di sottotipo, polimorfismo e genericità.

Secondo lo stile che abbiamo mantenuto in tutto il testo, cercheremo di rimanere indipendenti da uno specifico linguaggio di programmazione, anche se questo non sarà sempre possibile. Il linguaggio al quale maggiormente ci ispireremo sarà Java, per la coerenza del suo progetto, che ci permette di discutere dei concetti (e non dei dettagli linguistici) in modo più chiaro e sintetico di altri linguaggi.

## 12.1 Limiti dei tipi di dato astratti

I tipi di dato astratti (o anche, per quel che ci interessa qui, l'uso dei moduli per ottenere astrazione sui dati) sono un meccanismo che garantisce in modo pulito ed efficace l'incapsulamento e l'occultamento dell'informazione. In particolare,

hanno la bella caratteristica di riunire in un unico costrutto sia i dati che i modi legali per manipolarli: si tratta di un principio metodologico molto importante per lo sviluppo del software. Questi obiettivi, tuttavia, sono ottenuti al prezzo di una certa rigidità d'uso, che si manifesta soprattutto quando si voglia estendere o riusare un'astrazione. Discuteremo di questi problemi attraverso un semplice esempio, certo non realistico, ma che mette in luce le questioni più importanti.

Per non dover appesantire la notazione con l'uso di passaggi per riferimento, è conveniente porci in un linguaggio con modello delle variabili a riferimento. Il seguente ADT realizza un semplice contatore:

```
abstype Counter{
    type Counter = int;
    signature
        void reset(Counter x);
        int get(Counter x);
        void inc(Counter x);
    operations
        void reset(Counter x) {
            x = 0;
        }
        int get(Counter c) {
            return x;
        }
        void inc(Counter c) {
            x = x+1;
        }
}
```

La rappresentazione concreta del tipo Counter è il tipo degli interi; le sole operazioni possibili sono l'azzeramento di un contatore, la lettura del suo valore, il suo incremento.

Vogliamo ora definire un contatore arricchito di qualche nuova operazione; ad esempio, vogliamo tener conto di quante volte abbiamo chiamato l'operazione reset su un dato contatore. Abbiamo due possibilità: una prima è quella di definire *ex novo* un ADT, per molte cose simile al precedente, con l'aggiunta della nuova operazione:

```
abstype NewCounter1{
    type NewCounter1 = struct{
        int c;
        int num_reset = 0;
    }
    signature
        void reset(NewCounter1 x);
        int get(NewCounter1 x);
        void inc(NewCounter1 x);
        int quanti_reset(NewCounter1 x);
    operations
        void reset(NewCounter1 x) {
            x.c = 0;
            x.num_reset = x.num_reset+1;
        }
        int get(NewCounter1 x) {
```

```
        return x.c;
    }
    void inc(NewCounter1 x) {
        x.c = x.c+1;
    }
    int quanti_reset(NewCounter1 x) {
        return x.num_reset;
    }
}
```

Si tratta di una soluzione accettabile quanto a encapsulamento, ma ci ha costretto a ridefinire le operazioni che già avevamo definito per un contatore semplice (si tratta di operazioni con lo stesso nome, ma con tipo dell'argomento diverso: sono dunque nomi overloaded che il compilatore distinguerà in base al contesto). In questo esempio scolastico si tratta di poche linee di codice, ma l'aggravio in una situazione reale può divenire assai significativo<sup>1</sup>. Più importante ancora è ciò che si presenta quando, per qualche motivo, si voglia cambiare l'implementazione di un contatore semplice. Non essendoci alcuna relazione tra un Counter e un NewCounter1, tali modifiche non si ripercuotono su un nuovo contatore: un NewCounter1 continua a funzionare perfettamente. Ma se la modifica era dettata da motivi di utilità o di efficienza (per esempio, abbiamo scoperto un modo astutissimo per implementare una inc), tale modifica deve essere riportata manualmente nella definizione di un NewCounter1, con i noti problemi che ciò comporta (localizzare tutti i posti dove si è usata una variante della vecchia inc, non introdurre qualche errore sintattico ecc.).

La seconda possibilità che abbiamo è quella di sfruttare un Counter per definire un contatore arricchito:

```
abstype NewCounter2{
    type NewCounter2 = struct{
        Counter c;
        int num_reset = 0;
    }
    signature
        void reset(NewCounter2 x);
        int get(NewCounter2 x);
        void inc(NewCounter2 x);
        int quanti_reset(NewCounter2 x);
    operations
        void reset(NewCounter2 x) {
            reset(x.c);
            x.num_reset = x.num_reset+1;
        }
        int get(NewCounter2 x) {
            return get(x.c);
        }
        void inc(NewCounter2 x) {
```

<sup>1</sup> Si tratta non solo di aggravio durante la scrittura del programma, ma anche nelle dimensioni del codice prodotto: ci sono *due* copie di ogni operazione.

```

    }
    int quanti_reset(NewCounter2 x) {
        return x.num_reset;
    }
}

```

La soluzione è chiaramente migliore della precedente: le operazioni che non devono essere modificate sono solo richiamate dall'interno di NewCounter2 (col solito overloading dei nomi), cosicché l'ultimo problema menzionato precedentemente è risolto. Rimane l'inconveniente di effettuare le chiamate esplicitamente anche per le operazioni (come `get` e `inc`) che non subiscono alcuna modifica: sarebbe preferibile avere un meccanismo automatico col quale *ereditare* da Counter l'implementazione di queste operazioni.

Restano dei problemi, tuttavia, per trattare in modo uniforme i valori di Counter e quelli di NewCounter2. Supponiamo, infatti, di avere a che fare con una serie di contatori, alcuni semplici, altri arricchiti, e di volerli riportare tutti al valore iniziale. Per fissare le idee, possiamo immaginare di disporre di un array di contatori e di voler ripristinare il valore iniziale di ogni elemento di questo array. Un primo problema sorge immediatamente con il tipo di questo vettore. Se lo dichiariamo come

```
Counter V[100];
```

non possiamo memorizzarvi dei NewCounter2, perché i due tipi sono distinti; la stessa cosa succede dichiarando l'array di tipo NewCounter2. Per risolvere il problema abbiamo bisogno di una nozione di compatibilità tra i due tipi. Ricordiamo che tra le varie forme di compatibilità che abbiamo discusso nel Paragrafo 10.6, troviamo la seguente:

T è compatibile con S quando tutte le operazioni sui valori di S sono possibili anche sui valori di T.

Siamo esattamente in un caso del genere: tutte le operazioni su un Counter sono possibili anche su un NewCounter2 (sul quale invece possiamo applicare un'operazione in più). Non è dunque del tutto insensato richiedere che un ipotetico linguaggio che voglia rilassare la rigidità degli ADT ammetta questa forma di compatibilità. Supponiamo dunque che NewCounter2 sia compatibile con Counter e sia pertanto lecito avere un vettore dichiarato come Counter V[100] nel quale sono memorizzati sia contatori semplici sia contatori estesi. Veniamo ora al nostro scopo, che è quello di ripristinare il valore iniziale di tutti questi contatori. L'idea ovvia è

```
for (int i=1; i<100; i=i+1)
    reset(V[i]);
```

che non pone problemi di tipo: il risolutore dell'overloading interpreta il corpo come una chiamata all'operazione `reset` di Counter. Se tutto va bene per il controllo dei tipi, lo stesso non si può dire per lo stato dei NewCounter2 memorizzati nell'array. Ci aspetteremmo che i loro campi `num_reset` siano stati incrementati di uno, ma così non è, perché non è stata eseguita l'operazione `reset`

definita per NewCounter2, ma quella di Counter. La compatibilità ha risolto il problema di manipolare in modo uniforme i valori dei due tipi, ma in un certo senso ha distrutto l'incapsulamento, permettendo di applicare ad un valore di tipo NewCounter2 un'operazione che non è corretta rispetto alla specifica dell'ADT. Un momento di riflessione mostra come il problema nasca dalla risoluzione statica dell'overloading di `reset`: se potessimo decidere *dinamicamente* quale `reset` applicare, in dipendenza del tipo "effettivo" dell'argomento (cioè del tipo dell'elemento memorizzato nell'array prima che vi sia applicata la coercizione connessa alla compatibilità), anche questo problema sarebbe risolto.

### 12.1.1 Un primo bilancio

I tipi di dato astratti garantiscono encapsulamento ed occultamento dell'informazione, ma sono rigidi da impiegare in un progetto di una certa complessità. Da quanto abbiamo appena detto non è irragionevole prevedere costrutti che:

- permettano l'*incapsulamento* e l'occultamento dell'informazione;
- siano dotati di un meccanismo che, sotto certe condizioni, permetta di *ereditare* l'implementazione di determinate operazioni da altri costrutti analoghi;
- siano inquadrati in una nozione di *compatibilità* definita in termini delle operazioni ammissibili su un certo costrutto;
- permettano la *selezione dinamica* delle operazioni, in funzione del "tipo" effettivo (o dell'implementazione) dell'argomento cui vengono applicate.

Queste quattro richieste sono soddisfatte nel paradigma orientato agli oggetti. Anzi, possiamo prenderle come caratteri essenziali di questo paradigma, che separano un linguaggio orientato agli oggetti da uno che non lo è. Le discuteremo approfonditamente dopo aver introdotto alcuni concetti di base e fissato la terminologia.

## 12.2 Concetti fondamentali

Affronteremo in questo paragrafo i concetti fondamentali che strutturano il paradigma orientato agli oggetti. Inizieremo con l'introduzione della terminologia e dei concetti linguistici macroscopici (oggetti, classi, metodi, ridefinizione ecc.) per poi riprendere i quattro aspetti che abbiamo elencato nel Paragrafo 12.1.1, con l'obiettivo di discuterli più approfonditamente. Li affronteremo nell'ordine seguente:

1. encapsulamento ed astrazione;
2. sottotipi, cioè una relazione di compatibilità basata sulle funzionalità di un oggetto;
3. ereditarietà, cioè la possibilità di riusare l'implementazione di un metodo definito precedentemente per un altro oggetto o per un'altra classe;
4. selezione dinamica dei metodi.

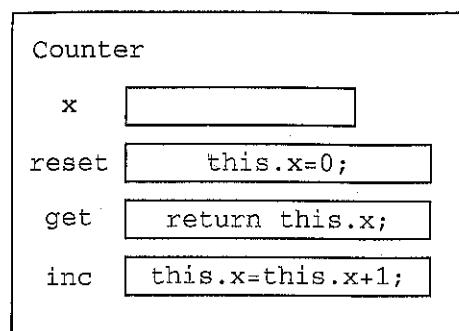


Figura 12.1 Un oggetto per un contatore.

Si tratta di meccanismi distinti, presenti singolarmente in molti linguaggi di altri paradigmi. Come si vedrà, è soprattutto la loro interazione a rendere il paradigma orientato agli oggetti così attraente per lo sviluppo di progetti software anche di grandi dimensioni.

## 12.2.1 Oggetti

Il costrutto principale di ogni linguaggio orientato agli oggetti è (ovviamente...) quello di oggetto: una capsula contenente sia dati che operazioni per manipolarli e che fornisce all'esterno un'interfaccia attraverso la quale l'oggetto è accessibile. L'idea metodologica, che gli oggetti condividono con gli ADT, è che i dati devono "stare insieme" alle operazioni per la loro manipolazione. C'è tuttavia un'importante differenza concettuale con gli ADT (che poi si tradurrà anche in una sostanziale diversità di notazione): sebbene nella definizione di un ADT dati e operazioni stiano assieme, quando dichiariamo una variabile di un tipo di dato astratto, quella variabile rappresenta solo i dati, che possiamo manipolare tramite le operazioni. Nel nostro esempio dei contatori, potremmo dichiarare un contatore come `Counter c` e poi incrementarlo: `c.inc()`. *Ciascun* oggetto, invece, è un contenitore che (almeno concettualmente) incapsula sia dati che operazioni. Un oggetto contatore, definito sullo stesso modello della definizione che abbiamo dato per `Counter`, potrebbe essere rappresentato come in Figura 12.1.

La figura suggerisce come non sia sbagliato immaginarsi un oggetto come un record: alcuni campi corrispondono a dati (modificabili), come il campo `x`; altri campi corrispondono alle operazioni che possono manipolare i dati, come `reset`, `get` e `inc`.

Le operazioni sono chiamate *metodi* (o campi funzione, o *member function*) e possono accedere ai dati contenuti nell'oggetto, che sono detti *variabili di istanza* (o *data member*, o campi). L'esecuzione di un'operazione è invocata mandando all'oggetto un *messaggio* che consiste nel nome del metodo da eseguire e nei suoi

eventuali parametri<sup>2</sup>. Per quest'invocazione useremo la notazione di C++ e Java, che rinforza l'analogia con un record:

`oggetto.metodo(parametri)`

Nel seguito leggeremo spesso la precedente notazione come "invocazione del metodo `metodo` (con parametri `parametri`) sull'oggetto `oggetto`", invece che "invio del messaggio `metodo(parametri)` all'oggetto `oggetto`" come sarebbe formalmente più preciso.

Un aspetto non sempre ovvio è che l'oggetto che riceve il messaggio è contemporaneamente anche un parametro (implicito) del metodo invocato: per incrementare un oggetto contatore o analogo a quello della Figura 12.1 scriveremmo `o.inc()` ("con il messaggio `inc` chiediamo all'oggetto `o` di applicare a se stesso il metodo `inc`"). Anche i dati sono accessibili dall'esterno con lo stesso meccanismo (se non nascosti dalla capsula, ovviamente): se un oggetto `o` ha una variabile di istanza `v`, con `o.v` richiediamo ad `o` il valore di `v`. Pur nell'uniformità della notazione, osserviamo subito che, in generale, l'accesso ai dati è un meccanismo distinto dall'invocazione di un metodo: l'invocazione di metodo comporta (o può comportare) una *selezione dinamica* di quale metodo debba essere eseguito, mentre l'accesso ad un dato è statico, sebbene vi siano eccezioni a questo criterio.

Il grado di opacità della capsula è definito al momento della creazione dell'oggetto stesso: i dati possono o meno essere accessibili direttamente dall'esterno (ad esempio, i dati potrebbero essere inaccessibili direttamente e l'oggetto fornire invece degli "osservatori"; si ricordi il riquadro di pag. 352), alcune operazioni possono essere visibili ovunque, altre solo per alcuni oggetti, altre infine completamente private, cioè disponibili solo all'interno dell'oggetto stesso.

Insieme agli oggetti, i linguaggi di questo paradigma mettono a disposizione anche dei *meccanismi di organizzazione* degli oggetti stessi, che permettono di raggruppare gli oggetti con la stessa struttura (o struttura simile). Sebbene sia concettualmente lecito immaginare che *ogni* oggetto contenga davvero i propri dati e i propri metodi, questo risulterebbe in un enorme spreco. In uno stesso programma saranno usati molti oggetti che si differenziano tra loro solo per il valore dei dati, e non per la loro struttura o per i metodi (per esempio, molti contatori, tutti analoghi a quello della Figura 12.1). Senza un principio di organizzazione che espliciti la somiglianza di tutti questi oggetti, il programma perderebbe chiarezza espositiva e documentale. Inoltre, da un punto di vista implementativo è evidente che sarebbe opportuno che il codice di ogni metodo fosse memorizzato una sola volta, invece di essere replicato all'interno di tutti gli oggetti simili. Per rispondere a questi problemi, ogni linguaggio orientato agli oggetti si basa su qualche principio di organizzazione. Tra questi principi, quello di gran lunga più noto e più usato è quello delle *classi*, sebbene vi sia tutta una famiglia di linguaggi orientati agli oggetti senza classi (cui accenneremo brevemente nel riquadro a pag. 369).

<sup>2</sup>Questa è la significativa terminologia di Smalltalk; C++ esprime la stessa cosa dicendo: chiamata del campo funzione (*member function*) di un oggetto.

## 12.2.2 Classi

Una classe è un modello di un insieme di oggetti: stabilisce quali sono i suoi dati (quanti, di quale tipo, con quale visibilità) e fissa nome, segnatura, visibilità e implementazione dei suoi metodi. In un linguaggio con classi, ogni oggetto “appartiene” ad (almeno) una classe, nel senso che la struttura dell’oggetto corrisponde alla struttura fissata dalla classe. Ad esempio, un oggetto come quello della Figura 12.1 potrebbe essere un’istanza della seguente classe:

```
class Counter{
    private int x;
    public void reset(){
        x = 0;
    }
    public int get(){
        return x;
    }
    public void inc(){
        x = x+1;
    }
}
```

Si osservi come la classe riporta l’implementazione dei tre metodi, che sono dichiarati `public`, cioè visibili da chiunque, mentre il dato `x` è `private`, cioè inaccessibile dall’esterno dell’oggetto stesso (ma accessibile nell’implementazione dei metodi di questa classe).

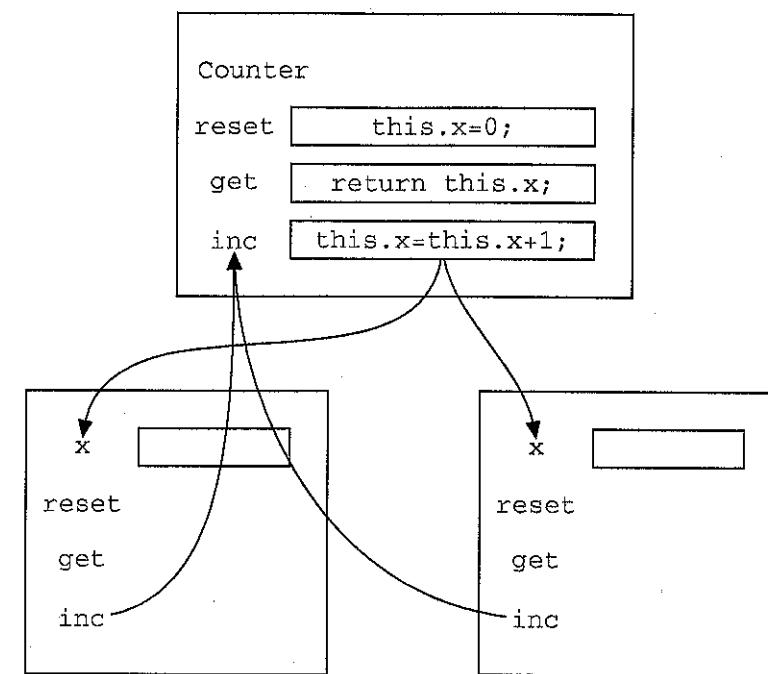
Gli oggetti sono creati dinamicamente mediante *istanziazione* di una classe: viene allocato uno specifico oggetto con la struttura stabilita dalla definizione della classe. Quest’operazione varia in modo essenziale da un linguaggio ad un altro e dipende dallo *status* linguistico delle classi. In Simula una classe è una procedura che restituisce un (puntatore ad un) RDA contenente variabili locali e definizioni di funzioni (dunque una chiusura, che costituisce l’oggetto istanza della classe); in Smalltalk una classe è linguisticamente un oggetto, che serve da schema per la definizione dell’implementazione di un insieme di oggetti; in C++ e Java una classe corrisponde ad un tipo e tutti gli oggetti istanza di una classe A sono valori di tipo A. Adottando questo punto di vista, e prendendo ad esempio Java, con il suo modello a riferimento per le variabili, potremo creare un oggetto contatore:

```
Counter c = new Counter();
```

Il nome `c`, di tipo `Counter`, è legato ad un nuovo oggetto istanza di `Counter`. Possiamo creare un nuovo oggetto, distinto dal precedente, del quale tuttavia condivide la struttura, e legarlo ad un altro nome:

```
Counter d = new Counter();
```

A destra del simbolo di assegnamento possiamo distinguere due azioni distinte: la creazione dell’oggetto (allocazione della memoria necessaria, costrutto `new`) e



**Figura 12.2** L’implementazione dei metodi risiede nella classe.

la sua inizializzazione (invocazione del *costruttore* della classe, rappresentato dal nome della classe con le parentesi, sul quale ritorneremo tra poco)<sup>3</sup>.

Possiamo senz’altro supporre che il codice dei metodi sia memorizzato una sola volta nella classe, e che quando ad un oggetto viene richiesto di eseguire un determinato metodo, questo venga ricercato nella classe di cui è istanza. Affinché ciò possa accadere, il codice del metodo deve accedere correttamente alle variabili di istanza, che sono distinte per ogni oggetto e che dunque non sono memorizzate insieme alla classe, ma all’interno dell’istanza, come è indicato schematicamente nella Figura 12.2. Nella figura i metodi della classe `Counter` si riferiscono alle variabili di istanza attraverso il nome `this`. Abbiamo già osservato che quando un oggetto riceve un messaggio che richiede l’esecuzione di un metodo, l’oggetto stesso è un parametro implicito del metodo: quando nel corpo del metodo si fa riferimento alle variabili di istanza c’è un隐式 riferimento all’oggetto corrente che sta eseguendo il metodo. Da un punto di vista linguistico, l’oggetto corrente viene di solito denotato da un nome particolare, usualmente `self` o `this`. Ad esempio, la definizione del metodo `inc` poteva essere scritta, esplicitando il

<sup>3</sup>In C++, a differenza di Java, è possibile creare un oggetto anche senza invocare un costruttore; C++ permette infatti di definire oggetti anche sulla pila (si veda la pagina 368).

riferimento implicito all'oggetto corrente:

```
public void inc(){
    this.x = this.x+1;
}
```

Nel caso dell'invocazione di un metodo tramite `this`, il legame tra questo nome e l'oggetto cui si riferisce è determinato solo dinamicamente: si tratta di un aspetto importante di questo paradigma che discuteremo meglio nel Paragrafo 12.2.6.

Alcuni linguaggi permettono di avere variabili o metodi associati alla classe (e non alle loro istanze): sono detti variabili e metodi di classe, o *statici*: le variabili statiche sono memorizzate assieme alla classe, e i metodi statici non possono ovviamente fare riferimento a `this` nel loro corpo, dato che non c'è alcun oggetto corrente.

**Oggetti nello heap o nello stack** Tutti i linguaggi orientati agli oggetti permettono di creare oggetti dinamicamente. Dove tali oggetti vengano creati, dipende dal linguaggio. La soluzione più comune è quella di allocare un oggetto sullo heap e di accedere all'oggetto stesso mediante un riferimento (che sarà un vero e proprio puntatore nei linguaggi che li possiedono, e sarà invece una variabile qualora il linguaggio abbia scelto un modello a riferimento). Alcuni linguaggi permettono allocazione e deallocazione esplicita degli oggetti sullo heap (C++ è tra questi), altri invece, e sono probabilmente la maggioranza, optano per un garbage collector.

Non è molto diffusa la possibilità di creare oggetti sulla pila, come normali variabili. C++ è un linguaggio con questa caratteristica: quando viene elaborata la dichiarazione di una variabile di tipo classe viene contestualmente creato e inizializzato un oggetto di quel tipo, che viene assegnato come valore di quella variabile<sup>4</sup>. Le due azioni che concorrono alla creazione di un oggetto (allocazione e inizializzazione) avvengono in tal caso implicitamente, senza indicazione specifica di invocazione del costruttore. Alcuni linguaggi, infine, permettono di creare oggetti sulla pila e lasciarli non inizializzati.

Nel nostro pseudolinguaggio, assumeremo di creare oggetti esplicitamente sullo heap e di avere un modello delle variabili a riferimento.

### 12.2.3 Incapsulamento

L'incapsulamento e l'occultamento dell'informazione sono uno dei cardini dell'astrazione sui dati. Da un punto di vista linguistico, non c'è molto da aggiungere a quello che abbiamo già detto: ogni linguaggio permette di definire un oggetto nascondendone una parte (che può essere costituita sia da dati che metodi). Di ogni classe esistono dunque almeno due *viste*: quella privata e quella pubblica.

<sup>4</sup>La situazione è ovviamente diversa quando venga dichiarata una variabile di tipo puntatore ad un tipo classe: in questo caso non viene creato alcun oggetto (a meno di un'esplicita richiesta in tal senso).

### Linguaggi basati su delega

Esistono anche linguaggi orientati agli oggetti senza classi. In questi linguaggi *basati su delega* (o su *prototipi*) il meccanismo di organizzazione (vedi la fine del Paragrafo 12.2.1) non è la classe, ma la *delega*, cioè la possibilità per un oggetto di demandare ad un altro oggetto (il suo *genitore*) l'esecuzione di un metodo. Di questi linguaggi il capostipite è Self, sviluppato presso Xerox PARC e Stanford verso la fine degli anni ottanta; altri linguaggi basati su delega sono Dylan, sviluppato per la programmazione dei PDA Newton di Apple, e Javascript, progettato per essere inglobato in una delle prime versioni di Netscape.

I campi di un oggetto possono contenere sia valori (dati semplici o altri oggetti), sia metodi (cioè codice), sia riferimenti ad altri oggetti (il *genitore* dell'oggetto). Un oggetto può essere creato dal nulla oppure, più comunemente, mediante copia (o *clonazione*) di un altro oggetto, il suo *prototipo*. I prototipi svolgono il ruolo metodologico delle classi, visto che fungono da modello per la struttura e il funzionamento di altri oggetti, ma non sono oggetti speciali: sono linguisticamente oggetti ordinari che, tuttavia, vengono usati non per la computazione ma come modello. Al momento della clonazione di un prototipo, la copia mantiene un riferimento al prototipo come proprio genitore.

Quando un oggetto riceve un messaggio e non ha un campo con quel nome, passa il messaggio al suo genitore, sul quale il processo si ripete. Quando il messaggio raggiunge un oggetto che lo comprende, il codice associato al metodo viene eseguito; il linguaggio (con meccanismi assai diversi da un linguaggio all'altro) assicura che il riferimento a `self` è correttamente quello all'oggetto che originariamente aveva ricevuto il messaggio. In genere i dati di un oggetto sono del tutto equiparati ai metodi: accedere ad un dato (anche di `self`) corrisponde all'invio di un messaggio, al quale l'oggetto risponde restituendo il valore del campo.

Il meccanismo della delega (e l'unificazione di codice e dati) rende l'ereditarietà più potente: è possibile e naturale creare oggetti che condividono porzioni dei dati (in un linguaggio basato su classi ciò è possibile solo mediante dati statici associati alla classe, ma il meccanismo è troppo "statico" per essere utilizzabile con profitto). Inoltre un oggetto può cambiare dinamicamente il riferimento al proprio genitore, e dunque modificare il proprio comportamento.

Nella vista privata tutto è visibile: è la forma di accesso possibile all'interno della classe stessa (da parte dei suoi metodi). Nella vista pubblica invece, sono visibili solo le informazioni esplicitamente esportate: chiamiamo *interfaccia* di una classe il complesso delle informazioni pubbliche, per analogia a quanto abbiamo già fatto per gli ADT<sup>5</sup>. Vedremo che il meccanismo dell'ereditarietà suggerirà

<sup>5</sup>Nella terminologia dei singoli linguaggi, il termine "interfaccia" assume in genere altri significati. Si faccia attenzione, in particolare, a non confondere questo nostro uso di "interfaccia" col significato che gli dà Java, per il quale è un particolare costrutto del linguaggio (che consiste in

l'introduzione di una terza vista, quella da parte delle classi che ereditano.

Come si vede, a questo livello non emergono grandi differenze con le altre forme di astrazione sui dati che abbiamo già discusso. L'incapsulamento possibile con gli oggetti, tuttavia, è assai più flessibile e soprattutto estendibile di quello possibile con gli ADT; ma questo sarà chiaro solo al termine di questa nostra discussione.

### 12.2.4 Sottotipi

Ad una classe può essere fatto corrispondere in modo naturale l'insieme degli oggetti che sono istanza di quella classe: quest'insieme di oggetti è il *tipo associato* a quella classe. Nei linguaggi tipizzati questa relazione è esplicita: una definizione di classe introduce anche la definizione di un tipo, i cui valori sono le istanze della classe. In linguaggi senza tipi (come Smalltalk), la corrispondenza è solo concettuale e implicita.

Tra i tipi così ottenuti è definita una relazione di compatibilità (Paragrafo 10.6) in termini delle operazioni possibili sui valori di un tipo: il tipo associato alla classe *T* è un sottotipo di *S* quando ogni messaggio compreso (cioè che può essere ricevuto senza generare errore) dagli oggetti di *S* è compreso anche dagli oggetti di *T*. Se ci rappresentiamo un oggetto come un record che contiene dati e funzioni<sup>6</sup>, questa relazione di sottotipo corrisponde al fatto che *T* è un tipo record che contiene tutti i campi di *S*, più eventualmente altri. Più precisamente, per tener conto del fatto che alcuni campi dei due tipi potrebbero essere privati, si dovrà dire che *T* è un sottotipo di *S* quando l'interfaccia di *S* è un sottoinsieme dell'interfaccia di *T* (si osservi l'inversione: un sottotipo si ottiene quando si ha un'interfaccia più grande).

Certi linguaggi (come C++ e Java) adottano per i tipi classe un'equivalenza per nome (Paragrafo 10.5) che mal si adatta ad una compatibilità completamente strutturale. In tali linguaggi, non è pertanto la sola proprietà strutturale tra le interfacce a definire la relazione di sottotipo, ma essa deve essere introdotta esplicitamente dal programmatore. È questo il ruolo della definizione di *sottoclasse*, o *classe derivata*, che nel nostro pseudolinguaggio indicheremo con il costrutto neutro *extending*<sup>7</sup>:

```
class NamedCounter extending Counter{
    private String nome;
    public void set_name(String n){
        nome = n;
    }
    public String get_name(){
        return nome;
    }
}
```

una sorta di classe nei quali compaiono solo i nomi e le segnature dei metodi, ma non le loro implementazioni).

<sup>6</sup>Cioè come una chiusura...

<sup>7</sup>Al posto di *extending*, in C++ scriveremmo “: public”, mentre in Java *extends* oppure *implements* a seconda se stiamo estendendo una classe o un'interfaccia.

```
}
```

La classe *NamedCounter* è una sottoclasse di *Counter* (che a sua volta è la superclasse di *NamedCounter*)<sup>8</sup>, cioè il tipo *NamedCounter* è un sottotipo di *Counter*. Le istanze di *NamedCounter* contengono tutti i campi di un *Counter* (anche i campi privati, ma questi sono inaccessibili nella sottoclasse), oltre ad avere i nuovi campi introdotti con questa definizione. In questo modo si garantisce la compatibilità strutturale (una sottoclasse è esplicitamente derivata dalla sua superclasse), ma questa è esplicitamente “nominata” nel programma.

**Ridefinizione di un metodo** Nel semplice esempio di *NamedCounter* la sottoclasse si limita ad estendere l'interfaccia della superclasse. Una fondamentale caratteristica del meccanismo di sottoclasse è costituita dalla possibilità che una sottoclasse modifichi la definizione (l'implementazione) di un metodo presente nella superclasse. Questo meccanismo si chiama *ridefinizione di metodo* (*method overriding*)<sup>9</sup>. I nostri contatori estesi del Paragrafo 12.1 possono essere definiti come una sottoclasse di *Counter*:

```
class NewCounter extending Counter{
    private int num_reset = 0;
    public void reset(){
        x = 0;
        num_reset = num_reset + 1;
    }
    public int quanti_reset(){
        return num_reset;
    }
}
```

La classe *NewCounter* contemporaneamente estende l'interfaccia di *Counter* con nuovi campi e ridefinisce il metodo *reset*. Un messaggio *reset* inviato ad un'istanza di *NewCounter* causerà l'invocazione della nuova implementazione.

**Shadowing** Oltre a modificare l'implementazione di un metodo, una sottoclasse può anche ridefinire una variabile d'istanza (o campo) definita in una superclasse. Questo meccanismo si chiama *shadowing*, termine che possiamo approssimativamente tradurre in mascheramento. Lo shadowing è sostanzialmente diverso dal meccanismo dell'overriding per motivi implementativi che vedremo più avanti. Per adesso osserviamo solo che possiamo ridefinire in una sottoclasse una variabile d'istanza con lo stesso nome e lo stesso tipo di quella che appare in

<sup>8</sup>Stiamo usando la terminologia più comune, anche se sarebbe più corretto dire “sovraclasse”. In C++ *Counter* è la classe *base* e *NamedCounter* è la classe *derivata*.

<sup>9</sup>La ridefinizione di metodo è un meccanismo che interagisce in modo delicato con altri aspetti del paradigma orientato agli oggetti, in particolare con la selezione dinamica dei metodi: per il momento accenniamo solo a questa possibilità, riservandoci di tornare sulla questione a tempo debito.

### Sottotipi in Smalltalk?

Tra le caratteristiche essenziali del paradigma orientato agli oggetti abbiamo incluso la presenza di sottotipi. D'altra parte esistono linguaggi orientati agli oggetti che non hanno tipi (almeno non hanno tipi su cui si possano eseguire controlli significativi), e tra questi uno dei grandi pionieri di questo paradigma, Smalltalk. Come possiamo avere sottotipi in un linguaggio senza tipi?

Abbiamo dato la definizione di sottotipo con una certa attenzione in modo da includere anche il caso di Smalltalk e altri linguaggi senza tipi. Si ricordi che *T* è un sottotipo di *S* quando l'interfaccia di *S* è un sottinsieme dell'interfaccia di *T*, cioè quando possiamo liberamente (cioè senza generare errori) usare un oggetto della classe *T* al posto di un oggetto della classe *S*. Questa definizione operazionale, che si esprime mediante la sostituzione di oggetti al posto di altri oggetti, ha perfettamente senso anche in un contesto senza tipi, una volta che si associa ad una classe *T* l'insieme delle sue istanze (il suo "tipo").

D'altra parte, non essendoci nel linguaggio alcuna nozione linguistica di sottotipo, è solo nella riflessione del progettista che si può evidenziare la presenza o meno di tale relazione. Oltre tutto, in Smalltalk la definizione di una sottoclasse *non* genera sempre un sottotipo, perché in Smalltalk una sottoclasse può anche *rimuovere* un metodo della superclasse.

una superclasse. Potremmo ad esempio modificare i nostri contatori estesi usando la seguente sottoclasse di NewCounter dove, per qualche (strano) motivo, il valore iniziale di num\_reset è inizializzato a 2 e viene incrementato di 2 ogni volta:

```
class NewCounterPari extending NewCounter{
    private int num_reset = 2;
    public void reset(){
        x = 0;
        num_reset = num_reset + 2;
    }
    public int quanti_reset(){
        return num_reset;
    }
}
```

Seguendo l'usuale nozione di visibilità di un linguaggio a blocchi, ogni riferimento a num\_reset all'interno di NewCounterPari si riferisce alla variabile locale (inizializzata a 2) e non a quella dichiarata in NewCounter. Un messaggio reset inviato ad un'istanza di NewCounterPari causerà l'invocazione della nuova implementazione di reset che a sua volta userà il nuovo campo num\_reset. Tuttavia, come vedremo più avanti, vi è una profonda differenza, sia a livello semantico che a livello implementativo, fra overriding e shadowing.

**Classi astratte** Per semplicità di esposizione abbiamo introdotto il tipo associato ad una classe come l'insieme delle sue istanze. Molti linguaggi, tuttavia,

permettono la definizione di classi che non hanno istanze, perché la classe manca dell'implementazione di qualche metodo. In tali classi figura solo il nome ed il tipo (cioè la segnatura) di uno o più metodi, senza che sia data alcuna implementazione. Classi del genere sono dette *classi astratte*. Le classi astratte servono a fornire interfacce, di cui sarà data implementazione in qualche sottoclasse che ridefinirà (cioè, in questo caso, definirà per la prima volta) il metodo di cui manca l'implementazione<sup>10</sup>. Anche alle classi astratte corrispondono tipi, e il meccanismo con cui si fornisce implementazione ai loro metodi genera sottotipi.

**La relazione di sottotipo** I linguaggi vietano in genere che vi siano cicli nella relazione di sottotipo: non può accadere che due tipi *A* e *B* siano mutuamente sottotipi tra loro, a meno che *A* e *B* coincidano. La relazione di sottotipo è dunque un ordine parziale. In molti linguaggi quest'ordine ha un elemento massimo: un tipo di cui tutti gli altri tipi (associati a classi) sono sottotipi. Indicheremo tale tipo con Object. I messaggi che un'istanza di Object accetta sono assai limitati (si tratta di un oggetto della massima generalità): troviamo in genere un metodo di clonazione che restituisca una copia dell'oggetto cui il messaggio è inviato, un metodo di uguaglianza e poco più. Inoltre, alcuni di tali metodi potrebbero essere astratti in Object e dunque necessitare di essere ridefiniti prima di poter esser usati.

Si osservi che non è garantito che, dato un tipo *A*, esista un *unico* tipo *B* "supertipo"<sup>11</sup> immediato" di *A*. Nel seguente semplice esempio:

```
abstract class A{
    public int f();
}

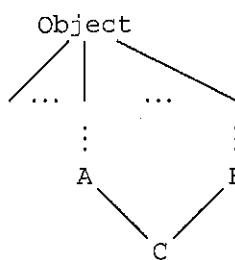
abstract class B{
    public int g();
}

class C extending A,B{
    private x = 0;
    public int f(){
        return x;
    }
    public int g(){
        return x+1;
    }
}
```

il tipo *C* è sottotipo sia di *A* che di *B* e non c'è alcun altro tipo compreso tra *C* e i due tipi menzionati. La gerarchia dei sottotipi non è pertanto un albero, in generale, ma solo un grafo orientato aciclico, si veda la Figura 12.3.

<sup>10</sup>Un metodo di cui si dà solo la segnatura si chiama in Java metodo *astratto*; in C++ è un membro funzione puramente virtuale (*pure virtual member function*).

<sup>11</sup>Anche qui, come nel caso delle classi, sarebbe più corretto usare il termine "sovratipo" invece che "supertipo"; ci uniformiamo però all'uso più comune.



**Figura 12.3** La relazione di sottotipo *non* è un albero.

**Costruttori** Abbiamo già visto che un oggetto è una struttura complessa, che comprende dati e codice impacchettati assieme. La creazione di un oggetto è dunque anch'essa un'operazione di una certa complessità che consta di due azioni distinte: occorre allocare la memoria necessaria (sullo heap o sulla pila) e inizializzare correttamente i dati. Quest'ultima azione è compiuta dal *costruttore* della classe, cioè da codice associato alla classe di cui il linguaggio garantisce l'esecuzione al momento in cui un'istanza della classe viene creata. Il meccanismo dei costruttori è di una qualche complessità, perché i dati di un oggetto consistono non solo di quelli esplicitamente dichiarati nella classe di cui si sta creando l'istanza, ma anche dei dati dichiarati nelle superclassi. Inoltre è in genere permessa la dichiarazione di più di un costruttore associato ad una classe (per permettere inizializzazioni diverse). Si pongono dunque una serie di questioni legate ai costruttori, che possiamo riassumere come segue.

- **Scelta del costruttore:** Qualora il linguaggio permetta più di un costruttore associato alla stessa classe, come viene scelto quello da utilizzare al momento di una specifica creazione di un oggetto? In alcuni linguaggi (tra i quali C++ e Java), il nome del costruttore coincide col nome della classe; costruttori multipli hanno tutti lo stesso nome e devono essere distinti per il tipo o il numero degli argomenti (sono dunque overloaded, con risoluzione statica). Siccome C++ permette la creazione implicita di oggetti sulla pila, vi sono meccanismi specifici per la selezione dell'opportuno costruttore da utilizzare in tal caso<sup>12</sup>. Altri linguaggi permettono al programmatore di scegliere liberamente il nome dei costruttori (ma i costruttori rimangono sintatticamente distinti dai metodi ordinari), richiedendo poi che ad ogni operazione di creazione (il nostro new) sia sempre associato un costruttore.
- **Concatenamento dei costruttori:** Come e quando avviene l'inizializzazione di quella parte di un oggetto che proviene dalle superclassi? Alcuni linguaggi si limitano ad eseguire il costruttore della classe di cui si sta creando l'istanza; se

<sup>12</sup>Il caso di un costruttore che prende come argomento un solo oggetto (un costruttore di copia, nel gergo C++) è particolarmente delicato e fonte di sottili errori di programmazione.

il programmatore intende chiamare i costruttori delle superclassi, lo deve fare esplicitamente. Altri linguaggi (e tra questi C++ e Java) garantiscono invece che al momento in cui un oggetto è inizializzato viene invocato anche il costruttore della superclasse (*constructor chaining*), prima di ogni altra azione compiuta dal costruttore specifico della sottoclasse. Ancora una volta vi sono qui molti dettagli che uno specifico linguaggio deve risolvere. Tra questi i due più importanti sono determinare quale costruttore della superclasse invocare e come determinare i suoi argomenti.

### 12.2.5 Ereditarietà

Abbiamo visto che una sottoclasse può ridefinire un metodo della superclasse. Ma cosa succede quando la sottoclasse *non* lo ridefinisce? In tal caso la sottoclasse *eredita* il metodo dalla superclasse, nel senso che l'implementazione del metodo della superclasse è resa disponibile alla sottoclasse. Ad esempio, NewCounter eredita da Counter il dato x e i metodi inc e get (ma non reset che viene ridefinito).

Più in generale (e per includere nella nostra definizione anche i fenomeni che si presentano in linguaggi orientati agli oggetti senza classi), possiamo caratterizzare l'ereditarietà come un meccanismo che permette la definizione di nuovi oggetti a partire da, e riusando, altri oggetti preesistenti.

L'ereditarietà permette il riuso del codice in un contesto estendibile: modificando l'implementazione di un metodo in una classe si rende automaticamente disponibile tale modifica a tutte le sottoclassi, senza bisogno di alcuna operazione da parte del programmatore.

È importante cogliere la differenza tra la relazione di ereditarietà e quella di sottotipo. La nozione di sottotipo ha a che vedere con la possibilità di usare un oggetto in un altro contesto: è una relazione tra le interfacce di due classi. La nozione di ereditarietà ha a che vedere con la possibilità di riusare del codice che manipola un oggetto: è una relazione tra le implementazioni di due classi. Si tratta di due meccanismi del tutto indipendenti, anche se spesso linguisticamente collegati negli specifici linguaggi orientati agli oggetti: sia C++ che Java hanno costrutti che possono introdurre contemporaneamente entrambe le relazioni tra due classi, ma questo non vuol dire che i due concetti non siano tra loro distinti. Nella letteratura si trova talvolta la distinzione tra *ereditarietà di implementazione* (la nostra ereditarietà) ed *ereditarietà di interfaccia* (la nostra relazione di sottotipo).

**Ereditarietà e visibilità** Abbiamo già osservato come di ogni classe esistono due diverse viste: quella privata e quella pubblica, quest'ultima condivisa da tutti i "clienti" della classe. Una sottoclasse è un particolare cliente delle sue superclassi: usa i metodi della superclasse, ma talvolta, per estendere le funzionalità della superclasse, può avere bisogno di accedere ad alcuni dati non pubblici. Molti linguaggi introducono così una terza vista di una classe: quella che hanno le sot-

### Ereditarietà e sottotipi in Java

La relazione di sottotipo è introdotta in Java sia con la clausola `extends` (che definisce una sottoclasse) che con la clausola `implements`, mediante la quale una classe è dichiarata sottotipo di una o più "interfacce" (per Java una interfaccia è una sorta di classe completamente astratta, dove sono presenti solo nomi e segnature di metodi, senza alcuna implementazione). La relazione di ereditarietà è introdotta con la clausola `extends`, qualora la sottoclasse non ridefinisca un metodo e dunque sfrutti l'implementazione della superclasse. Si noti che non c'è mai ereditarietà *da* un'interfaccia, perché un'interfaccia non ha nulla che si possa ereditare. Il linguaggio vincola ogni classe ad avere un'unica superclasse immediata (cioè una sola superclasse può essere nominata in una `extends`), ma permette che una stessa classe (o interfaccia) "implementi" più interfacce:

```
interface A{
    int f();
}
interface B{
    int g();
}
class C{
    int x;
    int h(){
        return x+2;
    }
}
class D extends C implements A,B{
    int f(){
        return x;
    }
    int g(){
        return x+1;
    }
}
```

Java ha dunque ereditarietà singola: la gerarchia dell'ereditarietà è un albero, strutturato dalle clausole `extends`; inoltre la relazione di ereditarietà è in Java sempre una sottogerarchia della gerarchia di sottotipo.

Una situazione come quella dell'esempio, dove si introduce ereditarietà da una superclasse e, contemporaneamente, si "eredita" da interfacce astratte, viene spesso indicata come ereditarietà *mix-in* (perché i nomi dei metodi astratti dell'interfaccia sono "mescolati" con le implementazioni ereditate). Al solito la terminologia corrente è spesso imprecisa e confonde sottotipi con ereditarietà: in molti manuali (e perfino nella definizione ufficiale di Java...) troviamo scritto che Java ha ereditarietà singola per le classi, ma multipla per le interfacce. Per noi non c'è nessuna "vera" ereditarietà quando le interfacce sono coinvolte.

toclassi. Prendendo il termine da C++ possiamo designarla come la vista *protetta* (`protected`) di una classe<sup>13</sup>.

Se la sottoclasse ha accesso ad alcuni dettagli implementativi della superclasse, la sottoclasse dipende in modo molto stretto dalla superclasse: ogni modifica della superclasse imporrà una modifica della sottoclasse. Dal punto di vista pragmatico, ciò è ragionevole solo se le due classi sono "vicine" tra loro, per esempio appartengono allo stesso "package": tanto più aumenta il grado di accoppiamento di due classi, tanto più il sistema risultante è complesso da modificare e mantenere. D'altra parte, far coincidere la visibilità protetta con la visibilità pubblica può essere troppo restrittivo. Si pensi ad una gerarchia di classi che forniscono strutture dati via via più specializzate: potendo accedere alla rappresentazione della struttura dati la sottoclasse sarà in grado di realizzare le proprie operazioni in modo assai più efficiente.

**Ereditarietà singola e multipla** In alcuni linguaggi una classe può ereditare da una sola superclasse immediata: la gerarchia dell'ereditarietà è dunque un albero e si dice che il linguaggio ha *ereditarietà singola*. Altri linguaggi, invece, permettono che una classe erediti metodi da più di una superclasse immediata: la gerarchia dell'ereditarietà è in tal caso un grafo orientato aciclico e il linguaggio ha *ereditarietà multipla*.

Non sono molti i linguaggi che supportano l'ereditarietà multipla (tra questi C++ e Eiffel), perché pone problemi che non hanno una soluzione elegante, sia da un punto di vista concettuale, sia, soprattutto, da un punto di vista implementativo. I problemi concettuali hanno a che vedere con i conflitti di nomi (*name clash*): si ha conflitto di nomi quando una classe C eredita contemporaneamente da A e B, le quali forniscono entrambe l'implementazione di un metodo con la stessa segnatura. Il seguente è un semplicissimo esempio:

```
class A{
    int x;
    int f(){
        return x;
    }
}
class B{
    int y;
    int f(){
        return y;
    }
}
class C extending A,B{
    int h(){
        return f();
    }
}
```

<sup>13</sup>Anche Java ha un modificatore di visibilità `protected`, che però è più liberale di quello di C++: permette la visibilità a tutto il package e non solo alle sottoclassi.

### Ereditarietà e sottotipi in C++

Come Java, anche in C++ i meccanismi che permettono di definire la relazione di ereditarietà e di sottotipo non sono indipendenti tra loro. La definizione di classe derivata (il termine C++ per sottoclasse) introduce la relazione di ereditarietà; essa introduce *anche* una relazione di sottotipo quando la classe derivata dichiara come `public` la classe base (cioè la superclasse); in caso contrario la classe derivata eredita dalla classe base, ma non vi è sottotipo.

```
class A{
public:
    void f() {...}
    ...
}

class B : public A{
public:
    void g() {...}
    ...
}

class C : A{
public:
    void h() {...}
    ...
}
```

Entrambe le classi `B` e `C` ereditano da `A`, ma solo `B` è un sottotipo di `A`.

Siccome la relazione di sottotipo segue da quella di sottoclasse, con gli strumenti visti sin qui non sarebbe possibile introdurre un “sottotipo di interfaccia”, cioè una classe derivata da una classe base che non fornisce alcuna implementazione, ma fissa solo un’interfaccia. È a questo scopo che C++ introduce il concetto di classe base *astratta*, cioè nella quale qualche metodo non abbia un’implementazione: in tal caso, come per le interfacce di Java, si ha sottotipo senza avere ereditarietà.

Quale dei due metodi di nome `f` è ereditato in `C`? Possiamo risolvere questo problema in tre modi distinti, nessuno dei quali è del tutto soddisfacente:

1. vietare sintatticamente che si possa produrre un conflitto di nomi;
2. imporre che ogni conflitto sia risolto esplicitamente dal programma, qualificando opportunamente ogni riferimento al nome in conflitto; ad esempio, il corpo di `h` nella classe `C` dovrebbe essere scritto come `B::f()`, o `A::f()`, che è la soluzione adottata da C++;
3. decidere convenzionalmente come risolvere il conflitto, ad esempio a favore della prima classe nominata nella clausola `extending`.

Da un punto di vista pragmatico, è possibile dare esempi di situazioni nelle quali ciascuna di queste soluzioni si presenta innaturale e controintuitiva. Quanto alla risoluzione esplicita, si osservi che il conflitto potrebbe verificarsi non in `C` (che potrebbe non chiamare esplicitamente il metodo `f`), ma in una sua sottoclasse: il

progettista deve così conoscere con una certa precisione la gerarchia delle classi. In ogni modo, in casi del genere è metodologicamente opportuno ridefinire esplicitamente `f` in `C` risolvendo il conflitto; ad esempio:

```
class C extending A,B{
    int f(){
        return A::f();
    }
    int h(){
        return this.f();
    }
}
```

in modo che nelle sottoclassi di `C` non si presenti più il conflitto di nomi.

I problemi più interessanti dell’ereditarietà multipla si presentano tuttavia nel cosiddetto problema del diamante, che si verifica quando le diverse superclassi da cui si eredita, ereditano esse stesse da una stessa superclasse. Una semplice situazione del genere è la seguente (di cui la Figura 12.4 rappresenta graficamente la gerarchia di ereditarietà):

```
class Top{
    int w;
    int f(){
        return w;
    }
}

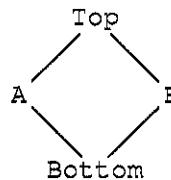
class A extending Top{
    int x;
    int g(){
        return w+x;
    }
}

class B extending Top{
    int y;
    int f(){
        return w+y;
    }
    int k(){
        return y;
    }
}

class Bottom extending A,B{
    int z;
    int h(){
        return z;
    }
}
```

Anche in questo caso abbiamo il solito problema di conflitto di nomi, ma è soprattutto il problema implementativo quello più rilevante, cioè come far sì che il metodo `f` invocato su un’istanza di `Bottom` selezioni correttamente ed efficientemente il codice opportuno. Affronteremo questa questione nel Paragrafo 12.3.4.

In conclusione, l’ereditarietà multipla è uno strumento molto flessibile per la combinazione di astrazioni corrispondenti a funzionalità distinte. Alcune delle



**Figura 12.4** Il problema del diamante nell'ereditarietà multipla.

situazioni cui risponde sono in realtà meglio espresse con la relazione di sottotipo (“ereditando” da classi astratte), ma vi sono significativi esempi dove sono davvero più implementazioni che si vogliono ereditare. Non vi sono soluzioni semplici, univoche, ed eleganti ai problemi che l'ereditarietà multipla pone e il bilancio costi-benefici tra ereditarietà singola e multipla non pende in modo netto da nessuna delle due parti.

## 12.2.6 Selezione dinamica dei metodi

La selezione dinamica dei metodi (*dynamic method lookup*, o *dispatch*) è il cuore del paradigma orientato agli oggetti, dove le altre caratteristiche che abbiamo già discusso si incontrano e danno luogo ad una sintesi nuova. In particolare, è la selezione dinamica a permettere la convivenza tra la compatibilità dei sottotipi e l'astrazione, che abbiamo visto essere problematica in un contesto di soli tipi di dato astratti (pag. 363).

Concettualmente, il meccanismo è molto semplice: abbiamo visto che un metodo definito per un oggetto può essere ridefinito (overridden) su oggetti che appartengono a sottotipi dell'oggetto originale. Quando un metodo *m* viene invocato su un oggetto *o*, pertanto, vi possono essere più versioni di *m* tutte possibili per *o*. La selezione di quale implementazione di *m* venga usata in un'invocazione

*o.m(parametri);*

avviene a tempo d'esecuzione, in funzione del *tipo* dell'oggetto che riceve il messaggio. Si faccia attenzione al fatto che si fa riferimento al tipo dell'*oggetto* che riceve il messaggio, e non al tipo del riferimento (o nome) di quell'oggetto (che è invece un'informazione statica).

Facciamo un esempio nel nostro pseudolinguaggio con classi. La Figura 12.5 riporta ancora una volta le classi dei contatori che abbiamo già definito nei Paragrafi 12.2.2 e 12.2.4. Nel contesto di quelle dichiarazioni di classi, eseguiamo ora il frammento seguente:

```

NewCounter n = new NewCounter();
Counter c;
c = n;
c.reset();
  
```

```

class Counter{
    private int x;
    public void reset(){
        x = 0;
    }
    public int get(){
        return x;
    }
    public void inc(){
        x = x+1;
    }
}
class NewCounter extending Counter{
    private int num_reset = 0;
    public void reset(){
        x = 0;
        num_reset = num_reset + 1;
    }
    public int quanti_reset(){
        return num_reset;
    }
}
  
```

**Figura 12.5** Due classi per contatori.

Il tipo (statico) del nome *c* è *Counter*, ma si riferisce (dinamicamente) ad un'istanza di *NewCounter*: sarà così il metodo *reset* di *NewCounter* ad essere invocato.

L'esempio canonico è quello col quale abbiamo concluso (negativamente) il Paragrafo 12.1: in una struttura dati del supertipo sono memorizzati sia *Counter* che *NewCounter*:

*Counter V[100];*

Ora applichiamo il metodo *reset* ad ognuno di essi:

```

for (int i=1; i<100; i=i+1)
    V[i].reset();
  
```

La selezione dinamica assicura che su ogni contatore sia invocato il metodo corretto. Si osservi come in generale il compilatore non possa decidere quale sarà il tipo dell'oggetto cui il metodo sarà inviato, da cui la dinamicità di questo meccanismo.

Il lettore potrebbe aver notato una certa analogia tra l'overloading e la selezione dinamica dei metodi. In entrambi i casi il problema è lo stesso: risolvere una situazione di ambiguità nella quale uno stesso nome può avere più significati. Nell'overloading, tuttavia, l'ambiguità è risolta staticamente, in base al tipo dei *nomi* coinvolti. Nella selezione dei metodi, invece, la risoluzione dell'ambiguità avviene a tempo d'esecuzione, sfruttando il tipo “dinamico” dell'oggetto e non del suo nome. Non è però errato pensare alla selezione dei metodi come ad un

### Selezione dinamica in C++

In linguaggi come Java o Smalltalk ogni invocazione di metodo avviene con selezione dinamica. C++ pone alla base del suo progetto l'efficienza d'esecuzione e la compatibilità con C, intendendo con questo anche che l'uso di una caratteristica di C da parte di un programma C++ deve essere efficiente quanto l'uso della stessa caratteristica in C.

In C++ abbiamo pertanto un'invocazione statica di metodi (analogia alla chiamata di una funzione) e un'invocazione dinamica, che si attua sulle *funzioni virtuali*. Al momento della definizione di un metodo (cioè di una member function, in terminologia C++), è possibile specificare se si tratta di una funzione virtuale o meno. È permesso l'overriding delle sole funzioni virtuali, sulle quali poi agisce la selezione dinamica.

Osserviamo, per inciso, che non è vietato definire in una sottoclasse una funzione con lo stesso nome e segnatura di una funzione non virtuale definita nella superclasse. In tal caso non si ha ridefinizione, ma overloading, che verrà risolto staticamente dal compilatore in base al tipo del *nome* col quale ci si riferisce all'oggetto. Nell'esempio che segue dichiariamo una classe A e una sottoclasse B:

```
class A{
public:
    void f(){printf("A");}
    virtual void g(){printf("A");}
}
class B : public A{
public:
    void f(){printf("B");}
    virtual void g(){printf("B");}
}
```

Se ora abbiamo un puntatore a di tipo A\* che punta ad un'istanza di B, l'invocazione della funzione a->f() stampa A, mentre l'invocazione della funzione virtuale a->g() stampa B.

“overloading a tempo d'esecuzione”, nel quale l'oggetto che riceve il messaggio è visto come il primo argomento (implicito) del metodo di cui si deve risolvere il nome<sup>14</sup>.

È importante osservare esplicitamente che la selezione dinamica dei metodi è all'opera anche quando un metodo di un oggetto invoca un metodo dello stesso oggetto, come avviene nel seguente frammento:

<sup>14</sup> Il lettore non si stupirà se, ancora una volta, il gergo della programmazione orientata agli oggetti contribuisce alla confusione: non è raro sentir dire (e trovar scritto) che la selezione dinamica dei metodi permette il polimorfismo. Dovrebbe essere inutile sottolineare che non c'è alcun polimorfismo, perché non siamo in presenza di un singolo codice “uniforme”. È possibile parlare di polimorfismo nella programmazione orientata agli oggetti, ma questo ha a che vedere con i sottotipi, e l'affronteremo nel Paragrafo 12.4.1.

```
class A{
    int a = 0;
    void f(){g();}
    void g(){a=1;}
}
class B extending A{
    int b = 0;
    void g(){b=2;}
}
```

Supponiamo ora di avere un oggetto b istanza di B e di invocare su b il metodo f che b eredita da A. Ora f invoca g: quale delle due implementazioni di g viene eseguita? Ricordiamo che l'oggetto che riceve il messaggio è sempre un parametro隐式的 del metodo: la chiamata a g nel corpo di f potrebbe essere scritta più esplicitamente this.g(), dove, lo ricordiamo, this è un riferimento all'oggetto corrente. L'oggetto corrente è b e dunque il metodo che viene invocato è quello della classe B. Si osservi che in questo modo una chiamata di metodo come this.g() nel corpo di f può riferirsi a (implementazioni di) metodi non ancora scritti, che saranno disponibili solo in un secondo momento, per effetto della gerarchia delle classi. Questo meccanismo, col quale il nome this viene dinamicamente legato all'oggetto corrente, viene indicato come il *late binding* (legame tardivo) di self (o di this).

Osserviamo esplicitamente che, a differenza dell'overriding, lo shadowing è un meccanismo del tutto statico. Consideriamo, ad esempio, il codice:

```
class A{
    int a = 1;
    int f(){return -a;}
}
class B extending A{
    int a = 2;
    int f(){return a;}
}

B ogg_b = new B();
stampa(ogg_b.f());
stampa(ogg_b.a);

A ogg_a = ogg_b;
stampa(ogg_a.f());
stampa(ogg_a.a);
```

dove con stampa indichiamo un metodo che permette di stampare il valore intero passato come argomento. Le prime due invocazioni di stampa producono due volte il valore 2, come ovvio. La terza stamperà anch'essa il valore 2 dato che, come già visto prima, il metodo f è stato ridefinito nella classe B. Infatti l'oggetto creato (con la new B()) è un'istanza di B, per cui ogni riferimento a tale oggetto, anche attraverso variabili di tipo A (come nel caso di ogg\_a), usa il metodo ridefinito (tale metodo, ovviamente, userà i campi ridefiniti nella classe B). L'ultima occorrenza di stampa produce invece il valore 1. In questo caso, infatti, dato che non si tratta di invocare un metodo ma di accedere ad un campo, è il tipo del riferimento corrente a determinare quale campo si deve considerare. Dato che ogg\_a è

## Linguaggi multimedodo

Nei linguaggi che abbiamo considerato sin qui un'invocazione di metodo della forma

```
o.m(parametri);
```

viene risolta dinamicamente per quanto riguarda l'oggetto che riceve il messaggio, ma staticamente per quanto riguarda il tipo dei parametri. In alcuni linguaggi (per esempio CLOS) questa asimmetria viene rimossa: i metodi non sono più associati alle classi, ma sono nomi globali. Ogni nome di metodo è overloaded su più codice (si dicono in tal caso *multimedodi*) e ad ogni multimedodo viene passato come argomento l'oggetto sul quale il metodo viene invocato. Al momento dell'invocazione di un multimedodo, viene dinamicamente selezionato quale codice debba essere eseguito, sulla base del tipo (dinamico) sia del ricevitore sia di tutti gli argomenti.

In questi linguaggi si parla di *dispatch multiplo*, in opposizione al dispatch singolo dei linguaggi nei quali esiste un ricevitore privilegiato.

Nei linguaggi con dispatch multiplo l'analogia tra selezione dinamica (multiplo) e "overloading dinamico" è ancora più netta: il dispatch multiplo consiste nella risoluzione (a tempo d'esecuzione) di un overloading sulla base di informazioni di tipo dinamiche.

di tipo A, quando scriviamo `ogg_a.a` il campo `a` in questione è quello della classe A (inizializzato ad 1).

Questa distinzione tra overriding e shadowing, che per certi versi può apparire poco chiara, si spiega a livello implementativo con il fatto che l'oggetto istanza della classe B contiene anche tutti i campi delle superclassi di B, come chiariremo nel prossimo paragrafo.

## 12.3 Aspetti implementativi

Ogni linguaggio adotta un proprio modello implementativo, ottimizzato per le caratteristiche specifiche che quel linguaggio fornisce. Possiamo tuttavia indicare alcune linee comuni, che indicano i principali problemi e le relative soluzioni.

**Oggetti** Un oggetto istanza della classe A può essere usato come valore di ogni superclasse di A; in particolare è possibile accedere non solo alle variabili di istanza (i campi per i dati) esplicitamente definite in A, ma anche a quelle definite nelle superclassi. Un oggetto può essere così rappresentato come se fosse un record, con tanti campi quante sono le variabili della classe di cui è istanza più tutte quelle che appaiono nelle sue superclassi. Nel caso di shadowing, ossia quando uno stesso nome di una variabile di istanza in una classe è usato con lo stesso tipo in una sottoclasse, l'oggetto ha più campi, ciascuno corrispondente ad una diversa dichiarazione (spesso il nome usato nella superclasse non è accessibile nella

```
class A{
    int a;
    void f(){...}
    void g(){...}
}
class B extending A{
    int b;
    void f(){...} // ridefinito
    void h(){...}
}
B o = new B();
```

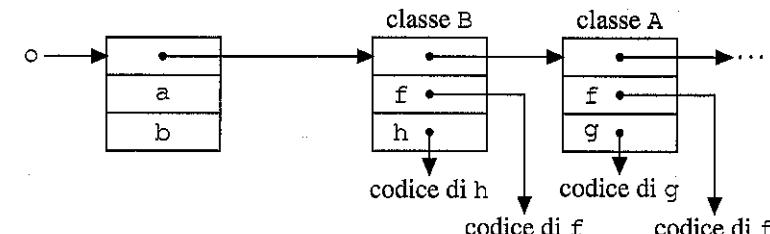


Figura 12.6 Una semplice implementazione dell'ereditarietà.

sottoclasse, se non usando qualificatori particolari, per esempio `super`). La rappresentazione dell'oggetto contiene anche un puntatore al descrittore della classe di cui è istanza.

Nel caso di un linguaggio con sistema di tipi statico, questa rappresentazione permette una semplice implementazione della compatibilità di sottotipo (in caso di ereditarietà semplice): di ogni variabile di istanza è noto staticamente l'offset rispetto all'inizio del record. Se ad un oggetto istanza di una certa classe si accede mediante un riferimento (statico) di una superclasse, il controllo statico dei tipi assicura che si può accedere solo ad un campo della superclasse, che sarà allocato nella parte iniziale del record.

**Classi ed ereditarietà** La rappresentazione delle classi è la chiave di volta di una macchina astratta orientata agli oggetti. L'implementazione più semplice e intuitiva è quella che rappresenta la gerarchia delle classi mediante una lista concatenata. Ogni elemento rappresenta una classe e contiene (puntatori al-) l'implementazione di tutti i metodi esplicitamente definiti o ridefiniti in quella classe. Gli elementi sono collegati tra loro mediante un puntatore che va dalla sottoclasse alla superclasse immediata. Quando viene invocato un metodo `m` su un oggetto `o` istanza di una classe `C`, si sfrutta il puntatore presente in `o` per accedere al descrittore di `C` e si verifica se `C` contiene un'implementazione per `m`; in caso negativo si risale il puntatore alla superclasse e il procedimento continua (si veda la Figura 12.6). Adottata in Smalltalk, questa implementazione è concettualmente

semplice, ma assai inefficiente, perché ogni invocazione di metodo richiede una scansione lineare della gerarchia delle classi. Discuteremo implementazioni più efficienti tra un attimo, dopo aver visto come accedere alle variabili di istanza.

**Late binding di self** Un metodo viene eseguito sostanzialmente come una funzione: sulla pila viene messo un RdA per le variabili locali del metodo, i parametri e tutte le altre informazioni. A differenza di una funzione, però, un metodo deve accedere anche alle variabili di istanza dell'oggetto sul quale è invocato, che non è noto al momento della compilazione. È però nota la struttura di tale oggetto (che dipende dalla sua classe) e, dunque, per ogni variabile di istanza è staticamente noto (sotto ampie ipotesi che dipendono dal linguaggio) il suo offset all'interno della rappresentazione dell'oggetto. Quando un metodo viene invocato, gli viene passato anche un puntatore all'oggetto che ha ricevuto il metodo: durante l'esecuzione del corpo del metodo tale puntatore è il *this* del metodo. Qualora il metodo sia invocato sullo stesso oggetto che lo sta invocando, viene passato *this*. Durante l'esecuzione del metodo, ogni accesso ad una variabile di istanza avviene per offset rispetto a questo puntatore (invece che per offset rispetto al puntatore all'RdA come avviene per le variabili locali). Da un punto di vista logico, possiamo supporre che questo puntatore sia passato attraverso l'RdA come tutti gli altri parametri, ma ciò causerebbe un doppio accesso indiretto per ogni riferimento ad una variabile di istanza (uno per accedere al puntatore attraverso il puntatore all'RdA e uno per accedere alla variabile attraverso il puntatore). Più efficientemente, la macchina astratta manterrà in un registro il valore corrente di *this*.

### 12.3.1 Ereditarietà singola

Nell'ipotesi che il linguaggio abbia un sistema di tipi statico, l'implementazione a liste concatenate può essere sostituita da un'altra assai più efficiente, nella quale la selezione dei metodi avviene in tempo costante (invece che lineare nella profondità della gerarchia delle classi).

Se i tipi sono statici, per ogni oggetto è noto a tempo di compilazione l'insieme di tutti i metodi che possono essere inviati a quell'oggetto. L'elenco di questi metodi è mantenuto nel descrittore della classe: non solo i metodi esplicitamente definiti o ridefiniti nella classe, ma anche tutti quelli ereditati dalle superclassi<sup>15</sup>. Seguendo la terminologia di C++, chiamiamo *vtable* (per *virtual function table*) una tale struttura dati. Ad ogni definizione di classe corrisponde una vtable e tutte le istanze di quella classe condividono la stessa vtable. Quando viene definita una sottoclasse B della classe A, la vtable di B si ottiene facendo una copia della vtable

<sup>15</sup>Stiamo assumendo che la selezione dinamica si applichi a tutti i metodi; in situazioni come quella di C++, dove solo i metodi virtuali possono essere ridefiniti, il descrittore contiene solo questi ultimi, mentre la chiamata di un metodo ordinario avviene come una normale chiamata di funzione.

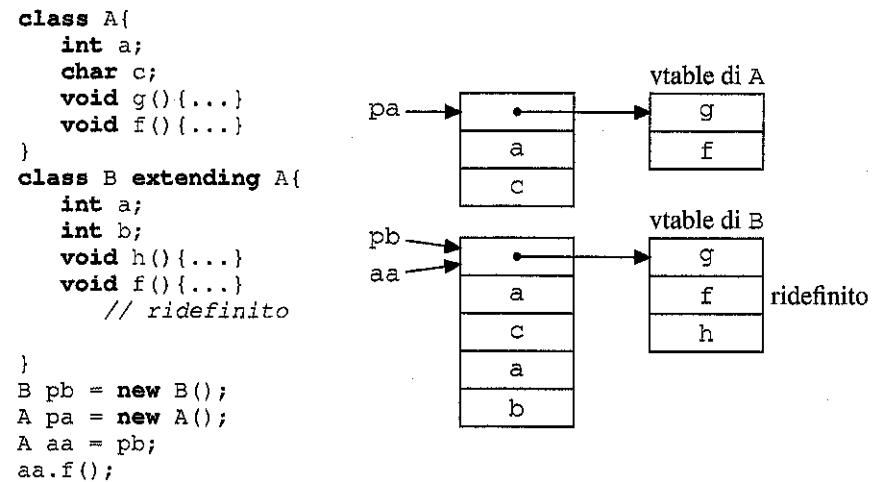


Figura 12.7 Implementazione dell'ereditarietà singola.

di A, sostituendo in questa copia tutti i metodi ridefiniti in B e aggiungendo poi in fondo alla vtable i nuovi metodi definiti in B, come descritto nella Figura 12.7.

La proprietà fondamentale di queste strutture dati è che, se B è sottoclasse di A, la vtable di B contiene come parte iniziale una copia della vtable di A, dove i metodi ridefiniti sono stati opportunamente modificati. In questo modo l'invocazione di un metodo avviene al prezzo di due accessi indiretti, visto che è staticamente noto l'offset di ogni metodo all'interno della vtable. Si osservi come questa implementazione tiene ovviamente conto che si può accedere ad un oggetto con un riferimento che appartiene staticamente ad una sua superclasse. Nell'esempio della Figura 12.7, quando si invoca il metodo f, il compilatore calcola per tale metodo un offset che rimane lo stesso sia quando f è invocato su un oggetto di classe A che quando è invocato su un oggetto di classe B. Allo stesso offset corrispondono, nelle vtable di A e di B, le diverse implementazioni che le due classi assegnano a f.

Nel complesso, se abbiamo un riferimento statico pa di classe A a qualche oggetto di una sua sottoclasse, possiamo compilare una chiamata al metodo f (che supponiamo essere l'*n*-esimo nella vtable di A) come segue (abbiamo supposto che l'indirizzo di un metodo occupi *w* byte):

```

R1 := pa                                // accesso all'oggetto
R2 := * (R1)                             // vtable
R3 := * (R2 + (n - 1) × w)               // indirizzo di f
call * (R3)                             // chiamata di f

```

Se si trascura il supporto dell'ereditarietà multipla, questo schema implementativo è nella sostanza quello di C++.

**Cast verso il basso** Se la vtable di una classe contiene anche il nome della classe stessa, l'implementazione che abbiamo discusso permette di gestire anche i cast verso il basso (o “cast all'ingiù”), un meccanismo abbastanza comune che permette di specializzare il tipo di un oggetto, andando in senso contrario rispetto alla gerarchia dei sottotipi. Un esempio canonico dell'uso di questo meccanismo si ha con alcuni metodi di libreria, definiti per essere i più generali possibile. La classe `Object` potrebbe definire un metodo per la duplicazione di un oggetto con la seguente segnatura:

```
Object clone() {...}
```

con la semantica che `clone` restituisce una copia esatta dell'oggetto cui è inviato (stessa classe, stesso valore dei campi ecc.). Se abbiamo un oggetto `o` di classe `A`, un'invocazione di `o.clone()` restituirà un'istanza di `A`, ma il tipo statico determinato dal compilatore per questa espressione è `Object`. Per poter usare in modo significativo questa nuova copia, la dovremo “forzare” all'ingiù nel tipo `A`, che possiamo indicare con un cast

```
A a = (A) o.clone();
```

nel quale intendiamo che la macchina astratta effettivamente verifichi che il tipo dinamico dell'oggetto è davvero `A` (in caso contrario si avrà un errore di tipo a tempo d'esecuzione)<sup>16</sup>.

### 12.3.2 Il problema della classe base fragile

L'implementazione che abbiamo appena descritto per l'ereditarietà semplice è assai efficiente, visto che tutte le informazioni importanti, tranne il puntatore a `this`, sono determinate staticamente. Questo stesso fatto, tuttavia, si può rivelare fonte di problemi, in un contesto noto come problema della classe base (cioè della superclasse) fragile.

Un sistema orientato agli oggetti è strutturato in un numero molto elevato di classi, con un'elaborata gerarchia di ereditarietà. Spesso alcune classi generali sono fornite in librerie, che uno specifico sistema eredita. Modifiche ad una classe in posizione molto alta nella gerarchia si possono ripercuotere nelle sottoclassi di quella classe. Alcune superclassi si possono così dimostrare “fragili”, perché modifiche apparentemente innocue alla superclasse possono causare malfunzionamenti nelle sottoclassi. Non è possibile individuare la fragilità analizzando solo la superclasse, ma occorre considerare l'intera gerarchia di ereditarietà, cosa spesso impossibile perché chi scrive la superclasse non ha in genere accesso a tutte le sottoclassi. Il problema può insorgere per più motivi; a noi interessa distinguere due casi principali.

<sup>16</sup>Si tratta della notazione di Java. C++ permette la stessa notazione, ma, per compatibilità con C, si limita ad accettare l'espressione a tempo di compilazione, senza introdurre controlli dinamici. Un cast con controllo dinamico si sarebbe scritto in C++ `dynamic_cast<A*>(o)`.

- Il problema è architettonale: qualche sottoclasse sfrutta aspetti dell'implementazione della superclasse che sono stati modificati nella nuova versione. Si tratta di un problema importantissimo dell'ingegneria del software, che può essere limitato riducendo l'ereditarietà (di implementazione) a favore della relazione di sottotipo (cioè l'ereditarietà di interfaccia).
- Il problema è implementativo: il malfunzionamento della sottoclasse dipende solo da come la macchina astratta (e il compilatore) ha rappresentato l'ereditarietà. Questo caso viene talvolta indicato come il problema dell'*interfaccia binaria fragile*.

In questo testo non è di nostro interesse primario il primo caso, bensì il secondo. Un caso tipico si verifica in un contesto di compilazione separata, quando ad una superclasse si aggiunge un metodo che non interagisce con nessun altro. Partendo da una situazione del tipo:

```
class Sopra{
    int x;
    int f(){...}
}
class Sotto extending Sopra{
    int y;
    int g(){return y + f();}
}
```

la superclasse viene modificata in

```
class Sopra{
    int x;
    int h(){return x;}
    int f(){...}
}
```

Se l'ereditarietà è implementata come descritto nel Paragrafo 12.3.1, la sottoclasse `Sotto` (già compilata e che venga ricollegata a `Sopra`) cessa di funzionare correttamente, perché nell'accesso al metodo `f` è cambiato l'offset, che era stato determinato staticamente. Per risolvere il problema occorre ricompilare tutte le sottoclassi della classe modificata, una soluzione per nulla ovvia in molte situazioni.

Per ovviare a questo problema occorre calcolare dinamicamente l'offset dei metodi nella vtable (ma anche delle variabili d'istanza nella rappresentazione degli oggetti), in modo ragionevolmente efficiente. Il prossimo paragrafo descrive una possibile soluzione.

### 12.3.3 Selezione dinamica dei metodi nella JVM

Presentiamo in questo paragrafo una semplificazione della tecnica usata dalla Java Virtual Machine (JVM), la macchina astratta che interpreta il linguaggio intermedio (bytecode) generato dal compilatore standard di Java. Per motivi di spazio non possiamo entrare nei dettagli dell'architettura della JVM, una macchina basata su pila (non ha registri accessibili all'utente e tutti gli operandi delle operazioni

sono passati su una pila che è contenuta nel RdA della funzione correntemente in esecuzione) con significativi moduli che controllano la sicurezza delle operazioni. Ci limiteremo a discutere a grandi linee l'implementazione dell'ereditarietà e del dispatching dei metodi.

In Java la compilazione delle classi avviene in modo separato: ogni classe dà luogo ad un file che la macchina astratta carica dinamicamente quando il programma in esecuzione effettua un riferimento a quella classe. Tale file contiene una tabella dei simboli (la *constant pool*) usati nella classe stessa: variabili di istanza, metodi pubblici e privati, metodi e campi di altre classi usati nel corpo dei metodi, nomi di altre classi usate nel corpo della classe ecc. Ad ogni nome di variabile d'istanza e di metodo sono associate alcune informazioni, tra le quali la classe dove i nomi sono definiti e il loro tipo. Tutte le volte che il codice sorgente usa un nome, nel codice intermedio della JVM per salvare spazio si trova l'indice che quel nome ha nella constant pool (e non il nome stesso). Nel momento in cui durante l'esecuzione si fa riferimento ad un nome per la prima volta (attraverso il suo indice), questo viene *risolto*: usando le informazioni della constant pool vengono caricate le classi necessarie (per esempio quelle dove il nome è introdotto), vengono controllati i vincoli di visibilità (ad esempio che il metodo invocato davvero esiste nella classe cui ci si riferisce, che non sia privato ecc.) e quelli di tipo. A questo punto la macchina astratta salva un puntatore a queste informazioni, cosicché la prossima volta che si usa lo stesso nome non sia necessaria una nuova risoluzione.

La rappresentazione dei metodi in un descrittore di classe può essere pensata analoga a quella di una vtable<sup>17</sup>: la tabella relativa ad una sottoclasse inizia con una copia di quella della superclasse, dove i metodi ridefiniti sono stati sostituiti con la nuova definizione. Gli offset, tuttavia, non sono calcolati staticamente. Quando deve essere invocato un metodo, possiamo distinguere quattro casi principali (che corrispondono a quattro distinte istruzioni nel bytecode):

1. il metodo è statico, cioè un metodo associato ad una classe e non ad un'istanza; come tale non può fare riferimento (esplicito o implicito) a *this*;
2. il metodo deve essere selezionato dinamicamente (metodi "virtuali");
3. il metodo deve essere selezionato dinamicamente ed è invocato tramite *this* (metodi "speciali");
4. il metodo proviene da un'interfaccia (cioè da una classe completamente astratta che non fornisce implementazioni: metodi "di interfaccia");

Rimandiamo per il momento l'ultimo caso; gli altri tre si differenziano tra loro soltanto (nella sostanza) per i parametri che vengono passati al metodo: nel primo caso non viene passato alcun oggetto oltre ai parametri nominati nella chiamata; nel secondo viene passato un riferimento all'oggetto su cui è chiamato il

<sup>17</sup> A dire il vero la specifica della JVM non prescrive alcuna forma di rappresentazione, limitandosi a richiedere che la ricerca del metodo da eseguire sia semanticamente equivalente alla ricerca dinamica del nome del metodo nella lista concatenata delle sottoclassi, come abbiamo descritto poc'anzi. La realizzazione più comune, tuttavia, è quella di una tabella molto simile ad una vtable.

metodo; nel terzo viene passato un riferimento a *this*. Supponiamo dunque che si debba invocare il metodo *m* sull'oggetto *o*:

*o.m(parametri)*

Attraverso il riferimento ad *o*, la macchina astratta accede alla constant pool della sua classe e da questa preleva il *nome* di *m*. A questo punto, ricerca questo nome nella vtable della classe e determina il suo offset. Questo offset viene salvato per gli usi futuri dello stesso metodo sullo stesso oggetto.

Tuttavia lo stesso metodo potrebbe essere invocato anche su altri oggetti, probabilmente di sottoclassi di quella cui appartiene *o* (si pensi al solito ciclo *for* nel corpo del quale si invoca *m* su tutti gli oggetti presenti in un array). Per evitare di calcolare ogni volta l'offset (che sarà sempre lo stesso indipendentemente dalla classe effettiva di cui *o* è istanza), l'interprete della JVM usa una tecnica di "riscrittura del codice": sostituisce all'istruzione di *lookup standard* generata dal compilatore una sua forma ottimizzata che prende come argomento l'offset del metodo nella vtable. Per fissare le idee (e semplificando molto), per tradurre l'invocazione di un metodo virtuale *m* il compilatore potrebbe aver generato un'istruzione in bytecode

*invokevirtual index*

dove *index* è l'indice del nome *m* nella constant pool. Nel corso dell'esecuzione di questa istruzione l'interprete della JVM calcola l'offset *d* di *m* nella sua vtable e sostituisce l'istruzione precedente con

*invokevirtual\_quick d np*

dove *np* è il numero dei parametri che *m* si aspetta (e che troverà sullo stack che fa parte del suo RdA). Tutte le volte che il flusso del controllo ritornerà su questa istruzione, si invocherà *m* (usando il riferimento all'oggetto sulla pila degli operandi) senza overhead per la ricerca.

Rimane da trattare il caso dell'invocazione di un metodo di interfaccia. In questo caso l'offset potrebbe non essere lo stesso nel caso di due invocazioni dello stesso metodo su oggetti di classi diverse. Si consideri infatti la situazione seguente (nella quale usiamo la sintassi di Java invece di pseudocodice):

```
interface Interfaccia{
    void foo();
}

public class A implements Interfaccia{
    int x;
    void foo(){...}
}

public class B implements Interfaccia{
    int y;
    int fie(){...}
    int foo(){...}
}
```

Sia *A* che *B* implementano *Interfaccia* e dunque sono suoi sottotipi, ma l'offset di *foo* è diverso nelle due classi. Nel solito ciclo

```
Interfaccia V[10];
...
for (int i = 0; i<10; i=i+1)
    V[i].foo();
```

non sappiamo se, al momento dell'esecuzione, gli oggetti contenuti in *V* saranno istanza di *A*, o di *B* o di qualche altra classe che implementi *Interfaccia*. Il compilatore, in corrispondenza del corpo del ciclo potrebbe aver generato un'istruzione per la JVM

```
invokeinterface index, 0
```

(lo zero serve a riempire un byte che sarà usato nella versione "quick"). Non sarebbe corretto sostituire direttamente questa istruzione con una versione "quick" che riporti solo l'offset, perché cambiando l'oggetto potrebbe anche cambiare la classe di cui è istanza. Quello che possiamo fare è salvare comunque l'offset, ma non distruggere il nome originale del metodo e riscrivere l'istruzione come

```
invokeinterface_quick nome_di_foo, d
```

dove *nome\_di\_foo* sono opportune informazioni con le quali ricostruire il nome e la segnatura di *foo* e *d* è l'offset determinato in precedenza. Quando questa istruzione sarà di nuovo eseguita, l'interprete accede alla vtable con offset *d* e controlla che vi sia un metodo col nome e la segnatura richiesti. In caso positivo, lo invoca; in caso negativo, ricerca per nome il metodo nella vtable, come per la prima volta, determina un nuovo offset *d'* e riscrive tale valore al posto di *d* nel codice.

### 12.3.4 Ereditarietà multipla

L'implementazione dell'ereditarietà multipla pone problemi interessanti e richiede un overhead non trascurabile. I problemi sono di due ordini: da una parte c'è da chiarire come sia possibile adattare la tecnica delle vtable per gestire le chiamate dei metodi; dall'altra (ed il problema è più interessante ed ha anche un risvolto linguistico) occorre stabilire cosa fare dei dati presenti nelle superclassi e questo darà luogo a due interpretazioni diverse dell'ereditarietà multipla, che possiamo chiamare ereditarietà con replicazione ed ereditarietà con condivisione. Trattiamo di questi problemi in successione.

**Struttura delle vtable** Imposteremo la nostra discussione discutendo l'esempio della Figura 12.8. È evidente che non è possibile organizzare né la rappresentazione di un'istanza di *C*, né una vtable per *C* in modo che le loro parti iniziali coincidano con le corrispondenti strutture per *A* e per *B*.

Per rappresentare un'istanza di *C* possiamo iniziare con i campi di *A*, farli seguire dai campi di *B* e infine elencare i campi specifici di *C* (Figura 12.9). Sappiamo che, per effetto della relazione di sottotipo, possiamo accedere ad un'istanza di *C* con riferimenti statici dei tre tipi *A*, *B* e *C*. Vi sono due situazioni distinte, che corrispondono a due "viste" diverse di un'istanza di *C*: qualora vi si acceda

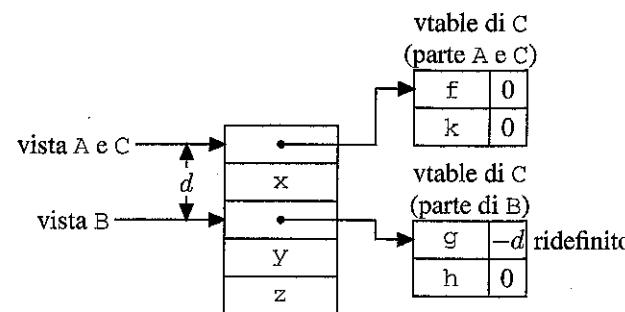
```
class A{
    int x;
    int f(){
        return x;
    }
}
class B{
    int y;
    int g(){
        return y;
    }
    int h(){
        return 2*y;
    }
}
class C extending A,B{
    int z;
    int g(){
        return x + y + z;
    }
    int k(){
        return z;
    }
}
```

Figura 12.8 Un caso di ereditarietà multipla.

con un riferimento statico di tipo *C* o di tipo *A*, la tecnica descritta per l'ereditarietà semplice funziona perfettamente (salvo che l'offset statico delle variabili di istanza proprie di *C* deve tener conto del fatto che ci sono "nel mezzo" le variabili statiche proprie di *B*).

Quando, invece, l'istanza di *C* sia vista come un oggetto di *B*, occorre tener conto che le variabili di *B* *non* sono all'inizio del record, ma ad una distanza *d* staticamente determinata dall'inizio di esso. Quando dunque l'accesso ad un'istanza di *C* avviene mediante un riferimento ma con tipo statico *B*, occorre sommare al riferimento questa costante *d*.

Problemi analoghi sorgono per la struttura della vtable. Una vtable per *C* è divisa in due parti distinte: la prima comprende i metodi di *A* (eventualmente con la loro ridefinizione) e i metodi propri di *C*; una seconda comprende i metodi di *B*, eventualmente con le loro ridefinizioni. Nella rappresentazione dell'oggetto istanza di *C* vi sono due puntatori alle vtable: in corrispondenza della "vista *A* e *C*" avremo il puntatore alla vtable con i metodi di *A* e *C*; in corrispondenza della "vista *B*" avremo il puntatore alla vtable con i metodi di *B* (si osservi che questa è una vtable della classe *C*; la classe *B* ha un'altra vtable, che è usata dalle istanze di quella classe: in generale ogni superclasse di una classe in eredità multipla ha una sua parte specifica nella vtable della sottoclasse). L'invocazione di un



**Figura 12.9** Rappresentazione di oggetti e vtable per l'ereditarietà multipla.

metodo proprio di C o che viene ridefinito o ereditato da A segue le stesse regole dell'ereditarietà singola. Per invocare un metodo ereditato (o ridefinito) da B, il compilatore deve tener conto che il puntatore alla vtable di questi metodi non risiede all'inizio dell'oggetto, ma è spostato in avanti di  $d$  posizioni. Nell'esempio della figura, per chiamare il metodo h di un'istanza di C vista come un oggetto di B (di cui abbiamo dunque un nome statico pb) si eseguirà: somma  $d$  al riferimento pb; mediante un accesso indiretto, ottieni l'indirizzo di inizio della seconda vtable (quella per B); mediante l'opportuno offset statico invoca finalmente il metodo h. Il costo di questa chiamata è di un'operazione in più (la prima somma) rispetto al caso dell'ereditarietà singola.

Abbiamo però trascurato il binding di this: quale riferimento all'oggetto corrente dobbiamo passare ai metodi di C? Se si tratta dei metodi della prima vtable (alla quale si accede con this che punta all'inizio dell'oggetto, cioè con la "vista A e C"), basta passare il valore corrente di this. Ma così non è per i metodi della seconda vtable (alla quale si accede col puntatore a distanza  $d$  da this). Dobbiamo distinguere due casi:

- il metodo è ereditato da B (è il caso di h in figura): in tal caso basta passare al metodo la vista dell'oggetto attraverso la quale abbiamo trovato la vtable;
- il metodo è ridefinito in C (è il caso di g): in tal caso il metodo potrebbe far riferimento alle variabili d'istanza di A, e dunque occorre passargli la vista della superclasse.

La situazione è delicata, perché la selezione dinamica dei metodi impone che questa "correzione" del valore di this sia fatta a tempo d'esecuzione. La soluzione più semplice è quella di memorizzare questa correzione nella vtable, assieme al nome del metodo. Al momento dell'invocazione del metodo, la correzione sarà sommata al valore corrente di this. Nel nostro esempio, le correzioni sono indicate nella Figura 12.9 accanto ai nomi dei relativi metodi: la correzione va sommata alla vista dell'oggetto attraverso la quale abbiamo trovato la vtable.

Nel complesso, se abbiamo un riferimento pa ad una vista di classe C di qualche oggetto, possiamo compilare una chiamata al metodo h (che supponiamo essere l' $n$ -esimo nella vtable di B) come segue (l'indirizzo di un metodo e la correzione occupano ciascuno  $w$  byte):

```
R1 := pa                                // vista A
R1 := R1 + d                            // vista B
R2 := * (R1)                            // vtable di B
R3 := * (R2 + (n - 1) × 2 × w)          // indirizzo di h
R2 := * (R2 + (n - 1) × 2 × w + w)      // correzione
this := R1 + R2
call * (R3)                            // chiamata di h
```

Si tratta di tre istruzioni e un accesso indiretto in più rispetto alla sequenza di chiamata di un metodo in ereditarietà singola.

**Ereditarietà multipla con replicazione** Il paragrafo precedente ha trattato il caso in cui una classe erediti da due superclassi. Tali superclassi, tuttavia, potrebbero esse stesse ereditare da una comune superclasse, dando luogo ad un diamante, come quello che abbiamo già discusso nel Paragrafo 12.2.5:

```
class Top{
    int w;
    int f(){
        return w;
    }
}
class A extending Top{
    int x;
    int g(){
        return w+x;
    }
}
class B extending Top{
    int y;
    int f(){
        return w+y;
    }
    int k(){
        return y;
    }
}
class Bottom extending A,B{
    int z;
    int h(){
        return z;
    }
}
```

Sia le istanze che la vtable di A hanno una parte iniziale che è una copia delle corrispondenti strutture di Top; lo stesso accade per le istanze e la vtable di B. Nell'ereditarietà multipla *con replicazione*, Bottom è costruita secondo lo schema che abbiamo appena discusso, e dunque comprende due copie delle variabili di istanza e dei metodi di Top, come indicato nella Figura 12.10.

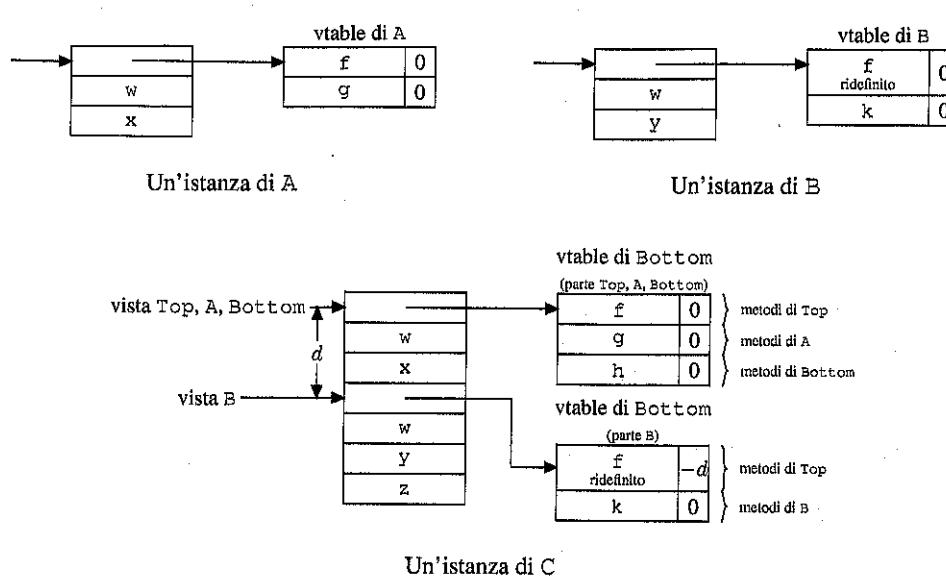


Figura 12.10 Implementazione dell'ereditarietà multipla con replicazione.

L'implementazione non pone ulteriori problemi rispetto a quelli che abbiamo discusso; i conflitti di nome devono essere risolti esplicitamente (in particolare non sarà possibile invocare il metodo *f* di *Top* su un oggetto di classe *Bottom*, né assegnare un'istanza di *Bottom* ad un riferimento statico di tipo *Top*, perché non si saprebbe quale delle due copie di *Top* selezionare).

**Ereditarietà multipla con condivisione** L'ereditarietà multipla con replicazione non è sempre la soluzione concettuale che un progettista di software ha in mente quando immagina una situazione a diamante. Qualora la classe in fondo al diamante contenga *una sola* copia della classe in cima del diamante, si parla di ereditarietà multipla *con condivisione*. In tal caso sia A che B possiedono la loro copia di *Top*, ma *Bottom* ne possiede una sola.

C++ permette l'ereditarietà multipla con condivisione attraverso le classi base *virtuali*: quando una classe è definita come *virtual*, allora tutte le sue sottoclassi comprendono sempre una sola copia di essa, anche se vi sono più cammini di ereditarietà, come nel caso del diamante. Con questo meccanismo una classe viene dichiarata virtuale o non virtuale una volta per tutte: se abbiamo una classe non virtuale e, in seguito, scopriamo che vogliamo ereditarla con condivisione, non c'è altro modo che riscrivere la classe e ricompilare tutto il sistema. Peggio, una classe è virtuale per tutte le sue sottoclassi, anche se in qualche condizione vorremmo che fosse virtuale per alcune e non virtuale (cioè replicata) per altre: in tali casi occorre definire due copie della classe, una virtuale e l'altra non virtuale.

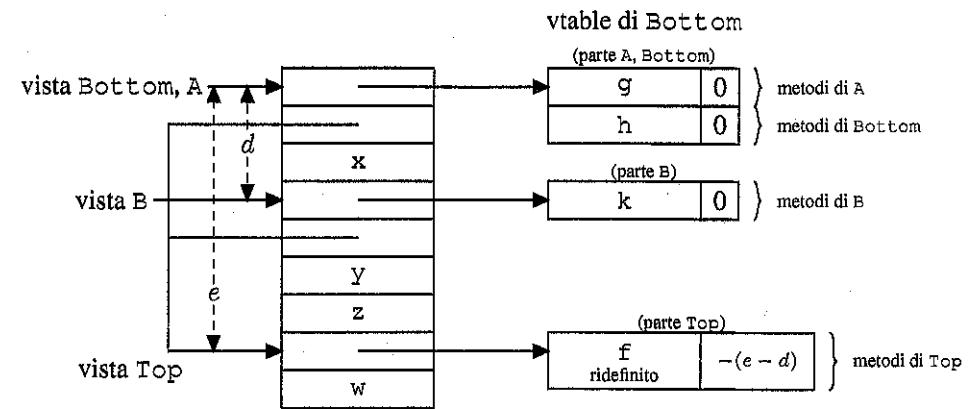


Figura 12.11 Implementazione dell'ereditarietà multipla con condivisione.

Con l'ereditarietà con condivisione c'è, al solito, un problema di conflitto dei nomi, che viene risolto in modo sostanzialmente arbitrario dai diversi linguaggi. In C++, ad esempio, si richiede che nel caso un metodo di una classe base virtuale sia ridefinito in una sottoclasse, vi sia sempre una ridefinizione che domina sulle altre, nel senso che compare in una classe che è sottoclasse di tutte le altre classi dove lo stesso metodo è definito. Nel nostro esempio, per quanto riguarda il metodo *f*, la ridefinizione dominante è quella di *B*, che è dunque quella ereditata in *Bottom*. Se entrambe le classi *A* e *B* ridefinissero *f*, le definizioni di classe sarebbero illegali in C++, perché non vi sarebbe una ridefinizione dominante. Altri linguaggi permettono alla classe che eredita di controllare lungo quale cammino ricercare il metodo ereditato, oppure ammettono una qualificazione completa dei nomi mediante la quale selezionare esplicitamente il metodo voluto. Come al solito quando si ha a che fare con l'ereditarietà multipla, non vi è una soluzione elegante che chiaramente si impone sulle altre.

Veniamo ora all'implementazione delle vtable per l'ereditarietà multipla con condivisione: la Figura 12.11 rappresenta schematicamente l'implementazione di C++, cioè di un linguaggio nel quale una classe virtuale è condivisa da tutte le sue sottoclassi. Siccome *Bottom* contiene una sola copia di *Top*, non è più possibile rappresentare in modo contiguo sottoclassi e superclassi. Ad ogni classe del diamante corrisponde una specifica vista sulla rappresentazione dell'oggetto; in corrispondenza di ogni vista c'è sia un puntatore alla vtable corrispondente, sia un puntatore alla porzione "condivisa" dell'oggetto. Alle variabili di istanza e ai metodi di *Bottom*, *A* e *B* si accede come nel caso già descritto per l'ereditarietà multipla senza condivisione. Per accedere alle variabili di istanza, o per invocare un metodo di *Top*, occorre invece un accesso indiretto preliminare (supponiamo di invocare *f*, l'*n*-esimo metodo della vtable di *Top* e che ogni indirizzo sia su *w* byte):

```

R1 := pa           // vista Bottom
R1 := * (R1 + w) // vista Top
R2 := * (R1)      // vtable di Top
R3 := * (R2 + (n - 1) × 2 × w) // indirizzo di f
R2 := * (R2 + (n - 1) × 2 × w + w) // correzione
this := R1 + R2
call * (R3)       // chiamata di h

```

La correzione necessaria per `this` è la differenza tra la vista della classe nella quale il metodo è stato dichiarato e la vista della classe nella quale il metodo è stato ridefinito.

In linguaggi che permettano situazioni più elaborate (per esempio una stessa superclasse è replicata in alcune sottoclassi e condivisa da altre) occorrono tecniche più sofisticate che non possiamo trattare in questa sede.

## 12.4 Polimorfismo e generici

Dopo aver studiato gli aspetti caratteristici del paradigma orientato agli oggetti e le relative implementazioni, in questo paragrafo presenteremo alcuni approfondimenti relativi alla nozione di sottotipo, con particolare riguardo alle relazioni che esistono con il polimorfismo e i generici. Inizieremo la discussione in modo generale, ma approfondiremo i generici nella versione di Java, che li supporta in modo molto flessibile.

### 12.4.1 Polimorfismo di sottotipo

Abbiamo introdotto la nozione di polimorfismo nel Paragrafo 10.7 dove abbiamo anche distinto tra due forme radicalmente diverse di questo fenomeno: l'overloading (o polimorfismo ad hoc) ed il polimorfismo universale, dove uno stesso valore ha un'infinità di tipi diversi, che si ottengono per istanziazione da uno schema di tipo generale. Abbiamo anche visto la nozione di polimorfismo di sottotipo: un valore esibisce polimorfismo di sottotipo quando ha un'infinità di tipi diversi, che si ottengono per istanziazione da uno schema di tipo generale, sostituendo ad un opportuno parametro i sottotipi di un tipo assegnato. Qui ci interessa approfondire quest'ultima nozione nel caso dei linguaggi orientati agli oggetti.

Quando un linguaggio ha una nozione di compatibilità "struturale" tra tipi, come appunto la relazione di sottotipo tra classi, è chiaro che si ha una qualche forma di polimorfismo, pur non completamente generale come il polimorfismo universale parametrico. Per la relazione di sottotipo, infatti, ogni istanza di una classe `A` ha per tipo anche tutte le superclassi di `A`. Questa proprietà è particolarmente interessante nel caso dei metodi. Consideriamo infatti un metodo (per semplicità imposteremo la discussione su metodi con un solo argomento):

```
B foo(A x) { ... }
```

In virtù della compatibilità di sottotipo, `foo` può ricevere come argomento un valore di una *qualsiasi sottoclasse* di `A`. Il codice di `foo` non ha bisogno di essere

adattato alle specifiche sottoclassi: la struttura stessa delle classi (e la relativa implementazione) assicura che le operazioni possibili per i valori di tipo `A` sono possibili anche per i valori delle sue sottoclassi. Il lettore riconoscerà certamente la situazione di polimorfismo: si tratta di polimorfismo di sottotipo perché non è completamente generale, ma è *limitato ai sottotipi di A*.

Usiamo il simbolo "`<:`" (già introdotto nel Paragrafo 10.7) per denotare la relazione di sottotipo: leggiamo `C <: D` come "`C sub D`" (`C` è un sottotipo di `D`). Usiamo anche la notazione con quantificatore universale introdotta nel riquadro di pag. 322. Con queste convenzioni possiamo scrivere il tipo del nostro metodo `foo` come

```
∀T<:A. T->B
```

che esprime il fatto che tale metodo può essere applicato a tutti i valori di una qualsiasi sottoclasse di `A`. Si tratta di una forma di polimorfismo di sottotipo implicito, perché l'istanziazione avviene automaticamente al momento dell'applicazione del metodo ad un valore della sottoclasse.

In realtà la notazione che usiamo è molto più espressiva di quello che è possibile scrivere nello pseudolinguaggio che abbiamo usato sin qui. Consideriamo un semplicissimo metodo che restituisce il suo argomento, di classe `A`:

```
A Ide(A x) { return x; }
```

Da un punto di vista semantico, possiamo dire che `Ide` ha tipo

```
∀T<:A. T->T
```

perché restituisce inalterato l'oggetto passato come argomento. Tuttavia, supposto che `C <: A` e che `c` sia un'istanza di `C`, il seguente assegnamento sarebbe rifiutato dal controllore *statico* dei tipi

```
C cc = Ide(c);
```

nonostante sia perfettamente sensato da un punto di vista semantico. Il punto è che i linguaggi comuni hanno un sistema di tipi poco astuto che non è in grado di riconoscere che `∀T<:A. T->T` è un tipo corretto per `Ide` (come invece farebbe l'inferenza di tipo alla ML, vedi il riquadro di pag. 325), né hanno un meccanismo linguistico col quale il programmatore possa esprimere che il tipo del risultato di un metodo dipende *dal tipo dell'argomento*. Tuttavia questo ragionamento garantisce che la correzione dinamica con un cast all'ingiù

```
C cc = (C) Ide(c);
```

non causerà mai errore a tempo d'esecuzione.

La notazione che abbiamo usato per i tipi, insomma, permette di esprimere delle relazioni tra il tipo dell'argomento e il tipo del risultato, che non sono possibili nel nostro pseudolinguaggio (né nel frammento di C++ o di Java che abbiamo analizzato sin qui).

Prima di vedere un modo col quale il linguaggio può essere esteso, presentiamo un altro esempio, questa volta un po' più significativo, di questa limitata forma di polimorfismo che è possibile realizzare con gli strumenti che abbiamo messo

```

class ELEM{
    Object info;
    ELEM prox;
}
class Stack{
    private ELEM top = null; //inizialmente pila vuota
    boolean isEmpty(){
        return top==null;
    }
    void push (Object o){
        ELEM ne = new ELEM();
        ne.info = o;
        ne.prox = top;
        top = ne;
    }
    Object pop() {
        Object tmp = top.info;
        top = top.prox;
        return tmp;
    }
}

```

Figura 12.12 Classi per pile basate su Object.

a punto sin qui. Vogliamo realizzare una pila di elementi, rappresentata come lista concatenata di elementi. Non vogliamo fissare subito il tipo degli oggetti che metteremo nella lista, così li dichiareremo di tipo Object. La Figura 12.12 mostra una possibile definizione di una classe Stack, costruita a partire da elementi generici. L'aspetto che ci interessa è che il metodo pop restituisce un Object (e non potrebbe fare altrimenti, visto che non abbiamo fatto alcuna ipotesi sul tipo degli elementi nella pila). Adesso possiamo usare le nostre pile, ma con una certa attenzione; sia C una qualsiasi classe:

```

C c;
Stack s = new Stack();
s.push(new C());
c = s.pop();           //errore di tipo

```

L'ultima linea è scorretta, perché cerchiamo di assegnare un Object ad una variabile di tipo C. Occorre forzare l'assegnamento con un cast (controllato dinamicamente):

```
c = (C) s.pop();       //controllo dinamico
```

Dopo questi esempi, potremmo accontentarci di osservare che tutto ciò non fa che rafforzare quanto già dicemmo nel Paragrafo 10.2.1: i linguaggi impongono di solito più vincoli di tipo di quelli semanticamente ragionevoli, al fine di garantire un controllo statico efficiente. Il prossimo paragrafo, tuttavia, discuterà

```

class ELEM<A>{
    A info;
    ELEM<A> prox;
}
class Stack<A>{
    private ELEM<A> top = null; //inizialmente pila vuota
    boolean isEmpty(){
        return top==null;
    }
    void push (A o){
        ELEM<A> ne = new ELEM<A>();
        ne.info = o;
        ne.prox = top;
        top = ne;
    }
    A pop() {
        A tmp = top.info;
        top = top.prox;
        return tmp;
    }
}

```

Figura 12.13 Classi per pile generiche.

di estensioni del linguaggio che permettono un polimorfismo di sottotipo *esplicito* più potente.

### 12.4.2 Generici in Java

Abbiamo già discusso nel Paragrafo 10.7 la nozione di *template* di C++: un frammento di programma dove alcuni tipi sono indicati come *parametri*, che possono poi essere opportunamente istanziati su tipi "concreti", dando luogo ad una forma assai interessante di polimorfismo. La versione 5 di Java introduce un concetto analogo (ma con potenzialità e implementazione assai diverse), che chiama *generici* e che presenteremo in questo paragrafo, con l'obiettivo di discutere le sue relazioni con i sottotipi.

In Java 5 possono essere generiche sia le definizioni di tipo (dunque classi e interfacce), che quelle di metodo. La sintassi usata è analoga a quella di C++ e utilizza le parentesi angolate per indicare il parametro. La Figura 12.13 riporta la versione "generica" delle stesse definizioni della Figura 12.12. Il tipo <A> tra parentesi angolate è il *parametro formale di tipo* della dichiarazione generica e servirà ad essere istanziato successivamente. Una specifica versione di Stack si ottiene specificando quale tipo deve essere sostituito al posto di A. Ad esempio, pile di stringhe e di interi (*Integer* è una classe che permette di vedere un intero come un oggetto, a differenza del tipo *int*, che è costituito dagli interi ordinari):

```

class Coppia<A,B>{
    private A a;
    private B b;
    Coppia(A x, B y){ //costruttore
        a=x; b=y;
    }
    A Primo(){ return a; }
    B Secondo(){ return b; }
}

```

**Figura 12.14** Copie generiche.

```

Stack<String> ss = new Stack<String>();
Stack<Integer> si = new Stack<Integer>();

```

I tipi che costituiscono i “parametri attuali” devono essere tipi classe o array<sup>18</sup>. Il cast dinamico che doveva essere aggiunto nella versione non generica non è ora più necessario:

```

Stack<String> ss = new Stack<String>();
ss.push(new String("pippo"));
String s = ss.pop();

```

La Figura 12.14 presenta un ulteriore semplice esempio: copie di elementi di due tipi qualsiasi. Una coppia può ovviamente essere istanziata su tipi specifici:

```

Integer i = new Integer(3);
String v = new String ("pippo");
Coppia<Integer,String> c = new Coppia<Integer,String>(3,v);
String w = c.Second();

```

Anche i metodi possono essere generici; supponiamo, ad esempio, di voler definire un metodo per costruire copie della diagonale (cioè con le due componenti uguali). Un primo tentativo potrebbe essere quello di sfruttare il polimorfismo di sottotipo che già è presente nel linguaggio e definire

```

Coppia<Object, Object> diagonale(Object x){
    return new Coppia<Object, Object>(x,x);
}

```

Tuttavia, in modo simile a quanto abbiamo già visto per il metodo pop poc’anzì, se applichiamo diagonale a una stringa, il risultato è solo una coppia di Object e non di String:

```

Coppia<Object, Object> co = diagonale(v);
Coppia<String, String> cs = diagonale(v); //errore di compilazione

```

Occorre parametrizzare la definizione di diagonale, in particolare il tipo del suo risultato, in funzione del tipo dell’argomento:

```

<T> Coppia<T,T> diagonale(T x){
    return new Coppia<T,T>(x,x);
}

```

La prima coppia di parentesi angolate introduce una *variabile di tipo*, il parametro (formale) che può poi venire utilizzato nella definizione del metodo. Il lettore avrà certo riconosciuto in  $\langle T \rangle$  un quantificatore universale, scritto con notazione diversa: la definizione sta esprimendo che diagonale ha tipo

$\forall T. T \rightarrow \text{Coppia} < T, T >$ .

La nostra diagonale può essere usata anche senza istanziarla esplicitamente:

```

Coppia<Integer, Integer> ci = diagonale(new Integer(4));
Coppia<String, String> cs = diagonale(new String("pippo"));

```

Il compilatore esegue una vera e propria inferenza di tipo (Paragrafo 10.8), in generale assai più complessa di quella, elementare, necessaria nell’esempio (nel nostro caso si riconosce che nella prima chiamata T deve essere sostituito con Integer, mentre nella seconda è sostituito con String).

Un aspetto molto importante dei parametri di tipo (sia nelle definizioni di tipi che in quelle di metodi) è che possono avere dei *limiti*, cioè specificare che solo sottotipi di determinate classi sono permessi. Illustriamo questa caratteristica con un esempio. Supponiamo di disporre di un’interfaccia per forme geometriche che possono essere disegnate; abbiamo poi varie classi specifiche che la implementano:

```

interface Forma{
    void disegna();
}
class Cerchio implements Forma{
    ...
    public void disegna() {...}
}
class Rombo implements Forma{
    ...
    public void disegna() {...}
}

```

Sfruttiamo ora una libreria standard di Java (`java.util`): abbiamo una lista di forme, cioè un oggetto di tipo `List<Forma>` e vogliamo invocare il metodo `disegna` su ciascun elemento della lista. La prima idea sarà quella di scrivere<sup>19</sup>

<sup>18</sup> Devono essere tipi riferimento, per i motivi di implementazione cui abbiamo accennato a pag. 326.

<sup>19</sup> Il corpo del metodo è un esempio di *for-each* (vedi il Paragrafo 8.3.3), un costruttore iterativo che applica il corpo a tutti gli elementi di una collezione (lista, array ecc.): in questo caso, per ogni f di forme, chiama il metodo `disegna`.

```
void disegnaTutti(List<Forma> forme) {
    for(Forma f : forme)
        f.disegna();
}
```

La definizione è corretta, ma il metodo può essere applicato solo ad argomenti di tipo `List<Forma>` (e non, per esempio, a `List<Rombo>`). Il motivo è che `Lista<Rombo>` *non* è un sottotipo di `Lista<Forma>`, per i motivi che discuteremo tra poco nel Paragrafo 12.4.4<sup>20</sup>.

Per ovviare al problema, possiamo rendere parametrica la definizione del metodo `disegnaTutti`; ovviamente non possiamo permettere come argomento una lista arbitraria, perché deve essere composta da elementi sui quali si possa chiamare il metodo `disegna`. Il linguaggio ci consente di specificare questo fatto nel modo seguente:

```
<T extends Forma> void disegnaTutti(List<T> forme) {
    for(Forma f : forme)
        f.disegna();
}
```

In questo caso il parametro formale di tipo non è un tipo qualsiasi, ma un tipo che estende `Forma` (qui “estende” è usato come sinonimo di “è sottotipo di”; in questa accezione, `Forma` estende se stesso). Usando la nostra notazione con i quantificatori universali, il tipo di `disegnaTutti` diviene

```
 $\forall T : \text{Forma} \cdot \text{List}\langle T \rangle \rightarrow \text{void}$ 
```

Ora `disegnaTutti` può essere chiamata su una lista i cui elementi appartengano ad un qualsiasi sottotipo di `Forma` (`Forma` è ovviamente un sottotipo di se stesso):

```
List<Rombo> lr = ...;
List<Forma> lf = ...;
disegnaTutti(lr);
disegnaTutti(lf);
```

Il meccanismo dei limiti sulle variabili di tipo è assai sofisticato e flessibile. In particolare, la variabile di tipo può comparire anche nel suo stesso limite. Ci limiteremo ad un solo esempio di questa possibilità, rimandando il lettore alla bibliografia per una discussione più approfondita<sup>21</sup>.

Gli elementi di una classe sono tra loro confrontabili se la classe implementa l’interfaccia `Comparable`. Vogliamo definire un metodo che, presa come argomento una lista di elementi di tipo generico, restituisce il massimo elemento di questa lista. Quale segnatura possiamo dare a questo metodo `max`? Il primo tentativo è

<sup>20</sup>Per il momento il lettore si accontenti di sapere che, se `A <: B`, e `DefPara<T>` è una qualche definizione parametrica di tipo (come `List<T>`), `DefPara<A>` e `DefPara<B>` non sono in relazione tra loro nella gerarchia di sottotipo.

<sup>21</sup>La possibilità che una variabile di tipo compaia nel suo stesso limite è nota nella letteratura come polimorfismo *F-bounded* ed è usata soprattutto per la tipizzazione dei metodi *binari*, cioè che hanno un parametro dello stesso tipo dell’oggetto che riceve il metodo.

### Wildcard

Il metodo `disegnaTutti` poteva essere scritto in modo più compatto ed elegante usando un *segnaposto* (o *jolly*, o *wildcard*): il carattere `?` (che si legge “sconosciuto”) sta per “un qualsiasi tipo” e può essere usato nelle definizioni generiche. Ad esempio, un valore di tipo `List<?>` è una lista di elementi di cui non si conosce il tipo. Si potrebbe pensare che scrivere `List<?>` sia la stessa cosa che scrivere `List<Object>`, ma così non è, perché `List<Object>` non è un supertipo, per esempio, di `List<Integer>`, mentre `List<?>` è proprio il supertipo di tutti i tipi `List<A>`, per ogni `A`.

Usando i segnaposto, il metodo `disegnaTutti` si poteva scrivere

```
void disegnaTutti(List<? extends Forma> forme) {
    for(Forma f : forme)
        f.disegna();
}
```

In generale, ogni segnaposto può essere sempre sostituito con un parametro esplicito; da un punto di vista pragmatico, di chiarezza del codice, si userà un segnaposto quando il parametro sarebbe usato solo una volta (come nel caso di `disegnaTutti`), mentre si dovrà usare un parametro esplicito quando la variabile di tipo è usata più di una volta (come è il caso di `diagonale`, dove la variabile è usata nel tipo del *risultato* del metodo).

Se due segnaposto compaiono nello stesso costrutto, devono essere considerati come variabili tra loro diverse.

```
public static <T extends Comparable<T>>
    T max(List<T> lista)
```

che esprime il fatto che gli elementi della lista devono essere confrontabili con elementi dello stesso tipo. Cerchiamo ora di usare `max`. Abbiamo un tipo `Foo` che permette il confronto con oggetti arbitrari:

```
class Foo implements Comparable<Object>{...}
List<Foo> cf = ....;
```

e invochiamo ora `max(cf)`: ogni elemento in `cf` (è un `Foo` e dunque) è confrontabile con qualsiasi oggetto, in particolare con ogni `Foo`. Ma il compilatore rigetta la chiamata: `T` dovrebbe essere `Foo`, ma `Foo` non implementa `Comparable<Foo>`. In realtà basta che `Foo` sia confrontabile con uno dei suoi supertipi:

```
public static <T extends Comparable<? super T>>
    T max(List<T> lista)
```

Adesso, nelle stesse condizioni di prima, `max(cf)` è corretta, perché `Foo` implementa `Comparable<Object>`.

### 12.4.3 Implementazione dei generici in Java

A differenza dei template di C++, che vengono risolti a tempo di linking mediante una duplicazione e specializzazione del codice, dei generici di Java esiste sempre

un'unica copia: anche nell'implementazione è rispettata l'idea del polimorfismo parametrico che esiste un solo valore (una sola classe, un solo metodo ecc.) che appartiene a tanti tipi diversi (e, nel caso dei metodi, funziona "uniformemente" nei vari tipi).

I generici sono implementati a livello di compilazione mediante un meccanismo di *cancellazione*: un programma che contenga generici viene prima di tutto sottoposto al controllore dei tipi. Quando questo controllo di semantica statica ha verificato che tutto è a posto, il programma originale viene trasformato in un programma analogo dove tutti i generici sono stati cancellati: tutte le informazioni tra parentesi angolate sono eliminate (per esempio, ogni `List<Integer>` diviene `List`); tutti gli altri usi delle variabili di tipo sono sostituiti con il limite superiore della variabile stessa (in genere si tratta di `Object`); infine, se dopo queste manipolazioni il programma "cancellato" fosse scorretto rispetto ai tipi, sono inseriti degli opportuni cast (dinamici). Con questo procedimento l'uso dei generici non provoca aggravio (nella maggior parte dei casi) né nella dimensione del codice né nel tempo d'esecuzione. Forse più importanti sono altre due conseguenze di questa implementazione:

- la macchina astratta sottostante (la JVM) non ha bisogno di essere modificata; l'aggiunta dei generici non comporta modifiche da replicare per architetture diverse, ma è localizzata nel compilatore;
- è possibile mescolare codice generico e codice non generico in modo relativamente semplice e, soprattutto, sicuro: eventuali "buchi" nel sistema dei tipi statico che si dovessero presentare localmente dall'uso simultaneo di codice generico e non generico saranno rilevati a tempo d'esecuzione per effetto dei cast dinamici inseriti dal compilatore durante il procedimento di cancellazione.

#### 12.4.4 Generici, array e gerarchia di sottotipo

Abbiamo già osservato che, in Java, una definizione di tipo generica `DefPara<T>` *non* preserva la gerarchia di sottotipo: dati due tipi `A` e `B` con `A < : B`, `DefPara<A>` e `DefPara<B>` non sono in relazione tra loro nella gerarchia di sottotipo. Vogliamo adesso chiarire alcune motivazioni di questa scelta, anche alla luce del comportamento contrastante che Java mantiene nei confronti degli array: se `A < : B`, `A[]` è sottotipo di `B[]`! Dopotutto gli array sono una forma primitiva di costrutto generico (l'unico costrutto "array" viene specializzato su array di uno specifico tipo): perché i due meccanismi si comportano in modo tanto diverso?

Iniziamo col discutere perché le definizioni generiche non preservano i sottotipi; consideriamo il seguente frammento, che usa le definizioni generiche della Figura 12.13:

```
Stack<Integer> si = new Stack<Integer>();
Stack<Object> so = si; // scorretta per Java
```

Supponiamo, contrariamente ai fatti, che le definizioni di generico preservino i tipi, cioè che `Stack<Integer> < : Stack<Object>`. La seconda linea del frammento è ora legale: abbiamo due riferimenti diversi (e di tipo diverso) ad un'unica pila (di interi). Continuiamo il nostro codice:

#### Funzioni covarianti e controvarianti

Sia  $D$  un insieme sul quale è definita una relazione di preordine, denotata da  $\leq$  (si ricordi il riquadro di pag. 314). Una funzione  $f : D \rightarrow D$  è *covariante*, qualora  $f$  rispetti il preordine, cioè  $x \leq y$  implica  $f(x) \leq f(y)$ . Una funzione è *controvariante*, qualora rovesci il preordine, cioè  $x \leq y$  implica  $f(x) \geq f(y)$ .

Se la relazione è in realtà un ordine parziale, la terminologia più comune in matematica è quella di funzione *monotona crescente* al posto di covariante e *antimonotona*, o monotona descrescente, per controvariante. Nel contesto di tipi e sottotipi, viene sempre usata la terminologia covariante-controvariante (che è presa a prestito dalla teoria delle categorie).

```
so.push(new String("pluto"));
Integer i = si.pop(); // pericolo!
```

Nelle ipotesi fatte, entrambe le linee sono corrette per i tipi: essendo `so` uno stack di `Object`, possiamo inserirvi un oggetto qualsiasi, per esempio una stringa. D'altra parte, essendo `si` uno stack di interi, prelevando un elemento si ottiene un intero. Ma si tratta di un'evidente violazione della sicurezza del sistema di tipi: siccome la sorgente del problema è che `Stack<Integer> < : Stack<Object>`, è proprio questa relazione che deve essere abbandonata. Il tipo `Stack<Object>` non è dunque il supertipo comune di tutte le pile specifiche `Stack<A>`. D'altra parte, la pragmatica suggerisce che un tale supertipo esista nel linguaggio, perché altrimenti troppi esempi di programmazione risulterebbero difficili, se non impossibili, da scrivere. È per questo che è introdotto il segnaposto (*wildcard*) `? : Stack<?>` è supertipo di ogni pila specifica.

L'idea intuitiva che `A < : B` implichi `DefPara<A> < : DefPara<B>`, è sbagliata perché non tiene conto del fatto che le collezioni possono cambiare nel tempo. Una volta che una `Stack<Integer>` è divenuta una `Stack<Object>` non è più possibile controllare staticamente che le sue modifiche siano consistenti con la sua struttura originale.

Veniamo ora al problema degli array. Se sostituiamo gli array alle pile nel nostro frammento di poc'anzi si ottiene, *mutatis mutandis*:

```
Integer[] ai = new Integer[10];
Object[] ao = ai; // corretto: Integer[] < : Object[]
ao[0] = new String("pluto"); // corretto; errore a run time
```

Il frammento è staticamente corretto, perché gli array in Java preservano i sottotipi (teoricamente si dice che sono un costrutto *covariante*), ma il compilatore, per garantire la sicurezza rispetto ai tipi, è costretto ad inserire un controllo dinamico di tipo in corrispondenza dell'ultima linea che, nel caso dell'esempio, causerà un errore durante l'esecuzione, perché stiamo cercando di memorizzare una stringa in una variabile di tipo `Integer`.

Perché dunque inserire nel linguaggio array covarianti se questi impongono controlli dinamici (che rimangono comunque controintuitivi quanto la non covarianza dei generici)? Il punto è che gli array covarianti permettono qualche forma

limitata di polimorfismo. Consideriamo ad esempio il problema di scambiare tra loro i primi due elementi di un vettore arbitrario. Una possibile soluzione è:

```
public void swap(Object[] vett) {
    if (vett.length > 1) {
        Object temp = vett[0];
        vett[0] = vett[1];
        vett[1] = temp;
    }
}
```

Il metodo `swap` può essere chiamato su array arbitrari, in virtù della covarianza.

Gli array covarianti sono presenti in Java sin dai primi momenti del progetto, ben prima che i progettisti si ponessero il problema dei generici. In retrospettiva, tuttavia, e soprattutto alla luce dell'introduzione dei generici, la covarianza degli array deve essere considerata un aspetto poco riuscito del progetto di Java.

#### 12.4.5 Overriding covarianti e controvarianti

Concludiamo lo studio della relazione di sottotipo discutendo quale tipo sia permesso nelle ridefinizioni dei metodi. Java e C++ richiedono che, quando si ha ridefinizione, vi sia identità tra i tipi degli argomenti del metodo ridefinito. Riprendiamo il nostro pseudolinguaggio neutro e sia `C` un tipo fissato; data una classe come

```
class F{
    C fie (A p) {...}
}
```

una sottoclasse di `F` può ridefinire (`override`) `fie` solo con un metodo che prende come argomento un `A`. Se i tipi sono diversi, come per esempio in

```
class G extending F{
    C fie (B p) {...}
}
```

non si ha ridefinizione, ma semplicemente la definizione di due metodi overlaoded.

Lo stesso non accade per il tipo del *risultato* del metodo: sia C++ che Java<sup>22</sup> permettono al tipo del risultato del metodo ridefinito nella sottoclasse di essere un sottotipo del tipo corrispondente della superclasse. Supponendo che `D` sia un sottotipo di `C` (`D <: C`):

```
class E extending F{
    D fie (A p) {...}
}
```

<sup>22</sup>Sino alla versione 4, Java richiedeva che anche il tipo del risultato di un metodo ridefinito dovesse coincidere col tipo del risultato del metodo della superclasse.

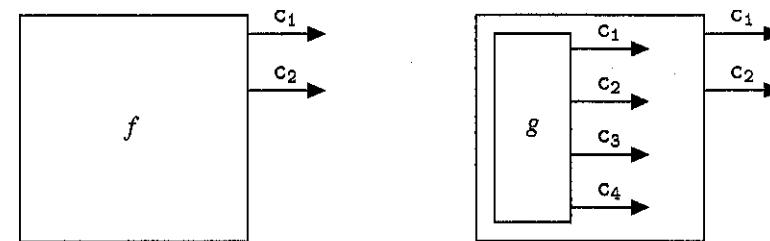


Figura 12.15 Un'istanza `f` di `F` e un'istanza `g` di `G` travestita da `F`.

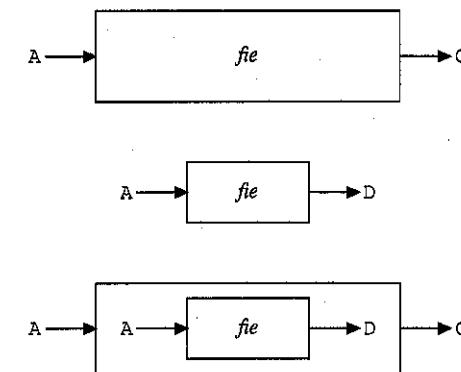
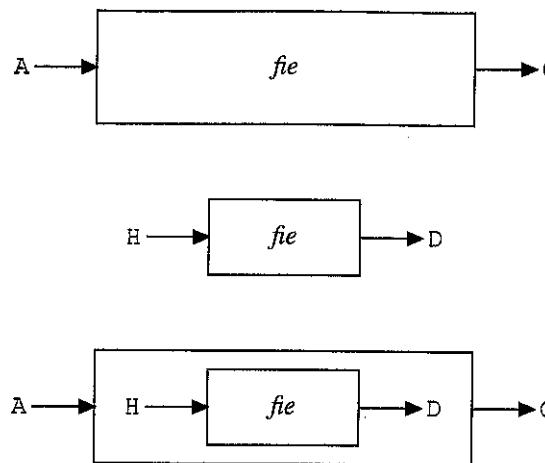


Figura 12.16 Ridefinizione di un metodo: covarianza del tipo del risultato.

in `E` (che è sottotipo di `F`: `E <: F`) il metodo `fie` è ridefinito rispetto ad `F`. Il lettore non avrà difficoltà a convincersi da solo che questa estensione è legittima semanticamente e non causa errori di tipizzazione. Possiamo tuttavia aiutarlo in questo ragionamento con qualche considerazione generale.

**Supertipi e viste** In un linguaggio con polimorfismo di sottotipo, un tipo di un oggetto può essere interpretato come una particolare *vista* su quell'oggetto, ovvero, in modo un po' più colorato, come un "travestimento" dell'oggetto. La Figura 12.15 rappresenta graficamente questa idea nel caso di due tipi `G <: F` (`F` è una classe di oggetti che rispondono ai metodi `c1` e `c2`; gli oggetti di `G` rispondono a tutti i metodi di `F` più `c3` e `c4`).

Se applichiamo questa idea (informale, ma che rispetta in pieno la semantica dei sottotipi) alla ridefinizione di un metodo, otteniamo la situazione della Figura 12.16, dove il nostro metodo `fie` di poc'anzi, definito in `F` con segnatura `fie:A->C` viene ridefinito in una sottoclasse con segnatura `fie:A->D`. La figura rende immediata l'osservazione che siamo in una situazione corretta rispetto ai



**Figura 12.17** Ridefinizione di un metodo: controvarianza del tipo dell'argomento.

tipi proprio qualora  $D < : C$ . Infatti, il metodo `f1e` ridefinito produce un valore di tipo  $D$ ; se  $D < : C$ , questo valore è anche un valore di  $C$  e dunque i tipi sono rispettati.

Possiamo riassumere questa discussione in un principio generale: rispetto al tipo del *risultato* di un metodo, l'overriding è semanticamente corretto (e permesso in molti linguaggi) quando è *covariante*: il risultato del metodo ridefinito in una sottoclasse, è un *sottotipo* del risultato del metodo originale.

Questo ragionamento, tuttavia, può essere facilmente applicato anche al tipo dell'*argomento* di un metodo. Lo schema della Figura 12.16 può infatti essere generalizzato in quello della Figura 12.17, dove il metodo `f1e` viene ridefinito con segnatura `f1e: H -> D`. Quale relazione deve valere tra  $H$  e  $A$  affinché la situazione sia semanticamente corretta? La figura dà una risposta immediata: deve essere  $A < : H$ . Rispetto al tipo dell'*argomento* di un metodo, l'overriding è semanticamente corretto quando è *controvariante*: l'argomento del metodo ridefinito in una sottoclasse, è un *supertipo* dell'argomento del metodo originale.

Da un punto di vista semantico, dunque, il tipo dei metodi  $S \rightarrow T$  è sottotipo di  $S' \rightarrow T'$  ( $S \rightarrow T <: S' \rightarrow T'$ ) qualora  $S < : S'$  e  $T' < : T$ : il tipo dei metodi è covariante nel tipo del risultato e controvariante nel tipo dell'argomento.

Per quanto semanticamente corretto, l'overriding controvariante (sull'argomento dei metodi) è controintuitivo in diverse situazioni. Il caso più importante è quello dei cosiddetti *metodi binari*, cioè metodi con un parametro dello stesso tipo dell'oggetto che riceve il metodo. Il caso tipico è quello di un metodo di uguaglianza, o di confronto. Supponiamo di avere una classe come la seguente

```

class Point{
    ...
    boolean eq(Point p){...}
}

```

e di volerla ora specializzare in una sottoclasse:

```

class ColoredPoint extending Point{
    ...
    boolean eq(ColoredPoint p){...}
}

```

Ebbene, secondo la regola controvariante la classe `ColoredPoint` non è una sottoclasse di `Point`, perché l'argomento di `eq` sarebbe un sottotipo (e non un supertipo) del tipo dell'argomento di `eq` in `Point`.

Per questi (e altri) motivi, l'overriding controvariante è poco diffuso (uno dei pochi linguaggi che lo adottano è Emerald), mentre molti tra i linguaggi più diffusi (C++ e Java in testa) richiedono identità del tipo dell'argomento.

Vi sono tuttavia linguaggi che sacrificano la sicurezza statica rispetto ai tipi ed adottano la regola (semanticamente scorretta) di overriding *covariante* per il tipo dell'argomento dei metodi. Tra questi vi sono linguaggi abbastanza diffusi, come Eiffel e il linguaggio di O<sub>2</sub> (uno tra i più venduti sistemi database orientati agli oggetti). La motivazione è la supposta maggiore naturalezza, in specie per i metodi binari; sperimentazioni estese con O<sub>2</sub>, tra l'altro, mostrano che la non correttezza semantica del controllo dei tipi non causa mai problemi nella pratica (cioè in situazioni che non siano costruite ad hoc per violare il sistema di tipi).

## 12.5 Sommario del capitolo

Il capitolo ha fornito una presentazione generale, ma approfondita, del paradigma orientato agli oggetti, introdotto come un modo per ottenere astrazione sui dati in modo più flessibile ed estensibile di quello possibile con i tipi di dato astratti.

Abbiamo caratterizzato il paradigma orientato agli oggetti quando siamo in presenza di:

- *incapsulamento* dei dati;
- una relazione di compatibilità tra tipi che chiamiamo *sottotipo*;
- un modo per riusare codice che chiamiamo *ereditarietà* e che si distingue in singola e multipla;
- un meccanismo per la *selezione dinamica* dei metodi.

Questi quattro concetti sono stati inquadrati principalmente nel contesto dei linguaggi basati su *classi*, anche se abbiamo accennato ai linguaggi basati su *delega*. Abbiamo poi discusso l'*implementazione* dell'ereditarietà singola e multipla e della selezione dinamica dei metodi.

Il capitolo si è concluso con lo studio di alcuni aspetti rilevanti dei sistemi di tipi dei linguaggi orientati agli oggetti:

- il *polimorfismo di sottotipo*;
- un'altra forma di polimorfismo parametrico che è possibile quando il linguaggio ammetta *generici* (che abbiamo studiato nella versione di Java 5);
- il problema dell'overriding dei metodi covariante o controvariante.

Il paradigma orientato agli oggetti, oltre che una specifica classe di linguaggi, identifica anche una vera e propria metodologia di sviluppo di sistemi software, codificata da regole semiformali e basata sull'organizzazione dei concetti mediante oggetti e classi. Ovviamente i due aspetti, quello linguistico e quello metodologico, non sono completamente separati dato che la metodologia di sviluppo orientata agli oggetti trova una sua naturale applicazione nell'uso dei linguaggi orientati object oriented. Tuttavia gli aspetti metodologici di solito sono trattati in modo autonomo, nell'ambito dell'ingegneria del software, senza far riferimento ad alcun linguaggio specifico. In questo testo dunque non approfondiamo gli aspetti metodologici della programmazione orientata agli oggetti, per i quali rimandiamo ai testi di ingegneria del software (si veda anche la nota bibliografica qui sotto).

## 12.6 Nota bibliografica

Il primo linguaggio orientato agli oggetti è Simula 67 [14, 74], che abbiamo già citato nel capitolo precedente. Sarà però Smalltalk (che esplicitamente dipende da Simula) ad avere la maggiore influenza sui linguaggi successivi e ad introdurre la bella metafora antropomorfa dei messaggi inviati agli oggetti [39].

Su C++ il riferimento canonico è [96]; su Java si veda la definizione del linguaggio [41] e il testo, più divulgativo, del suo autore principale [10]. Un'introduzione alla Java Virtual Machine (e dunque a come i vari meccanismi siano implementati in Java) si può trovare in [66]; per la specifica ufficiale della JVM si veda [56].

La nostra descrizione dell'implementazione dell'ereditarietà multipla è basata su quella di C++ [95]. L'introduzione dei generici in Java è il risultato di molti lavori di ricerca sul polimorfismo di sottotipo. Un'introduzione ai generici così come appaiono in Java 5 è in [16], che abbiamo ampiamente seguito nella nostra presentazione.

Sul progetto di Self si veda l'articolo originale [100]; la retrospettiva [93] è una bella lettura sui criteri di progetto di un linguaggio di programmazione innovativo. Sulla diatriba tra overriding covariante e controvariante si veda [22].

Infine per quanto riguarda la metodologia di sviluppo software orientata agli oggetti si può consultare un qualsiasi testo di ingegneria del software di carattere generale, quale ad esempio il classico [82].

## 12.7 Esercizi

- Si prendano in considerazione le classi della Figura 12.8 e le ulteriori definizioni:

```
class E{
    int v;
    void n() {...}
}
```

```
class D extending E, C{
    int w;
    int g(){return x + y + v;} // ridefinito rispetto a C
    void m(){...}
}
```

Si disegni la rappresentazione di un oggetto istanza di D, insieme alla struttura delle vtable di tale classe, indicando per ogni metodo l'opportuno valore necessario a correggere il valore di this.

- Sono date le seguenti definizioni nel nostro pseudolinguaggio:

```
abstract class A {
    int val = 1;
    int foo (int x);
}

abstract class B extending A {
    int val = 2;
}

class C extending B {
    int n = 0;
    int foo (int x){ return x+val+n;}
}

class D extending C {
    int n;
    D(int v){n=v;}
    int foo (int x){return x+val+n;}
}
```

Si consideri ora il seguente frammento di programma

```
int u, v, w, z;
A a;
B b;
C c;
D d = new D(3);
a = d;
b = d;
c = d;
u = a.foo(1);
v = b.foo(1);
w = c.foo(1);
z = d.foo(1);
```

Si dia il valore di u, v, w e z al termine dell'esecuzione.

- Sono date le seguenti definizioni Java:

```
interface A {
    int val=1;
    int foo (int x);
}

interface B {
    int z=1;
    int fie (int y);
}

class C implements A, B {
    int val = 2;
```

```

int z = 2;
int n = 0;
public int foo (int x){ return x+val+n; }
public int fie (int y){ return z+val+n; }
}
class D extends C {
    int val=3;
    int z=3;
    int n=3;
    public int foo (int x){return x+val+n; }
    public int fie (int y){ return z+val+n; }
}

```

Si consideri ora il seguente frammento di programma

```

int u, v, w, z;
A a;
B b;
D d = new D();
a = d;
b = d;
System.out.println(u = a.foo(1));
System.out.println(v = b.fie(1));
System.out.println(w = d.foo(1));
System.out.println(z = d.fie(1));

```

Si dia il valore di u, v, w e z al termine dell'esecuzione.

4. È corretto il seguente frammento Java? In caso positivo, il metodo fie è ridefinito (overridden)? Cosa stampa?

```

class A {
    int x = 4;
    int fie (A p) {return p.x;}
}

class B extends A{
    int y = 6;
    int fie (B p) {return p.x+p.y;}
}

public class binmeth {
    public static void main (String [] args) {
        B b = new B();
        A a = new A();
        int zz = a.fie(a)+ b.fie(a) ;
        System.out.print(zz);
    }
}

```

## Il paradigma funzionale

Presentiamo in questo capitolo gli aspetti principali del paradigma di programmazione funzionale, nel quale la computazione avviene per riscrittura di funzioni e non per modifica dello stato. La caratteristica fondamentale dei linguaggi di questo paradigma, almeno nella loro versione "pura", è proprio quella di non possedere il concetto di memoria (e, dunque, di effetto collaterale): fissato un ambiente, un'espressione denota sempre lo stesso valore.

Discuteremo il paradigma puro nelle prime sezioni, evidenziandone gli aspetti fondamentali. I linguaggi di programmazione funzionale, tuttavia, immagazzinano questi ingredienti "puri" in un contesto che aggiunge molti altri meccanismi, che passeremo in rassegna nel Paragrafo 13.3.

Accenneremo quindi alla macchina SECD, una macchina astratta per linguaggi funzionali di ordine superiore che costituisce il prototipo di molte implementazioni reali.

Saremo a questo punto nella posizione di discutere i motivi che rendono interessante il paradigma di programmazione funzionale, in rapporto agli ordinari linguaggi imperativi.

Conclude il capitolo un paragrafo più teorico che fornisce una succinta introduzione al  $\lambda$ -calcolo, un sistema formale per la calcolabilità al quale tutti i linguaggi funzionali si ispirano e che ha costituito, sin dai tempi di ALGOL e LISP, un modello costante per il progetto dei linguaggi di programmazione.

### 13.1 Computazioni senza stato

Per quanto evoluti ed astratti, tutti i linguaggi di programmazione convenzionali basano il proprio modello di computazione sulla trasformazione dello *stato*. Cuore di tale modello è la nozione di *variabile modificabile*, cioè di un contenitore con un nome al quale, durante la computazione, possono essere assegnati valori diversi, mantenendo sempre la stessa associazione nell'ambiente. Corrispondentemente, il costrutto principale dei linguaggi convenzionali è l'*assegnamento*, che modifica il valore contenuto in una variabile (ma non modifica l'associazione tra il nome della variabile e la locazione che gli corrisponde: modifica il r-valore, ma non

il l-valore, che è fissato una volta per tutte al momento della dichiarazione della variabile).

I linguaggi convenzionali, a partire da quello della più semplice macchina fisica, per arrivare ad un sofisticato linguaggio di ultima generazione, differiscono nel grado di astrazione, nei tipi, nei costrutti che permettono di manipolare le variabili, ma tutti condividono questo stesso modello di calcolo, che è una visione astratta della macchina fisica convenzionale sottostante: la computazione procede modificando valori memorizzati in locazioni. Si tratta di un modello importanzissimo che va sotto il nome di *Macchina di von Neumann*, dal nome del fisico-matematico ungherese-americano che negli anni quaranta del secolo scorso intuì che la macchina di Turing (vedi riquadro di pag. 139) poteva essere ingegnerizzata in un prototipo fisico, dando così origine ai calcolatori moderni.

Non si tratta tuttavia dell'unico modello possibile per ottenere un linguaggio di programmazione nel quale si possano esprimere tutte le funzioni calcolabili (cioè che sia Turing completo, Capitolo 5). È possibile calcolare senza far uso di variabili modificabili, cioè senza far riferimento alla nozione di stato. La computazione procede non più per modifica dello stato, ma mediante *riscrittura* di espressioni, cioè mediante modifiche che hanno luogo solo nell'ambiente e non comportano la nozione di memoria. Se non vi sono variabili modificabili, non vi è più bisogno dell'assegnamento, cioè del comando principale dei linguaggi convenzionali. Tutta la computazione sarà espressa mediante una modifica sofisticata dell'ambiente, nella quale la possibilità di manipolare funzioni (di ordine superiore, vedi il Paragrafo 9.2), gioca un ruolo fondamentale.

Senza assegnamento anche l'iterazione perde il proprio senso: un ciclo non fa altro che modificare ripetutamente lo stato fin quando il valore di determinate variabili soddisfa una guardia. Il lettore sa già (Capitolo 8) che i costrutti iterativi e la ricorsione sono i due meccanismi che permettono computazioni di lunghezza illimitata (ed eventualmente computazioni divergenti). Nel modello di computazione senza stato, scomparsa l'iterazione rimane la ricorsione, che dunque è il costrutto fondamentale per il controllo di sequenza.

Funzioni, ordine superiore e ricorsione sono gli ingredienti fondamentali di questo modello di computazione senza stato. Vista la centralità della nozione di funzione (sia come funzione matematica che come sottoprogramma), non sorprenderà che i linguaggi di programmazione che presuppongono questo modello siano detti *linguaggi funzionali* ed il paradigma che ne risulta, paradigma di programmazione funzionale.

Si tratta di un paradigma antico (ovviamente sempre nella scala dei tempi dell'informatica...) tanto quanto quello imperativo: accanto alla macchina di Turing esisteva sin dagli anni trenta il lambda-calcolo ( $\lambda$ -calcolo), un modello astratto di caratterizzazione delle funzioni calcolabili basato proprio sui concetti che abbiamo brevemente delineato. LISP è un linguaggio di programmazione esplicitamente ispirato al  $\lambda$ -calcolo, e molti sono seguiti negli anni (Scheme, ML con tutti i suoi vari dialetti, Miranda, Haskell, per citare solo i più diffusi). Tra questi solo Miranda e Haskell sono "funzionali puri": gli altri hanno anche delle componenti imperative (spesso per facilitare il programmatore avvezzo ai linguag-

gi convenzionali), ma la struttura del linguaggio assegna a tali caratteristiche un ruolo subalterno.

In questo capitolo forniremo un'introduzione non superficiale al paradigma funzionale puro, discutendone diversi aspetti generali. Per concretezza di esposizione, si è preferito far qui riferimento ad un linguaggio specifico, piuttosto che al solito pseudolinguaggio neutro. Si è scelto ML che, per coerenza ed eleganza del progetto, è più adatto degli altri ad una presentazione didattica. Non è nostro scopo introdurre il linguaggio (per il quale rimandiamo ai testi in bibliografia), ma solo usare la sintassi in modo strumentale per discutere di alcune questioni generali che si applicano anche ad altri linguaggi funzionali.

### 13.1.1 Espressioni e funzioni

Nell'usuale pratica matematica c'è qualche ambiguità tra la definizione di una funzione e la sua applicazione ad un valore. Non è raro trovare frasi della forma:

Sia  $f(x) = x^2$  la funzione che associa ad  $x$  il suo quadrato. Sia ora  $x = 2$ : ne segue che  $f(x) = 4$ .

L'espressione sintattica  $f(x)$  è usata per denotare due cose tra loro assai diverse: la prima volta serve ad introdurre il *nome*  $f$  per una specifica funzione; la seconda serve ad indicare il risultato dell'*applicazione* di tale funzione  $f$  ad uno specifico valore. Nella pratica del matematico tale ambiguità è del tutto innocua, perché il contesto permette di distinguere chiaramente quale dei due usi sia inteso. Lo stesso non accade in un linguaggio artificiale per descrivere funzioni (qual è un linguaggio di programmazione), in cui è opportuno distinguere i due casi.

Quando il matematico afferma di star definendo la funzione  $f(x)$ , in realtà sta definendo la funzione  $f$ , con un parametro *formale*  $x$  che serve ad indicare quale trasformazione  $f$  applica al proprio argomento. Per distinguere linguisticamente tra nome e "corpo" della funzione, seguendo la sintassi di ML possiamo scrivere

`val f = fn x => x*x;`

La parola riservata `val` introduce una dichiarazione, con la quale l'ambiente viene esteso con nuova associazione tra un nome e un valore; in questo caso il nome `f` è legato a quella funzione che trasforma `x` in `x*x`. In tutti i linguaggi funzionali le funzioni sono valori *esprimibili*, cioè possono essere il risultato della valutazione di un'espressione complessa: nel nostro caso l'espressione a destra di `=`, introdotta da `fn`, è appunto un'espressione che denota una funzione.

Per l'applicazione di una funzione ad un argomento manteniamo la notazione tradizionale, scrivendo `f(2)`, o `(f 2)`, o anche `f 2`, per l'espressione risultante dall'applicazione della funzione `f` all'argomento 2. Possiamo ancora usare l'operatore `val` per introdurre nuovi nomi, come in

`val quattro = f 2;`

L'introduzione di specifica sintassi per un'espressione che denota una funzione ha un'importante conseguenza: è possibile scrivere (ed eventualmente applicare) una funzione senza dover necessariamente assegnarle un nome. Ad esempio, l'espressione

```
(fn y => y+1) (6);
```

ha valore 7, che risulta dall'applicazione della funzione (anonima) `fn y => y+1` all'argomento 6. Per rendere meno pesante la notazione assumeremo (come fa ML) che l'applicazione possa essere denotata dalla semplice giustapposizione (cioè anche senza parentesi) e che essa associ a sinistra (notazione prefissa). Se `g` è il nome di una funzione, dunque, con

```
g a1 a2 ... ak
```

si indicherà

```
(...((g a1) a2)... ak).
```

Nulla vieta che un'espressione funzionale compaia all'interno di un'altra espressione funzionale, come in

```
val somma = fn x => (fn y => y+x);
```

Il valore `somma` è una funzione che, preso un argomento `x`, restituisce una funzione (anonima) che, preso a sua volta un argomento `y` restituisce `x+y`. Possiamo usare `somma` in molti modi diversi:

```
val tre = somma 1 2;
val sommadue = somma 2;
val cinque = sommadue 3;
```

Si osservi soprattutto `sommadue`: è una funzione che è ottenuta come risultato della valutazione di un'altra espressione.

La notazione che usa `val` e `fn` è molto importante, ma è un po' verbosa da utilizzare. ML permette di usare anche una notazione più compatta, che assomiglia al modo usuale di definire funzioni in un linguaggio di programmazione. La prima funzione `f` che abbiamo definito (quella che calcola il quadrato del suo argomento) potrebbe essere definita anche come

```
fun f x = x*x;
```

In generale una definizione della forma

```
fun F x1 x2 ... xn = corpo;
```

non è altro che zucchero sintattico (cioè, nel gergo dei linguaggi di programmazione, solo un'abbreviazione più gradevole) per

```
val F = fn x1 => (fn x2 => ... (fn xn => corpo) ...);
```

Come ulteriore esempio di funzioni che manipolano funzioni, stavolta sotto forma di parametro formale, vediamo la definizione

```
fun comp f g x = f(g(x));
```

che restituisce la funzione composta dei suoi primi due argomenti (a loro volta funzioni).

Infine, tutti i linguaggi funzionali ammettono la definizione di funzioni ricorsive. Supponendo di avere a disposizione un'*espressione condizionale* (che ML scrive con l'usuale sintassi `if then else`), possiamo definire il solito fattoriale come

```
fun fatt n = if n=0 then 1 else n*fatt(n-1);
```

### 13.1.2 Computazione come riduzione

Se si eccettuano le funzioni aritmetiche (che possiamo supporre predefinite con l'usuale semantica) e l'espressione condizionale, a livello concettuale possiamo descrivere il procedimento col quale si passa da un'espressione complessa ad un suo valore (*valutazione*) come un processo di *riscrittura*, che chiamiamo *riduzione*: in un'espressione complessa, una sottoespressione della forma "funzione applicata ad un argomento" viene testualmente sostituita con il corpo della funzione nel quale al posto del parametro formale è stato posto il parametro attuale<sup>1</sup>. Possiamo calcolare una semplice espressione con questo modello di calcolo (usiamo la freccia → per indicare un passo di riduzione):

```
fatt 3 → (fn n => if n=0 then 1 else n*fatt(n-1)) 3
      → if 3=0 then 1 else 3*fatt(3-1)
      → 3*fatt(3-1)
      → 3*fatt(2)
      → 3*((fn n => if n=0 then 1 else n*fatt(n-1)) 2)
      → 3*(if 2=0 then 1 else 2*fatt(2-1))
      → 3*(2*fatt(2-1))
      → 3*(2*fatt(1))
      → 3*(2*((fn n => if n=0 then 1 else n*fatt(n-1)) 1))
      → 3*(2*(if 1=0 then 1 else 1*fatt(1-1)))
      → 3*(2*(1*fatt(0)))
      → 3*(2*(1*((fn n => if n=0 then 1 else n*fatt(n-1)) 1)))
      → 3*(2*(1*(if 0=0 then 1 else n*fatt(n-1))))
      → 3*(2*(1*1))
      → 6
```

Si osservi come, fatta eccezione per i calcoli aritmetici e l'espressione condizionale, tutto il resto della computazione proceda per manipolazione simbolica di stringhe: niente variabili, niente aggiornamenti di valori sulla pila. La Figura 13.1 riporta un ulteriore esempio di pura manipolazione simbolica: un modo complicato per scrivere la funzione identità! (Nello studio della figura si ricordi che `fun` è solo un'abbreviazione per `val...fn`).

Infine, il lettore non avrà difficoltà a convincersi che, data la definizione

```
fun r x = r(r(x));
```

ogni computazione che coinvolga una valutazione di `r` si risolve in una riscrittura infinita: diciamo in tal caso che la computazione *diverge* e che il risultato è indefinito.

### 13.1.3 Gli ingredienti fondamentali

In queste prime sezioni abbiamo introdotto tutti gli ingredienti fondamentali del paradigma funzionale puro. Possiamo precisare e riassumere i concetti principali

<sup>1</sup> Il lettore avrà certamente riconosciuto in questa descrizione la "regola di copia" che abbiamo enunciato quale semantica del passaggio dei parametri per nome. Per il momento rimaniamo vaghi sulla semantica esatta da assegnare a questo procedimento: dedicheremo ad esso tutto il prossimo Paragrafo 13.2.

```

fun K x y = x;
fun S p q r = p r (q r);
val a = ...;

S K K a → (fn p => (fn q => (fn r => p r (q r)))) K K a
→ (fn q => (fn r => K r (q r))) K a
→ (fn r => K r (K r)) a
→ K a (K a)
→ (fn x => (fn y => x)) a (K a)
→ (fn y => a) (K a)
→ a

```

**Figura 13.1** Alcune definizioni e una computazione per riscrittura.

nel modo seguente.

Da un punto di vista sintattico il linguaggio non ha comandi (non essendovi stato da modificare per effetto collaterale), ma solo espressioni. Oltre ad eventuali valori ed operatori primitivi per i dati (quali interi, booleani, caratteri ecc.) e all'espressione condizionale, i due costrutti principali che permettono di definire espressioni sono

- l'*astrazione*, che data un'espressione qualsiasi *exp* ed un identificatore *x*, permette di costruire una nuova espressione *fn x => exp*, che denota la funzione che trasforma il parametro formale *x* in *exp* (l'espressione *exp* viene "astratta" dal valore specifico legato a *x*);
- l'*applicazione* di un'espressione *f\_exp* ad un'altra espressione *a\_exp*, che scriviamo *(f\_exp a\_exp)* (o anche senza parentesi), che denota l'applicazione della funzione (denotata da *f\_exp*) all'argomento (denotato da *a\_exp*).

Non vi sono vincoli sulla possibilità di passare funzioni come argomento ad altre funzioni o di restituire funzioni come risultato di altre funzioni (ordine superiore). Come conseguenza, vi è perfetta *omogeneità* tra programmi e dati.

Da un punto di vista semantico, un programma consiste in una serie di definizioni di valore, ciascuna delle quali inserisce una nuova associazione nell'ambiente e può richiedere la valutazione di espressioni arbitrariamente complesse. La presenza di funzioni di ordine superiore e la possibilità di definire funzioni ricorsive rende tale meccanismo di definizione molto flessibile e potente.

Ad un primo livello di comprensione la semantica della computazione (*valutazione*) non fa riferimento ad altri aspetti linguistici oltre a quelli visti sin qui e può essere definita mediante un semplice procedimento di riscrittura simbolico di stringhe (*riduzione*), che semplifica ripetutamente un'espressione fino al raggiungimento di una forma semplice, che denota immediatamente un valore. Tale procedimento procede con due operazioni principali. La prima è una semplice ricerca nell'ambiente: quando si incontra un identificatore legato in ambiente, sostituisce l'identificatore con la sua definizione. Ad esempio, nella Figura 13.1

abbiamo usato questa operazione per la prima e la quinta riduzione. Nel seguito questo passo non sarà più considerato esplicitamente: considereremo un nome come una semplice abbreviazione per il valore ad esso associato.

La seconda operazione, più interessante, tratta il caso di un'espressione funzionale applicata ad un argomento e consiste in un'opportuna versione della regola di copia (che in questo contesto viene detta  $\beta$ -regola).

**Definizione 13.1** • *Redex*: Un redex (che sta per *reducible expression*) è un'applicazione della forma  $((\text{fn } x \Rightarrow \text{corpo}) \text{ arg})$ .

- *Ridotto*: Il ridotto di un redex  $((\text{fn } x \Rightarrow \text{corpo}) \text{ arg})$  è l'espressione che si ottiene sostituendo in *corpo* ogni occorrenza (libera) del parametro formale *x* con una copia di *arg* (evitando cattura di variabili<sup>2</sup>, si ricordi pag. 254).
- $\beta$ -regola: Un'espressione *exp* nella quale compaia come sottoespressione un redex, si riduce (o si riscrive, si semplifica) in *exp1* (notazione: *exp → exp1*), dove *exp1* si ottiene da *exp* rimpiazzando il redex con il suo ridotto.

Da un punto di vista implementativo, infine, tutte le macchine astratte per linguaggi funzionali adottano estesi meccanismi di garbage collection, perché in caso di funzioni restituite come valore da altre funzioni sappiamo che gli ambienti locali devono essere preservati illimitatamente (si ricordi il Paragrafo 9.2.2).

Se questi sono i concetti cardine di un linguaggio funzionale, ad una lettura più attenta queste poche righe suscitano più problemi di quanti ne risolvano. Affronteremo un'analisi più approfondita di queste questioni semantiche nel prossimo paragrafo.

## 13.2 Valutazione

Nella succinta descrizione semantica con cui abbiamo concluso il paragrafo precedente, non abbiamo fornito dettagli su due aspetti fondamentali:

- quale sia la condizione di terminazione della riduzione (cioè cosa significhi la locuzione "una forma semplice, che denota immediatamente un valore");
- quale semantica precisa si debba dare alla  $\beta$ -regola, non tanto sull'aspetto (che già discuteremo nel contesto del passaggio dei parametri) della possibile cattura di variabile, ma soprattutto dell'ordine da seguire nella riscrittura quando in una stessa espressione siano presenti più redex.

### 13.2.1 Valori

Un *valore* è un'espressione che non dev'essere ulteriormente riscritta. In un linguaggio funzionale vi sono valori di due specie: valori di tipo primitivo e funzio-

<sup>2</sup>Nel contesto della programmazione funzionale ci si riferisce spesso agli identificatori legati a valori chiamandoli "variabili": il lettore è ormai abbastanza maturo per permetterci di mantenere quest'uso tradizionale senza che possa creare confusione circa l'assenza di variabili modificabili nei linguaggi funzionali puri.

ni. Per quanto riguarda i valori di tipo primitivo c'è poco da dire: se il linguaggio fornisce alcuni tipi primitivi (interi, booleani, caratteri ecc.) è evidente che ad ogni tipo siffatto è associato un insieme di valori primitivi, che non danno luogo a valutazione (per esempio, le costanti di tipo intero, booleano, carattere ecc.). Nell'esempio del paragrafo precedente abbiamo terminato la valutazione di *fatt* quando abbiamo raggiunto valori primitivi di tipo intero: 3, 2, 1.

Più interessanti sono i valori funzionali. Consideriamo la seguente definizione:

```
val G = fn x => ((fn y => y+1) 2);
```

Abbiamo detto che una definizione comporta la valutazione dell'espressione a destra dell'uguale ed il legame del valore così ottenuto al nome a sinistra di `=`. Ma, in questo caso, non è immediatamente chiaro quale sia il valore da associare a *G*. Si tratta di

```
fn x => 3
```

nel quale abbiamo riscritto il corpo di *G* valutando il redex che vi è contenuto, oppure si tratta di

```
fn x => ((fn y => y+1) 2)
```

nel quale non è intervenuta alcuna valutazione nel corpo di *G*? Il primo caso appare quello più rispettoso della semantica informale che abbiamo fornito alla fine del precedente paragrafo; il secondo, invece, è quello più vicino al senso di una definizione di funzione in un linguaggio convenzionale, nel quale il corpo di una funzione non viene valutato se non al momento di una chiamata a tale funzione.

Sebbene a prima vista ciò possa apparire strano, è la seconda scelta quella adottata da tutti i linguaggi funzionali di uso più comune. Non avviene alcuna valutazione "sotto" un'astrazione: ogni espressione della forma

```
fn x => exp
```

costituisce un valore e dunque eventuali redex presenti in *exp* *non* vengono mai riscritti sino a quando una tale espressione non venga applicata a qualche argomento.

### 13.2.2 Sostituzione senza cattura

Per realizzare una sostituzione senza cattura, abbiamo visto nel Capitolo 9 che si possono usare delle chiusure. Questo è in effetti il meccanismo usato anche nelle macchine astratte dei linguaggi funzionali (si veda il Paragrafo 13.4). Per la descrizione elementare che ne stiamo dando in questo momento, tuttavia, ci può bastare una convenzione sintattica: in ogni espressione non vi sono mai due parametri formali con lo stesso nome, ed i nomi delle eventuali variabili che non sono parametri formali sono tutti distinti da quelli dei parametri formali. In una parola: lo stesso nome non è mai usato per indicare due variabili distinte.

```
fun K x y = x;
fun r z = r(r(z));
fun D u = if u=0 then 1 else u;
fun succ v = v+1;

val v = K (D (succ 0)) (r 2);
```

**Figura 13.2** Un'espressione con più redex.

Con questa convenzione, nei semplici esempi che considereremo non si verificherà mai una cattura di variabile<sup>3</sup>.

### 13.2.3 Strategie di valutazione

Già nel Capitolo 8, trattando delle espressioni, abbiamo visto come ogni linguaggio debba fissare una specifica strategia (cioè un ordine fissato) di valutazione delle espressioni. La presenza di funzioni di ordine superiore rende tale questione ancora più fondamentale nel contesto dei linguaggi funzionali. Per render chiaro il problema, si considerino le definizioni della Figura 13.2. Quale valore viene associato a *v* e come viene determinato?

La  $\beta$ -regola, da sola, non è di grande aiuto, perché nella parte destra della definizione di *v* sono presenti quattro redex (dopo l'espansione dei nomi con i valori loro associati dalle definizioni):

```
K (D (succ 0))
D (succ 0)
succ 0
r 2
```

Quale di essi viene ridotto per primo? Tutti i linguaggi più diffusi usano una strategia *da sinistra a destra* (*leftmost*), che riduce i redex a partire da quello più a sinistra. Ma anche dopo aver stipulato ciò, non è chiaro quale sia il redex più a sinistra tra

```
K (D (succ 0))
D (succ 0)
succ 0
```

perché questi tre redex sono tra loro sovrapposti. Fissata dunque una valutazione da sinistra, distinguiamo ulteriormente tre strategie distinte.

<sup>3</sup>Nel caso più generale, se non si vuole tirare in causa la nozione di chiusura, per descrivere correttamente la computazione per riscrittura occorre definire in modo preciso la nozione di variabile libera e legata e quella di sostituzione, tutti concetti che tratteremo formalmente nel Paragrafo 13.6.

**Valutazione per valore** Nella strategia di valutazione per valore, detta anche in ordine applicativo (o semplicemente applicativa), o *eager*, o anche *innermost*, un redex viene valutato solo se l'espressione che costituisce la sua parte argomento è già un valore.

Più precisamente, la valutazione da sinistra in ordine applicativo procede come segue.

1. Scandisci l'espressione da valutare a partire da sinistra, selezionando la prima applicazione che incontri: sia essa  $f\_exp\ a\_exp$ .
2. Valuta per prima (applicando ricorsivamente questo stesso metodo)  $f\_exp$ , fino a ridurla ad un valore (di tipo funzionale) della forma  $(fn\ x\ => \dots)$ .
3. Valuta quindi la parte argomento  $a\_exp$  dell'applicazione, per ridurla ad un valore  $val$ .
4. Riduci infine il redex  $((fn\ x\ => \dots)\ val)$  e riparti dal punto (1).

Nel caso della Figura 13.2, il punto (1) seleziona per prima l'applicazione  $K$  ( $D\ (\text{succ}\ 0)$ ). Ora alcune applicazioni elementari dei passi (1), (2) e (3) servono per rendersi conto che  $K$ ,  $D$  e  $\text{succ}$  sono già valori (si ricordi che sottintendiamo che un nome sia un'abbreviazione per l'espressione ad esso associata). Il primo redex ad essere ridotto è quindi  $(\text{succ}\ 0)$  (cioè  $((fn\ v\ => v+1)\ 0)$ ) che viene completamente valutato in 1. Viene quindi ridotto il redex  $(D\ 1)$  (cioè  $((fn\ u\ => \text{if}\ u=0\ \text{then}\ 1\ \text{else}\ u)\ 1)$ ) che dà valore 1; quindi viene valutato  $(K\ 1)$  che produce il valore  $(\text{fun}\ y\ => 1)$ . A questo punto della valutazione l'espressione ha raggiunto la forma

$(\text{fun}\ y\ => 1)\ (r\ 2)$

Siccome la parte funzionale di questa applicazione è già un valore, la strategia prescrive che sia adesso valutato l'argomento  $(r\ 2)$ . La valutazione porta alla sua riscrittura in  $r\ (r\ 2)$ , quindi in  $r\ (r\ (r\ 2))$  e così via in una computazione divergente.

A  $v$  non sarà dunque mai associato alcun valore, perché la computazione complessivamente diverge.

**Valutazione per nome** Nella strategia di valutazione per nome, detta anche in ordine normale (o semplicemente normale), o anche *outermost*, un redex viene valutato prima della sua parte argomento.

Più precisamente, la valutazione da sinistra in ordine normale procede come segue.

1. Scandisci l'espressione da valutare a partire da sinistra, selezionando la prima applicazione che incontri: sia essa  $f\_exp\ a\_exp$ .
2. Valuta per prima (applicando ricorsivamente questo stesso metodo)  $f\_exp$ , fino a ridurla ad un valore (di tipo funzionale) della forma  $(fn\ x\ => \dots)$ .
3. Riduci il redex  $((fn\ x\ => \dots)\ a\_exp)$  usando la  $\beta$ -regola e riparti dal punto (1).

Nel caso della Figura 13.2, il primo redex ad essere ridotto è quindi

$K\ (D\ (\text{succ}\ 0))$

che viene riscritto in

$fn\ y\ => D\ (\text{succ}\ 0)$

che è un valore funzionale. L'espressione è ora nella forma

$(fn\ y\ => D\ (\text{succ}\ 0))\ (r\ 2)$

per la quale la strategia prescrive di ridurre il redex più esterno, ottenendo  $D\ (\text{succ}\ 0)$

Riducendo ora tale espressione si ottiene

$\text{if}\ (\text{succ}\ 0)=0\ \text{then}\ 1\ \text{else}\ (\text{succ}\ 0)$

Ora, procedendo da sinistra, si ottiene

$\text{if}\ 1=0\ \text{then}\ 1\ \text{else}\ (\text{succ}\ 0)$

e quindi

$\text{succ}\ 0$

dalla quale si ottiene il valore finale 1, che è quello che viene associato a  $v$ .

**Valutazione lazy** Nella valutazione per nome, uno stesso redex può dover essere valutato più volte, per effetto di qualche duplicazione che è intervenuta durante la riscrittura. Nell'esempio che stiamo discutendo, il redex  $(\text{succ}\ 0)$  è stato duplicato per effetto della funzione  $D$  ed è stato ridotto due volte<sup>4</sup> nell'espressione condizionale che costituisce il corpo di  $D$ . Ciò costituisce il prezzo da pagare per la mancata valutazione dell'argomento prima dell'applicazione di una funzione (ed è proprio ciò che consente alla valutazione per nome di ottenere un valore laddove la strategia per valore diverge), ma è molto costoso in termini di efficienza (si pensi, ovviamente, a situazioni in cui il redex duplicato richieda computazioni significative).

Per ovviare a questo inconveniente e mantenere i vantaggi della valutazione per nome, la strategia lazy procede come quella per nome, ma la prima volta che si incontra la "copia" di un redex, viene salvato il suo valore, che verrà usato quando si incontreranno le eventuali altre copie dello stesso redex.

Le strategie per nome e lazy sono esempi di strategie *call by need*, nelle quali un redex viene ridotto solo se necessario per la computazione.

### 13.2.4 Confronto tra strategie

Nell'esempio del precedente paragrafo la strategia per nome ha determinato un valore laddove la strategia per valore divergeva. È sensato porsi il problema se

<sup>4</sup>Il purista non parlerà di copie di redex, ma dirà piuttosto che il redex  $(\text{succ}\ 0)$  ha dato luogo a due *residui*, ciascuno dei quali è stato ridotto indipendentemente dall'altro.

non possa accadere che le due strategie possano determinare due valori *distinti* per la stessa espressione.

Una risposta a questa domanda è fornita dal seguente teorema, che esprime una delle caratteristiche più importanti del paradigma funzionale *puro*. Diciamo che un'espressione del linguaggio è *chiusa* se tutte le sue variabili sono legate da qualche  $f_n$ ; ricordiamo poi, dal Paragrafo 13.2.1, che per *valore primitivo* intendiamo un valore di tipo primitivo (interi, booleani, caratteri ecc.), escludendo dunque valori funzionali.

**Teorema 13.2** *Sia  $\exp$  un'espressione chiusa. Se  $\exp$  si riduce ad un valore primitivo  $\text{val}$  usando una qualsiasi delle tre strategie del Paragrafo 13.2.3, allora  $\exp$  si riduce a  $\text{val}$  seguendo la strategia per nome. Se  $\exp$  diverge usando la strategia per nome, allora diverge anche con le altre due strategie.*

Osserviamo che il teorema esclude che possa presentarsi il caso che un'espressione (chiusa) possa dare un valore primitivo  $\text{val}$  in una strategia e dare un altro valore primitivo  $\text{val}_2$  secondo un'altra strategia<sup>5</sup>. Due strategie possono dunque distinguersi tra loro solo per il fatto che una determina un valore mentre l'altra diverge, come abbiamo visto nel nostro esempio. Non possiamo scendere qui in dettagli su come il teorema possa essere dimostrato, ma è importante sottolineare che un tassello fondamentale per la sua validità è la seguente proprietà fondamentale:

Fissata una strategia qualsivoglia, nello scope dello stesso ambiente la valutazione di tutte le occorrenze di una stessa espressione produce sempre lo stesso valore.

Questa proprietà, che ovviamente fallisce immediatamente in presenza di effetti collaterali, è presa da alcuni autori come *criterio* per un linguaggio funzionale puro: un linguaggio è funzionale puro se soddisfa questa condizione. Si tratta di una proprietà molto importante che rende agevole *ragionare* su un programma funzionale e sulla quale torneremo nel Paragrafo 13.5. Osserviamo ancora, prima di procedere, che è proprio in virtù di questa proprietà che la valutazione lazy è corretta quanto la valutazione per nome: il valore ottenuto con la valutazione di un'espressione non cambia quando si incontra un'altra copia della stessa espressione (ovviamente senza aver nel frattempo modificato l'ambiente).

<sup>5</sup> Il teorema non vale più se si rimuove l'ipotesi che il valore sia *primitivo*. Avendo supposto di non valutare sotto un'astrazione, infatti, le due strategie possono dare valori (funzionali) distinti. A titolo d'esempio, definiamo  $I = f_n x \Rightarrow x$  e  $P = f_n x \Rightarrow (f_n y \Rightarrow y x)$ . Il termine  $P (I I)$  si riduce a  $f_n y \Rightarrow y (I I)$  con la strategia per nome, mentre con la strategia per valore si riduce a  $f_n y \Rightarrow y I$ . Questa differenza non è molto importante nei linguaggi di programmazione (dove vogliamo principalmente calcolare valori di tipo primitivo). Nel Paragrafo 13.6 definiremo un calcolo astratto nel quale la riduzione avviene anche in presenza di astrazioni: per quel calcolo vale una proprietà di *confluenza* (più forte e generale del Teorema 13.2), che discuteremo a pag. 442.

Alla luce del teorema precedente, ci si può chiedere quale interesse abbia la strategia per valore, visto che quella per nome è la più generale tra tutte quelle possibili. Inoltre motivi di efficienza sembrerebbero suggerire di adottare solo strategie call by need, visto che non si capisce perché si debbano ridurre redex inutili, con conseguente spreco di tempo di calcolo. Il punto è che il bilancio di efficienza non è così semplice: implementare una strategia call by need è in genere assai più costoso di una semplice strategia call by value. La strategia per valore ha efficienti implementazioni su architetture convenzionali, anche se talvolta compie del lavoro inutile (tutte le volte che viene valutato un argomento che non è poi necessario nel corpo della funzione, come accade per il secondo argomento della funzione  $K$  della Figura 13.2); quest'ultimo caso, tuttavia, può essere trattato in modo efficiente con tecniche di valutazione astratta che cerchino di identificare gli argomenti inutili.

Tra i linguaggi funzionali che abbiamo citato in apertura del capitolo, LISP, Scheme e ML adottano una strategia per valore (anche perché includono importanti aspetti imperativi), mentre Miranda e Haskell (che sono funzionali puri) adottano una valutazione lazy.

### 13.3 Programmare in un linguaggio funzionale

I meccanismi che abbiamo descritto nelle sezioni precedenti sono sufficienti ad esprimere programmi per tutte le funzioni calcolabili (costituiscono infatti un linguaggio Turing completo). Si tratta del nucleo fondamentale di ogni linguaggio funzionale, che, tuttavia, è troppo austero per poter esser usato come vero linguaggio di programmazione. Tutti i linguaggi funzionali, pertanto, immergono tale nucleo in un contesto assai più ampio, che fornisce meccanismi di varia natura, tutti tesi a rendere la programmazione più semplice ed espressiva. Passiamo in rassegna alcuni di tali meccanismi.

#### 13.3.1 Ambiente locale

Il meccanismo delle definizioni globali in ambiente che abbiamo usato sin qui è troppo poco strutturato per un linguaggio moderno. Come nei linguaggi convenzionali, è opportuno fornire meccanismi esplicativi per introdurre definizioni con scope limitato, come per esempio

```
let x = exp in exp1 end
```

che introduce il legame tra  $x$  ed il valore di  $\exp$  in uno scope che include solo  $\exp1$ .

A dire il vero la presenza di espressioni funzionali già introduce scope annidati e relativi ambienti, costituiti dai parametri formali di una funzione legati ai parametri formali. Da un punto di vista della valutazione, possiamo infatti considerare un costrutto

```
let x = exp in exp1 end
```

come zucchero sintattico per  
`(fn x => exp1) exp.`

L'uso degli scope locali è così importante, tuttavia, da giustificare l'introduzione di specifica sintassi per esso.

### 13.3.2 Interattività

Tutti i linguaggi funzionali forniscono un ambiente interattivo. L'uso del linguaggio consiste nell'immissione di un'espressione, che la macchina astratta valuta e di cui restituisce il valore. Le definizioni sono particolari espressioni che modificano l'ambiente globale (e restituiscono o meno un valore a seconda del linguaggio). Sono ovviamente forniti meccanismi per l'importazione di definizioni da file testuali e per l'esportazione dell'ambiente corrente.

Questo modello suggerisce immediatamente implementazioni di tipo interpretativo, anche se esistono implementazioni efficienti di tipo compilativo, che generano il codice compilato la prima volta che una definizione è immessa nell'ambiente.

### 13.3.3 Tipi

Come nei linguaggi convenzionali, anche nei linguaggi funzionali il sistema di tipi costituisce un aspetto di primaria importanza. Tutti i linguaggi funzionali che abbiamo citato forniscono gli usuali tipi primitivi (interi, booleani, caratteri) con le ordinarie operazioni sui loro valori. Con l'eccezione di Scheme, che è un linguaggio con controllo dinamico dei tipi, tutti gli altri hanno elaborati sistemi di tipi statici. Tali sistemi di tipi permettono la definizione di nuovi tipi quali le coppie, le liste, i "record" (cioè ennuple di valori etichettati). Ad esempio, in ML potremmo definire una funzione `somma_c` che prende come argomento una coppia di interi e restituisce la loro somma:

```
fun somma_c (n1,n2) = n1+n2;
```

Si osservi la differenza fondamentale tra `somma_c` e `somma`, che abbiamo definito a pag. 418: `somma_c` richiede una coppia di interi e non avrebbe senso fornirle solo il valore per `n1` e non quello per `n2`; `somma`, invece, è una funzione che prende un numero e restituisce una funzione che prende un altro numero e restituisce la somma dei due<sup>6</sup>.

Una parte importante dei sistemi di tipi dei linguaggi funzionali è quella dedicata ai tipi delle funzioni, visto che le funzioni sono valori denotabili ed espribibili (e spesso anche memorizzabili, se il linguaggio ammette degli aspetti imperativi). Nei linguaggi funzionali tipizzati ciò vuole anche dire che alcune espres-

<sup>6</sup> Il passaggio da una funzione che richiede come argomento un coppia (o più in generale una ennupla) ad una funzione unaria di ordine superiore, viene indicato come *curryfication* di una funzione, dal nome di Haskell Curry, uno dei padri fondatori della teoria del  $\lambda$ -calcolo.

sioni funzionali sono illegali perché non tipizzabili. Ad esempio, in un linguaggio tipizzato la seguente funzione di ordine superiore è illegale

```
fun F f n = if n=0 then f(1) else f("pippo");
```

perché il parametro formale `f` dovrebbe avere contemporaneamente tipo `int -> 'a` e tipo `string -> 'a` (con '`a`' indichiamo una variabile di tipo, cioè un tipo generico non ancora istanziato).

Un'altra espressione illegale perché viola i tipi è quella costituita da un'autoapplicazione

```
fun Delta x = x x;
```

L'espressione `(x x)` è illegale perché non c'è alcun modo per assegnare un unico tipo consistente a `x`. Visto che occorre a sinistra in un'applicazione, deve avere un tipo della forma `'a -> 'b`. Siccome poi compare come argomento di una funzione (a destra dell'applicazione) deve avere il tipo che la funzione richiede: dunque `x` deve avere tipo `'a`. Mettendo insieme i due vincoli si ha che `x` deve avere contemporaneamente il tipo `'a` ed il tipo `'a->'b` e non c'è modo per "unificare" queste due espressioni.

In linguaggi senza un sistema di tipi forte, come Scheme, la funzione `Delta` è invece perfettamente legale. In Scheme possiamo anche applicare `Delta` a se stessa: l'espressione (ovviamente scritta nell'opportuna sintassi Scheme)

```
(Delta Delta)
```

costituisce un semplice esempio di programma divergente. La mancanza di un controllo statico dei tipi rende possibile scrivere in Scheme anche espressioni quali

```
(4 3)
```

nella quale si vorrebbe applicare l'intero 4 all'intero 3. Siccome la parte sinistra di questa applicazione non è una funzione, la macchina astratta genererà errore a tempo d'esecuzione.

Nel caso di ML l'aspetto più interessante del suo sistema di tipi è il suo supporto per il polimorfismo, per il quale rimandiamo alla discussione del Paragrafo 10.7 (in particolare il riquadro di pag. 325).

### 13.3.4 Pattern matching

Uno degli aspetti più noiosi della programmazione di funzioni ricorsive consiste nella gestione dei casi terminali per mezzo di "if" espliciti. Prendiamo ad esempio la funzione (inefficiente) che restituisce l' $n$ -esimo termine della successione di Fibonacci

```
fun Fibo n = if n=0 then 1
             else if n=1 then 1
                   else Fibo(n-1)+Fibo(n-2);
```

Il meccanismo di *pattern matching*, presente in alcuni linguaggi tra cui ML e Haskell, permette di darne una definizione (in tutto e per tutto equivalente) come segue:

```
fun Fibo 0 = 1
| Fibo 1 = 1
| Fibo n = Fibo(n-1)+Fibo(n-2);
```

Il carattere | si legge "oppure": ogni ramo della definizione corrisponde ad un diverso caso della funzione. La parte più interessante di questa definizione è costituita dei parametri formali: questi non sono più ristretti ad essere identificatori, ma possono essere *pattern*, cioè espressioni formate a partire da variabili (l'ultimo caso dell'esempio), costanti (negli altri due casi) ed altri costrutti che dipendono dal sistema di tipi del linguaggio (vedremo un esempio sulle liste tra poco). Un pattern svolge il ruolo di uno schema, di un modello sul quale confrontare il parametro attuale. Quando la funzione viene applicata ad un parametro attuale, questo viene confrontato con i pattern (secondo l'ordine in cui essi compaiono nel programma) e viene selezionato il corpo corrispondente al primo pattern che si accorda (in inglese, *to match*) col parametro attuale.

Il meccanismo di pattern matching è particolarmente flessibile quando viene usato su tipi strutturati, come per esempio le liste. In sintassi ML, una lista viene indicata tra parentesi quadre, con gli elementi separati da virgole. Ad esempio,

```
["uno", "due", "tre"]
```

è una lista di tre stringhe. L'operatore :: indica il "cons" (cioè l'operatore di inserimento di un elemento in testa ad una lista):

```
"zero)::["uno", "due", "tre"]
```

è un'espressione il cui valore è la lista

```
["zero", "uno", "due", "tre"].
```

Infine, nil è la lista vuota:

```
"quattro":nil
```

è un'espressione con valore ["quattro"]. Usando il pattern matching, possiamo definire la funzione che calcola la lunghezza di una generica lista come

```
fun lung nil = 0
| lung e::resto = 1 + lung(resto);
```

Si osservi che i nomi usati nei pattern sono usati come parametri formali per indicare parti del parametro attuale. Dovrebbe essere chiaro il vantaggio in termini di concisione e chiarezza rispetto all'usuale definizione

```
fun lung lista = if lista = nil then 0
                  else 1 + lung(tl lista);
```

la quale, oltretutto, necessita dell'introduzione di una funzione di *selezione* (che abbiamo indicato con tl) per ottenere la parte di una lista privata del suo primo elemento. L'operazione di selezione è invece implicita con l'uso del pattern matching.

Un vincolo importante è che una variabile non può comparire due volte nello stesso pattern. Ad esempio, la seguente definizione di una funzione che, applicata ad una lista, restituisce true se e solo se la lista ha i primi due elementi uguali, è sintatticamente illegale

```
fun Eq nil = false
| Eq [e] = false
| Eq x::x::resto = true
| Eq x::y::resto = false;
```

perché il terzo pattern dall'alto contiene due volte la variabile x, con lo scopo di verificare l'uguaglianza dei primi due elementi. Il meccanismo di pattern matching è un modo per controllare che la *forma* di un parametro attuale si accordi con un pattern, non che i valori in esso contenuti abbiano tra loro determinate relazioni. Detto altrimenti, il pattern matching non è un'*unificazione*, un meccanismo assai più generale (e di implementazione più complessa) che discuteremo nel contesto del paradigma logico (Capitolo 14)<sup>7</sup>.

### 13.3.5 Oggetti Infiniti

In presenza di strategie per nome o lazy, è possibile definire e manipolare *stream*, cioè strutture dati (potenzialmente) infinite. In questo paragrafo faremo un piccolo esempio di come ciò possa accadere. Non possiamo condurre l'esempio in ML, perché ML adotta una strategia per valore; l'esempio è possibile, invece, in Haskell. Per non gravare il lettore con sintassi diversa, scriveremo i termini con la stessa sintassi concreta di ML, ma con l'importante stipula che la valutazione deve essere intesa in modo lazy.

Dobbiamo in primo luogo chiarire la nozione di valore per strutture dati quali le liste. In un linguaggio con strategia eager, un valore di tipo T list è una lista i cui elementi sono *valori* di tipo T. In un linguaggio lazy non si tratterebbe di una buona nozione di valore, perché potrebbe richiedere la valutazione di redex inutili, contrariamente alla filosofia call by need. Per vederne il motivo, definiamo per prima cosa le funzioni

```
fun hd x::resto = x;
fun tl x::resto = resto;
```

che restituiscono rispettivamente il primo elemento (cioè la testa, head) ed il resto (cioè la coda, tail) di una lista non vuota (nel caso di lista vuota la macchina astratta genererà errore sul pattern matching). Consideriamo ora l'espressione

```
hd [2, ((fn n=>n+1) 2)].
```

Per calcolare il suo valore (cioè 2) non è necessario ridurre il redex che figura nel secondo elemento della lista, perché non sarà mai usato nel corpo della funzione hd. Per questi motivi, in un contesto lazy un valore di tipo lista è una qualsiasi espressione della forma

```
exp1 :: exp2
```

<sup>7</sup>L'implementazione del pattern matching non è altro che la sua traduzione "ovvia" in una sequenza di "if". Nel caso di Eq, il terzo pattern richiederebbe di controllare l'uguaglianza di due elementi di tipo arbitrario. Se si tratta di valori per i quali il linguaggio *non* definisce l'uguaglianza (come accade in genere, ad esempio, per i valori funzionali), non sarebbe possibile implementare il controllo di uguaglianza.

dove `exp1` ed `exp2` possono anche contenere redex.

Risulta allora lecito definire una lista per ricorsione, come per esempio

```
val infiniti2 = 2 :: infiniti2;
```

L'espressione `infiniti2` corrisponde ad una lista potenzialmente infinita i cui elementi sono tutti 2 (ed in effetti divergerebbe verso tale lista in una valutazione eager).

Tale valore è perfettamente manipolabile, in valutazione lazy. Per esempio

```
hd infiniti2
```

è un'espressione la cui valutazione termina con valore 2, così come

```
hd (tl (tl (tl infiniti2))).
```

Come ulteriore esempio di stream, la seguente funzione costruisce la lista infinita dei naturali a partire dal suo argomento `n`:

```
fun numeriDan n = n :: numeriDan (n+1);
```

Definita una funzione di ordine superiore che applica il suo argomento funzionale a tutti gli elementi di una lista

```
fun map f nil = nil
| map f e::resto = f(e)::resto;
```

possiamo, per esempio, ottenere la lista infinita di tutti i quadrati a partire da `n*n`:

```
fun quadratiDan2 n = map (fn y => y*y) (numeriDan n);
```

### 13.3.6 Aspetti imperativi

Molti linguaggi funzionali forniscono anche meccanismi imperativi, che introducono una nozione di stato modificato per effetto collaterale.

Per quanto riguarda ML, il linguaggio mette a disposizione delle vere e proprie variabili modificabili (dette "reference cell"), con i loro tipi. Per ogni tipo `T` del linguaggio (anche tipi funzionali, dunque), è definito il tipo `T ref`, i cui valori sono variabili modificabili che possono contenere valori di tipo `T`. Una variabile modificabile di tipo `T`, inizializzata al valore `v` (di tipo `T`) è creata col costrutto

```
ref v
```

I costrutti usuali per associare nomi a valori possono essere usati anche per valori di tipo `T ref`; ad esempio

```
val I = ref 4;
```

crea una reference cell di tipo `int ref`, inizializzata a 4 e col nome `I`: si tratta di quello che in linguaggio imperativo chiameremmo "la variabile modificabile (di nome) `I`". Ovviamente `I` è il nome della reference cell e non del valore in esso contenuto (è un l-valore). Per ottenere il suo r-valore occorre dereferenziarlo esplicitamente, con l'operatore `!`

```
val n = !I + 1;
```

### Effetti collaterali in LISP

LISP, il primo linguaggio funzionale, è basato su una struttura dati chiamata *coppia puntata* (o cella cons, si ricordi il riquadro di pag. 199), composta da due parti, *car* (per *contents of the address register*) e *cdr* (per *contents of the decrement register*). Un'espressione della forma

```
(cons a b)
```

alloca una coppia puntata e la inizializza in modo che il suo *car* punti al valore di `a` e il suo *cdr* punti al valore di `b`. Le due componenti di una coppia puntata possono essere selezionate con le funzioni *car* e *cdr*:

```
(cdr (cons a b))
```

ha valore `b`. Queste sono caratteristiche che appartengono alla parte pura di LISP.

LISP ha tuttavia anche meccanismi imperativi, per esempio funzioni quali `(rplaca x a)` e `(rplacd x a)` che, rispettivamente, assegnano al *car* (al *cdr*) della coppia puntata `x` il valore di `a`:

```
(cdr (rplacd (cons 'a 'b) 'c))
```

ha valore '`c`'. Altre funzioni con effetti collaterali sono *set* e *setq*.

L'espressione `!I` ha tipo `int`; in generale, se `v` è un'espressione di tipo `T ref`, `!v` ha tipo `T`. La linea qui sopra associa al nome `n` il valore 5. A rischio di apparire pedanti, osserviamo che `n` non è una variabile modificabile: è un nome ordinario al quale, in ambiente, è stato associato un valore.

Le reference cell possono essere modificate con un assegnamento:

```
I := !I + 1;
```

Si osservi la differenza con la definizione del valore `n` di poco fa: qui stiamo modificando (per effetto collaterale) lo r-valore di una reference cell già esistente. Il tipo di un costrutto imperativo di questa forma è `unit` (si ricordi il Paragrafo 10.3.7).

La presenza di variabili modificabili introduce, ovviamente, la possibilità di aliasing. Consideriamo infatti le definizioni

```
val I = ref 4;
val J = I;
```

La seconda linea associa il valore del nome `I` (cioè la reference cell: lo l-valore) al nome `J`. Siamo in presenza di una classica situazione di aliasing: `I` e `J` sono nomi diversi per lo stesso l-valore. Dopo il frammento

```
I := 5;
val z = !J;
```

al nome `z` è associato il valore 5.

La definizione del linguaggio non specifica come implementare un assegnamento tra variabili modificabili. Dagli esempi fatti sin qui sugli interi, si potrebbe

immaginare un'implementazione tradizionale con copia del r-valore dalla sorgente alla destinazione dell'assegnamento. In una stessa reference cell del tipo opportuno, tuttavia, possono essere memorizzati valori con occupazione di memoria assai diversa. Consideriamo ad esempio variabili modificabili che contengono liste e stringhe:

```
val S = ref "pera";
val L = ref ["uno", "due", "tre"];
```

Il nome `S` è di tipo `string ref`, mentre `L` è di tipo `(string list) ref`. Le due variabili modificabili associate ai nomi `S` e `L` possono contenere, rispettivamente, una *qualsiasi* stringa e una *qualsiasi* lista. Ad esempio possiamo effettuare degli assegnamenti:

```
S := "questa_è_una_stringa_molto_più_lunga_della_precedente";
L := "zero" :: "quattro" :: "cinque" :: !L;
```

I nuovi valori di `L` e di `S` hanno bisogno per essere rappresentati di più memoria di quelli precedenti: non è possibile implementare questi assegnamenti con una semplice (e tradizionale) copia dei valori. La macchina astratta copierà in questo caso dei riferimenti (insomma, dei puntatori) a tali valori. Tutto questo, tuttavia, è del tutto invisibile all'utente del linguaggio: l'implementazione gestisce le due diverse esigenze di spazio allocando la memoria necessaria per ogni caso e quindi modificando i riferimenti.

Insieme alle variabili modificabili, ML fornisce anche costrutti imperativi di controllo, quali il comando di sequenzializzazione (`;`) e i cicli.

È opportuno osservare che, in presenza di aspetti imperativi, sia il Teorema 13.2 che la proprietà che abbiamo citato immediatamente dopo di esso *falsiscono*. Ciò è particolarmente rilevante nel caso di una strategia di valutazione lazy, che risulterebbe scorretta in presenza di effetti collaterali. È per questo motivo che linguaggi quali Miranda e Haskell, che adottano appunto una strategia lazy, non ammettono effetti collaterali di alcun tipo (sono funzionali puri).

### 13.4 Implementazione: la macchina SECD

Le tecnologie per l'implementazione dei linguaggi funzionali sono oggi notevolmente sofisticate e non possono essere trattate in questo testo. Le tecniche di passaggio dei parametri funzionali che abbiamo descritto nel Capitolo 9 costituiscono il cuore di tali implementazioni. Per chiarire in dettaglio come si possano gestire le funzioni di ordine superiore, ci limiteremo a presentare in modo succinto la macchina SECD, un prototipo per la valutazione *per valore* proposto nel 1964 da Peter Landin, dal quale discendono molte delle macchine astratte esistenti.

Descriveremo la macchina SECD in modo molto astratto, con riferimento ad un linguaggio funzionale puro semplicissimo, costituito soltanto da astrazione e applicazione. Gli elementi primitivi del linguaggio sono nomi di costanti e variabili e alcune funzioni primitive. Non specifichiamo con esattezza quali siano le costanti e le funzioni primitive (per esempio, potremmo avere le costanti 0, 1,

`true` ecc. e le funzioni primitive `succ` e `pred`): la parte rilevante della macchina non riguarda tali aspetti del linguaggio, ma la presenza di funzioni di ordine superiore. Supporremo per semplicità di aver a che fare solo con funzioni unarie. Detto questo, supponiamo che `Var` sia un non terminale dal quale si possano derivare una quantità numerabile di nomi (di variabile) e che `Const` e `Fun` siano, rispettivamente, i non terminali dai quali si derivano le opportune costanti e funzioni primitive. Possiamo dare la grammatica del nostro linguaggio d'esempio come

$$\exp ::= \text{Const} \mid \text{Var} \mid \text{Fun} \mid (\exp \ \exp) \mid (\text{fn } \text{Var } \Rightarrow \ \exp).$$

La macchina SECD ha quattro componenti principali:

- una pila (`Stack`), che memorizza i risultati parziali che saranno utilizzati nel seguito della computazione;
- un ambiente (`Environment`), cioè una lista di associazioni tra nomi e valori. Abbiamo già osservato che “non si ha valutazione sotto un'astrazione”: il valore di un'astrazione è una *chiusura*, cioè una coppia costituita dall'astrazione e dall'ambiente nel quale valutare le variabili libere del suo corpo. Per semplificare la notazione, supporremo che una chiusura sia una *tripla* composta da un ambiente, un'espressione (che risulterà sempre dal corpo di un'astrazione) e da una variabile (che rappresenta la variabile che era legata nell'astrazione); scriveremo pertanto una chiusura come `cl(E, exp, x)`, che rappresenterà il valore, nell'ambiente `E`, dell'astrazione (`fn x => exp`);
- un `Controllo`, rappresentato da una pila di espressioni da valutare; tra le espressioni figura anche l'operatore speciale `@`, che si legge “app” e che indica che è possibile eseguire un'applicazione (perché le due espressioni che compongono un'applicazione sono già state valutate e si trovano nelle posizioni seguenti della pila);
- un'area di memorizzazione (`Dump`), cioè una pila nella quale sono stati salvati alcuni stati precedenti della macchina, quando la computazione è stata sospesa per valutare dei redex interni.

Indicheremo con  $[a_1, \dots, a_n]$  la pila composta dagli elementi  $a_1, \dots, a_n$ , con  $a_1$  sulla cima;  $[]$  è la pila vuota. Se  $P$  è una pila,  $\text{top}(P)$  indica l'elemento sulla cima, mentre  $\text{tl}(P)$  indica  $P$  privata dell'elemento sulla cima.

Uno stato della macchina è costituito da una quadrupla  $(S, E, C, D)$ ; un dump è sempre composto da una struttura della forma

$$(S_1, E_1, C_1, (S_2, E_2, C_2, (S_3, E_3, C_3, []))).$$

In questo caso è stato salvato per prima cosa lo stato  $(S_3, E_3, C_3, [])$ , poi è stato salvato lo stato  $(S_2, E_2, C_2, (S_3, E_3, C_3, []))$  che ingloba il dump precedente, e così via.

Per valutare l'espressione `exp` la macchina inizia il suo funzionamento nello stato  $([], [], [exp], [])$  e termina la propria esecuzione quando sia il controllo `C` sia il dump `D` sono vuoti.

Il funzionamento della SECD è descritto da una funzione di transizione che permette di passare da uno stato ad uno successivo secondo le regole che seguono; la macchina sceglie la regola a seconda dell'espressione che vede sul controllo (si immagini il controllo descritto con la sua sintassi astratta). Supponiamo dunque che la macchina si trovi nel generico stato  $(S, E, C, D)$ :

1. sul controllo c'è una *costante*  $c$  (come 1, o *true*), cioè un'espressione che non ha bisogno di esser valutata: in tal caso il nuovo stato della macchina è  $(c :: S, E, tl(C), D)$ , nel quale il valore (immediato) della costante è posto sullo stack ed il controllo è privato dell'espressione che si è appena valutata.
2. sul controllo c'è una *variabile*  $x$ : in tal caso il valore è quello che l'ambiente associa ad  $x$ ; dunque il nuovo stato della macchina è  $(E(x) :: S, E, tl(C), D)$ ;
3. sul controllo c'è un'*applicazione* della forma  $(exp_f \ exp_a)$ : in tal caso, siccome la SECD implementa una valutazione eager, si deve seguire la procedura di valutazione che abbiamo descritto nel Paragrafo 13.2.3. Il nuovo stato è dunque  $(S, E, exp_f :: exp_a :: @ :: tl(C), D)$ , che esprime il fatto che la prossima espressione da valutare è la parte funzionale dell'applicazione (che avrà come valore una chiusura, come vedremo) seguita dalla parte argomento dell'applicazione, seguita infine dall'espressione speciale  $@$  che forzerà l'applicazione vera e propria della funzione all'argomento;
4. sul controllo c'è un'*astrazione*  $(fn \ x \ => \ exp)$ : sappiamo che siamo in presenza già di un valore, che deve però essere "chiuso" con l'ambiente corrente. Il nuovo stato della macchina è dunque  $(cl(E, exp, x) :: S, E, tl(C), D)$ ;
5. sul controllo c'è  $@$ : sappiamo allora per costruzione che sullo stack ci sono due valori che rappresentano, rispettivamente, un argomento (sulla cima della pila) ed una funzione da applicare (il secondo elemento). Ci sono due sottocasi, che dipendono dalla funzione che si deve applicare.
  - La funzione è una *funzione primitiva*  $f$  (come *succ*): si deve allora direttamente applicare la funzione all'argomento ed il nuovo stato della macchina è  $(f(top(S)) :: tl(tl(S)), E, tl(C), D)$ ;
  - La funzione è una *chiusura*  $cl(E_1, exp, x)$ : è questo uno dei punti centrali della SECD, che chiama in gioco il dump. Occorre usare la regola di copia: la computazione passa ad  $exp$  nell'ambiente della chiusura  $E_1$  (e non nell'ambiente corrente  $E$ ) modificato col legame del parametro formale. Ma non si deve dimenticare la computazione eseguita sin'ora, che viene congelata nel dump. Il nuovo stato della macchina è dunque  $([], E_1[x \leftarrow top(S)], [exp], (tl(tl(S)), E, tl(C), D))$ .
6. il controllo è *vuoto*: ciò può accadere quando la computazione è terminata (nel qual caso anche il dump è vuoto), ma anche quando si è terminata la computazione iniziata con la regola precedente (quella della chiusura applicata ad un valore). In questo secondo caso (cioè dump non vuoto della forma  $(S_1, E_1, C_1, D_1)$ ) si deve ripristinare la computazione salvata sul dump, "restituendo" nel contempo il valore appena calcolato (che si trova sulla cima dello stack). Il nuovo stato della macchina è  $(top(S) :: S_1, E_1, C_1, D_1)$ .

S	E	C	D	reg
[]	[]	[(F 3)]		
[ch <sub>1</sub> ]	[]	[F, 3, @]		3
	[]	[3, @]		4
		dove $ch_1 = cl([], (sqrt ((fn \ y \ => (succ x)) x)), x)$		
[3, ch <sub>1</sub> ]	[]	@]		
		((sqrt ((fn \ y \ => (succ x)) x)))	d <sub>1</sub>	1
		dove $d_1 = ([], [], [], []])$		5
		[x ← 3]		
		sqrt, ((fn \ y \ => (succ x)) x), @]	d <sub>1</sub>	3
		[x ← 3]		
		((fn \ y \ => (succ x)) x), @]	d <sub>1</sub>	1
		[x ← 3]		
		(fn \ y \ => (succ x)), x, @, @]	d <sub>1</sub>	1
		[x ← 3]		
		[x, @, @]	d <sub>1</sub>	4
		dove $ch_2 = cl([x \leftarrow 3], (succ x), y)$		
		[x ← 3]		
		[@, @]	d <sub>1</sub>	2
		[x ← 3, y ← 3]		
		(succ x)	d <sub>2</sub>	5
		dove $d_2 = ([sqrt], [x \leftarrow 3], [@], d_1)$		
		[x ← 3, y ← 3]		
		succ, x, @]	d <sub>2</sub>	3
		[x ← 3, y ← 3]		
		[x, @]	d <sub>2</sub>	1
		[x ← 3, y ← 3]		
		[@]	d <sub>2</sub>	2
		[x ← 3, y ← 3]		
		[]	d <sub>2</sub>	5
		[4, sqrt]		
		[x ← 3]		
		[@]	d <sub>1</sub>	6
		[x ← 3]		
		[]	d <sub>1</sub>	5
		[2]		
		[]	d <sub>1</sub>	6

Figura 13.3 Una computazione della macchina SECD.

La Figura 13.3 esemplifica come la macchina SECD calcola il valore dell'espressione  $(F \ 3)$ , dove

$$F = fn \ x \ => (sqrt ((fn \ y \ => (succ x)) x)).$$

Il calcolo è disposto su una tabella, nella quale ogni riga rappresenta una transizione della macchina; la regola usata per la transizione è indicata nella colonna più a destra.

## 13.5 Il paradigma funzionale a confronto

Giunti al termine della nostra presentazione del paradigma funzionale è opportuno chiedersi in cosa risieda l'interesse di questo paradigma e dei relativi linguaggi. Una tale domanda, in realtà, deve essere posta su almeno due piani distinti:

- quello della pratica di programmazione;
- quello del progetto dei linguaggi di programmazione.

Cercheremo di dare una risposta su questi due livelli, ma è opportuno divagare un istante per chiarire il contesto nel quale si possa parlare correttamente di "pratica di programmazione".

**Correttezza dei programmi** L'informatico principiante ritiene spesso che il progetto e la redazione di un programma efficiente costituiscano le operazioni più importanti nelle quali si sostanzia il proprio mestiere. L'ingegneria del software, tuttavia, ha largamente dimostrato, sia in teoria che con ampiissimi studi sperimentali, che i fattori più critici di un progetto software sono la correttezza, la leggibilità, la manutenibilità, l'affidabilità. In termini economici, questi fattori influiscono per ben più del cinquanta per cento del costo complessivo; in termini sociali, la manutenzione del software (che dipende in modo cruciale dalla sua leggibilità) può coinvolgere centinaia di persone diverse, in un arco temporale di decine di anni; in termini etico-deontologici, dall'affidabilità e dalla correttezza di un sistema software possono dipendere la vita, o la salute, di centinaia di persone.

In particolare, è oggi ancora lontano il momento in cui l'informatico sarà in grado di produrre software con delle garanzie di correttezza paragonabili a quelle con le quali un progettista edile rilascia i propri manufatti (ponti, colonne, strutture). Per il progetto edile, infatti, l'ingegnere ha a disposizione tutto un *corpus* di matematica applicata col quale "calcola" le strutture: se un ponte deve sostenere un certo peso, è sottoposto a certi venti, può essere esposto ad eventi sismici di determinata magnitudo ecc., allora deve avere determinate dimensioni, deve esser realizzato con materiali con certe caratteristiche ecc. Tali caratteristiche non sono determinate dal gusto del progettista, ma sono calcolate usando gli opportuni strumenti matematici. Il progettista informatico è ben lontano dall'avere a disposizione una matematica anche solo lontanamente paragonabile a quella del progettista di strutture.

Una delle motivazioni per il ritardo con cui l'informatica si presenta all'appuntamento con le garanzie di correttezza è che ragionare su programmi con effetti collaterali è particolarmente difficile e costoso in termini di tempo di calcolo. Al contrario, vi sono tecniche standard, basate su opportune varianti dell'induzione, che permettono ragionamenti anche sofisticati su programmi senza effetti collaterali.

Ecco dunque una prima motivazione per lo studio dei linguaggi funzionali puri: se affidabilità, leggibilità, correttezza, sono più importanti dell'efficienza, non c'è dubbio che la programmazione funzionale genera software più leggibile, di cui è più facile dimostrare la correttezza e, dunque, più affidabile.

**Schemi di programmi** Anche l'ordine superiore, usato ampiamente nel paradigma funzionale, ha importanti vantaggi pragmatici. Un modo tipico di usare l'ordine superiore è quello di sfruttarlo per definire degli *schemi* (diremmo dei *paradigmi*, se non rischiassimo di fare confusione terminologica) di programmi generali, da cui ottenere programmi specifici per istanziazione. Consideriamo ad esempio un semplice programma per la somma degli elementi di una lista di interi, quale possiamo scrivere in ML:

```
fun sommal nil = 0
| sommal n::resto = n + sommal(resto);
```

Se ora vogliamo il prodotto di una lista, possiamo scrivere

```
fun prodl nil = 1
| prodl n::resto = n * prodl(resto);
```

È evidente come questi due programmi sono istanze di uno stesso schema di programmazione, che viene chiamato in genere *fold*:

```
fun fold f i nil = i
| fold f i n::resto = f(n, fold f i resto);
```

dove *f* è l'operazione binaria da iterare sugli elementi della lista (+ o \*), *i* è il valore da usare nel caso terminale (l'elemento neutro dell'operazione) e il terzo parametro è la lista su cui si esegue l'iterazione.

Se si trascura il fatto che + e \* hanno notazione infissa invece che prefissa, si vede subito che possiamo definire le nostre due funzioni a partire da *fold*:

```
val sommal = fold + 0;
val prodl = fold * 1;
```

Qui l'ordine superiore è usato in modo essenziale sia per passare a *fold* la funzione da iterare, sia per far sì che *fold* + 0 restituisca una funzione che necessita di un argomento di tipo lista.

L'uso estensivo di schemi di programmi aumenta la modularità del codice e permette di fattorizzare le dimostrazioni di correttezza.

**Bilancio** Gli argomenti che abbiamo portato a favore dei linguaggi funzionali non emergono solo da ambienti accademici. Probabilmente la più appassionata difesa del paradigma funzionale è quella pronunciata nel 1977 da John Backus in occasione del conferimento del premio Turing, che gli veniva assegnato per il progetto ed il compilatore di Fortran.

Anticipando in un certo senso i tempi, Backus poneva i concetti di correttezza e leggibilità al centro del processo di produzione del software, relegando l'efficienza ad un secondo piano. Identificava anche nella programmazione funzionale *pura* lo strumento con cui procedere nella direzione che indicava. Oggi abbiamo macchine estremamente più efficienti di quelle del 1977, ma non abbiamo fatto gli stessi progressi nel campo delle tecniche per la correttezza del software.

Nessuno ha però sperimentato davvero in grande il paradigma funzionale puro per ottenere una risposta sperimentale certa al confronto tra questo tipo di paradigma e quello imperativo tradizionale. Esperienze significative sono state condotte da IBM, con il linguaggio FP definito da Backus, e in centri di ricerca accademici con il linguaggio Haskell. Tutti gli altri linguaggi funzionali più diffusi hanno comunque aspetti imperativi che, sebbene utilizzati in modi e contesti molto diversi da quelli degli ordinari linguaggi imperativi, vanificano i vantaggi dei linguaggi funzionali in termini di dimostrazioni di correttezza (ma non in termini di leggibilità e di riuso degli schemi). In conclusione, dobbiamo dire che, dal punto di vista della pratica di programmazione (intesa alla luce dell'intero processo di produzione del software) non è ancora dimostrata la superiorità di un paradigma sull'altro.

È sull'altro piano, quello del progetto dei linguaggi di programmazione, che gli studi e l'esperienza sui linguaggi funzionali hanno avuto un impatto molto

importante. Molti concetti individuati e sperimentati nel contesto dei linguaggi funzionali sono poi migrati in altri paradigmi. Tra questi concetti quello più noto al lettore è quello di sistema di tipi: le nozioni di generico, di polimorfismo, di type safe, tutte originano nel contesto dei linguaggi funzionali (perché è più semplice studiarle ed implementarle in un ambiente senza effetti collaterali).

## 13.6 Fondamenti: $\lambda$ -calcolo

Abbiamo già osservato in apertura del capitolo che il paradigma funzionale è ispirato ad una diversa fondazione della teoria della calcolabilità, alternativa a quella basata sulle macchine di Turing (ma ad essa equivalente quanto a potere espressivo). Presenteremo in questo paragrafo i punti salienti di tale sistema formale, che va sotto il nome di  $\lambda$ -calcolo.

La sintassi del  $\lambda$ -calcolo è estremamente austera e può essere data in una riga mediante la seguente grammatica, dove  $X$  è un non terminale che rappresenta una generica variabile,  $M$  è il simbolo iniziale, il punto e le due parentesi sono simboli terminali:

$$M ::= X \mid (MM) \mid (\lambda X.M).$$

Inoltre supponiamo presente un insieme numerabile di simboli terminali di variabile che, per comodità, indicheremo in genere con le ultime lettere dell'alfabeto minuscolo:  $x, y, z$  ecc.

Il lettore riconoscerà in queste clausole sia applicazione che astrazione (scritta con  $\lambda$  invece che con  $\text{fn}$ ), che avevamo introdotto informalmente nel Paragrafo 13.1.3. Chiamiamo  $\lambda$ -termini (o semplicemente termini) le stringhe di terminali che si derivano con questa grammatica.

**Convenzioni sintattiche, variabili libere e legate** I libri "ufficiali" sul  $\lambda$ -calcolo introducono molte notazioni accessorie per semplificare la notazione. Per esempio, l'applicazione associa a sinistra, ossia

$$M_1 M_2 \cdots M_n \text{ sta per } (\cdots (M_1 M_2) \cdots M_n),$$

mentre la portata di un  $\lambda$  si estende a destra il più possibile (cioè  $\lambda x.x y$  sta per  $(\lambda x.(x y))$  e non per  $((\lambda x.x)y)$ ). Cercheremo di usare queste convenzioni il meno possibile, usando più parentesi. Parleremo anche di *sottotermine*, col significato ovvio (ad esempio,  $(xy)$  è un sottotermine di  $(\lambda x.(xy))$ , ma  $\lambda x$  non lo è).

L'operatore di astrazione *lega* la variabile sulla quale agisce<sup>8</sup>, nel duplice senso che, semanticamente, la ridenominazione consistente della variabile legata non modifica la semantica di un'espressione e, sintatticamente, eventuali sostituzioni non hanno effetto sulle variabili legate. Formalizziamo questi aspetti nelle definizioni seguenti. In primo luogo, definiamo, per una generica espressione  $M$ ,

<sup>8</sup>Si ricordi quanto già detto nel Paragrafo 9.1 in relazione ai parametri formali dei sottoprogrammi.

l'insieme delle sue *variabili libere*, che indichiamo con  $Fv(M)$  (dall'inglese free variables), e delle sue variabili legate,  $Bv(M)$  (bound variables):

$$\begin{array}{llll} Fv(x) & = & \{x\} & Bv(x) = \emptyset \\ Fv(MN) & = & Fv(M) \cup Fv(N) & Bv(MN) = Bv(M) \cup Bv(N) \\ Fv(\lambda x.M) & = & Fv(M) - \{x\} & Bv(\lambda x.M) = Bv(M) \cup \{x\} \end{array}$$

**Sostituzione** Possiamo ora definire in modo formale la nozione di sostituzione senza cattura di variabile che costituisce il nucleo della regola di copia. Definiamo dunque la nozione  $M[N/x]$  che leggiamo: la sostituzione di  $N$  al posto delle occorrenze libere di  $x$  in  $M$ :

$$\begin{array}{lll} x[N/x] & = & N \\ y[N/x] & = & y \\ (M_1 M_2)[N/x] & = & (M_1[N/x] M_2[N/x]) \\ (\lambda y.M)[N/x] & = & (\lambda y.M[N/x]) \\ (\lambda y.M)[N/x] & = & (\lambda y.M) \end{array} \quad \begin{array}{l} \text{qualora } x \neq y \\ \text{qualora } x \neq y \text{ e } y \notin Fv(N) \\ \text{qualora } x = y \end{array}$$

La definizione è semplice: praticamente, per sostituire  $N$  al posto di  $x$  in  $M$  "si spinge" la sostituzione fino alle foglie dell'albero sintattico, cioè le variabili, e si controlla se la foglia è etichettata con  $x$  o con un'altra variabile. Ma non è una definizione ovvia per la necessità di non far catturare le variabili presenti in  $N$  da parte di qualche lambda. Questo è il motivo per cui la penultima clausola richiede che  $y \notin Fv(N)$ . Cosa succede se, invece,  $y$  è *presente* in  $N$ ? Possiamo ridenominare la variabile legata da  $\lambda$ . Come dicevamo poc'anzi, infatti, non è il nome della variabile che conta, quando vi è applicato un  $\lambda$ , ma solo il modo con cui è usata. In altre parole, considereremo equivalenti due termini che differiscono solo per il nome delle loro variabili legate; per esempio  $\lambda x.x$  e  $\lambda y.y$ , oppure  $\lambda x.\lambda y.x$  e  $\lambda v.\lambda w.v$ .

**Alfa-equivalenza** L'idea intuitiva che due espressioni che differiscono solo per il nome di variabili libere sono equivalenti, è formalizzata dalla nozione di  $\alpha$ -equivalenza:

$$\lambda x.M \equiv_{\alpha} \lambda y.M[y/x] \quad y \text{ fresca}$$

dove con  $y$  "fresca" intendiamo che si tratta di una nuova variabile non presente in  $M$ .

Due termini che differiscono tra loro per il solo fatto che qualche sotto termine dell'uno è stato rimpiazzato con un sotto termine  $\alpha$ -equivalente, saranno considerati uguali.

**Computazione: beta riduzione** La computazione procede per riscrittura secondo la prescrizione della  $\beta$ -riduzione:

$$(\lambda x.M)N \rightarrow_{\beta} M[N/x].$$

Più in generale, diremo che  $M$   $\beta$ -riduce a  $N$  (e si scrive  $M \rightarrow N$ ) quando  $N$  è il risultato dell'applicazione di un passo di  $\beta$ -riduzione a qualche sottotermino di  $M$ . Un sottotermino della forma  $(\lambda x.M)N$  è un *redex*, il cui *ridotto* è  $M[N/x]$ . Si osservi che la nozione di  $\beta$ -riduzione è non deterministica: qualora esistano molteplici redex nello stesso termine, non è prescritta la scelta di un redex specifico.

La  $\beta$ -riduzione è una relazione non simmetrica: in generale se  $M \rightarrow N$ , non è vero che  $N \rightarrow M$ . È utile introdurre una relazione simmetrica, che chiamiamo  $\beta$ -uguaglianza, definita come la chiusura riflessiva e transitiva della  $\beta$ -riduzione e che indichiamo con  $=_\beta$ . Intuitivamente,  $M =_\beta N$  significa che  $M$  e  $N$  sono collegati da una sequenza di  $\beta$ -riduzioni (non necessariamente tutte nello stesso verso).

**Forme normali** La  $\beta$ -riduzione termina quando si raggiunge un  $\lambda$ -termine che non contiene alcun redex: tali termini sono detti *forme normali*. Ad esempio,  $\lambda x.\lambda y.x$  è una forma normale, mentre il termine  $\lambda x.(\lambda y.y)x$  non è una forma normale, perché contiene il redex  $(\lambda y.y)x$ . Si ha allora

$$\lambda x.(\lambda y.y)x \rightarrow \lambda x.x$$

e  $\lambda x.x$  è ora una forma normale.

Richiamiamo l'attenzione del lettore sul fatto che, in quest'ultimo esempio, abbiamo applicato un passo di  $\beta$ -riduzione *sotto* (dentro) un  $\lambda$ . Le forme normali sono valori nel senso del Paragrafo 13.2.1, mentre non tutti i valori sono forme normali.

Alcuni termini hanno riduzioni che terminano (su una forma normale); vi sono però anche termini che si riducono all'infinito senza mai produrre una forma normale, come per esempio

$$\begin{aligned} (\lambda x.xx)(\lambda x.xx) &\rightarrow (xx)[(\lambda x.xx)/x] \\ &= (\lambda x.xx)(\lambda x.xx) \\ &\rightarrow (\lambda x.xx)(\lambda x.xx) \\ &\rightarrow \dots \end{aligned}$$

**Confluenza** Una proprietà fondamentale del  $\lambda$ -calcolo è che il non determinismo insito nella nozione di  $\beta$ -riduzione non ha effetti dannosi. Comunque si scelgano i redex da ridurre all'interno di un termine, il risultato finale (forma normale) di un sequenza di riduzione è sempre lo stesso. Detto in modo più formale, vale la seguente proprietà, detta *confluenza* e rappresentata graficamente nella Figura 13.4:

Se  $M$  si riduce ad  $N_1$  con qualche passo di riduzione, e  $M$  si riduce anche ad  $N_2$  con qualche passo di riduzione, allora esiste un termine  $P$  tale che sia  $N_1$  che  $N_2$  si riducono a  $P$  con qualche passo di riduzione.

Una conseguenza importante di questa proprietà è che se un termine può essere ridotto ad una forma normale, questa è unica ed indipendente dall'ordine seguito per raggiungerla.

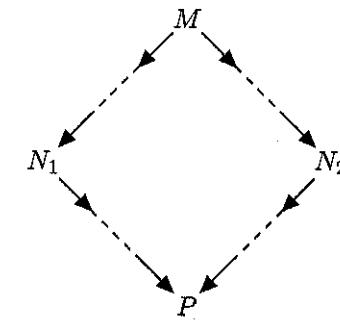


Figura 13.4 La proprietà di confluenza per i  $\lambda$ -termini.

**Operatori di punto fisso** Giunti a questo punto, abbiamo visto come il  $\lambda$ -calcolo formalizza molte delle nozioni che abbiamo visto in questo capitolo<sup>9</sup>. Non abbiamo però introdotto alcun meccanismo per definire funzioni ricorsive: non avendo il  $\lambda$ -calcolo una nozione di ambiente esterno (globale) nel quale assegnare nomi a termini, come fare per definire una funzione ricorsiva, per il quale il nome sembra essere una nozione fondamentale?

Il punto (sorprendente a prima vista!) è che le nozioni di astrazione e di ambiente locale (che discende dalla possibilità di avere  $\lambda$  annidati) sono *da sole* sufficienti per definire la ricorsione, o, più propriamente, punti fissi.

Consideriamo la seguente semplicissima definizione ricorsiva, che per semplicità scriviamo in un  $\lambda$ -calcolo esteso con interi ed espressione condizionale<sup>10</sup>

$$Z = \lambda n. \text{if } n = 0 \text{ then } 0 \text{ else } 1 + Z(n - 1). \quad (13.1)$$

La nostra esperienza di linguaggi di programmazione ci suggerisce forse di leggere questa relazione come una *definizione*:

$$Z \stackrel{\text{def}}{=} \lambda n. \text{if } n = 0 \text{ then } 0 \text{ else } 1 + Z(n - 1).$$

Ma non è l'unica lettura possibile (in particolare non è possibile nel  $\lambda$ -calcolo che non ha una nozione di ambiente globale che permetta di usare il nome di un termine all'interno di un'espressione). Il matematico probabilmente la leggerebbe come un'*equazione*, nell'incognita  $Z$  (si ricordi la discussione sulle definizioni

<sup>9</sup>Dovremmo dire meglio: formalizza molte delle nozioni che abbiamo visto in questo libro. Il  $\lambda$ -calcolo è stato un'ispirazione fondamentale per il progetto dei linguaggi di programmazione, a partire da ALGOL e LISP. Concetti quali scope, ambiente locale, passaggio per nome, per citare solo i più importanti, sono stati "importati" nei linguaggi di programmazione a partire dai concetti analoghi del  $\lambda$ -calcolo.

<sup>10</sup>Usiamo questa estensione per soli scopi di semplicità didattica: nel  $\lambda$  calcolo è possibile codificare numeri e condizionali usando solo variabili, astrazione e applicazione.

induttive del Paragrafo 8.5). Per questa equazione molto semplice è evidente che una soluzione (anzi, in questo caso, l'unica soluzione) è la funzione identità

$$Id(n) = n. \quad (13.2)$$

Quello di cui abbiamo bisogno è un metodo generale che ci permetta di passare da un'equazione quale la (13.1) ad un *algoritmo*, cioè un  $\lambda$ -termine, che sia una soluzione di quell'equazione quando il simbolo di uguaglianza sia interpretato come  $\beta$ -uguaglianza

$$Z =_{\beta} \lambda n. \text{if } n = 0 \text{ then } 0 \text{ else } 1 + Z(n - 1). \quad (13.3)$$

Si osservi che questo è chiedere di più che fornire una generica funzione come la (13.2): si richiede un termine che nel calcolo soddisfi la (13.3)<sup>11</sup>.

Consideriamo in primo luogo la funzione che si ottiene dall'equazione (13.3)  $\lambda$ -astraiendo sulla funzione incognita

$$F \stackrel{\text{def}}{=} \lambda f. \lambda n. \text{if } n = 0 \text{ then } 0 \text{ else } 1 + f(n - 1).$$

Possiamo riformulare il nostro problema come la ricerca di un'opportuna funzione  $g$  che soddisfi l'equazione

$$g =_{\beta} F(g).$$

Un termine  $g$  che soddisfi un'equazione del genere è detto *punto fisso* di  $F$ . Dunque il nostro problema di determinare una soluzione alla (13.3) può essere riformulato come un problema di ricerca del punto fisso di un'opportuna funzione.

Quello che adesso vogliamo mostrare è che il  $\lambda$ -calcolo fornisce un metodo generale per risolvere "automaticamente" tali equazioni. Consideriamo infatti il seguente termine, che per convenienza (e secondo la tradizione) indicheremo col nome  $Y$

$$Y \stackrel{\text{def}}{=} \lambda f. (\lambda x. f(xx))(\lambda x. f(xx)).$$

Si tratta di un termine che può essere ridotto all'infinito senza raggiungere una forma normale. Ma non è privo di senso: proviamo infatti ad applicarlo ad un generico termine  $M$ :

$$\begin{aligned} YM &=_{\beta} (\lambda x. M(xx))(\lambda x. M(xx)) \\ &=_{\beta} M((\lambda x. M(xx))(\lambda x. M(xx))) \\ &=_{\beta} M(YM), \end{aligned}$$

dove nell'ultimo passaggio abbiamo usato il fatto, stabilito nella prima riga, che  $YM =_{\beta} (\lambda x. M(xx))(\lambda x. M(xx))$  ed abbiamo poi sostituito uguali con uguali.  $YM$  è dunque un punto fisso di  $M$ :  $Y$  è un termine del calcolo che è in grado di

<sup>11</sup> Il termine  $\lambda n. n$ , che pure è un algoritmo per l'unica funzione che soddisfa la (13.1), non è una soluzione dell'equazione (13.3), che esprime un determinato comportamento per riduzione.

calcolare il punto fisso di un termine arbitrario<sup>12</sup>. Si dice che  $Y$  è un *operatore di punto fisso*. È allora del tutto evidente che  $Y$  fornisce soluzione ad una qualunque equazione ricorsiva. In particolare, ci dà una soluzione per l'equazione (13.3):

$$Id \stackrel{\text{def}}{=} YF.$$

Il termine  $Id$  così definito ha un comportamento per riduzione che soddisfa la relazione da cui siamo partiti. È istruttivo provare ad applicare questa funzione ad uno specifico argomento, per vedere come il meccanismo di punto fisso sia davvero in grado di calcolare una funzione ricorsiva. Ad esempio

$$\begin{aligned} Id 2 &\stackrel{\text{def}}{=} (YF) 2 \\ &\rightarrow F(YF) 2 \\ &\rightarrow \text{if } 2 = 0 \text{ then } 0 \text{ else } 1 + ((YF)(2 - 1)) \\ &\rightarrow 1 + ((YF) 1) \\ &\rightarrow 1 + (F(YF) 1) \\ &\rightarrow 1 + ((\text{if } 1 = 0 \text{ then } 0 \text{ else } 1 + ((YF)(1 - 1)))) \\ &\rightarrow 1 + (1 + ((YF) 0)) \\ &\rightarrow 1 + (1 + (\text{if } 0 = 0 \text{ then } 0 \text{ else } 1 + ((YF)(0 - 1)))) \\ &\rightarrow 1 + (1 + 0) \\ &\rightarrow 2. \end{aligned}$$

Ad ogni chiamata "ricorsiva"  $YF$  fornisce una copia nuova del testo della funzione, che viene usata per la prossima chiamata.

Quello che abbiamo appena osservato rimane vero per la definizione di valori ricorsivi anche non funzionali. Lo stream che abbiamo definito nel Paragrafo 13.3.5 come

`val infiniti2 = 2 :: infiniti2;`

non è altro che un punto fisso:

$$\text{infiniti2} \stackrel{\text{def}}{=} Y(\lambda l. 2 :: l).$$

Rimane da osservare che  $Y$  è un operatore di punto fisso rispetto alla  $\beta$ -uguaglianza<sup>13</sup>. Se, invece della  $\beta$ -riduzione usiamo una diversa strategia, per esempio l'analogo della riduzione per valore che abbiamo definito per i linguaggi funzionali,  $Y$  non è più in grado di calcolare punti fissi. Per la riduzione per valore deve essere usato un altro operatore, per esempio

$$H \stackrel{\text{def}}{=} \lambda g. ((\lambda f. ff)(\lambda f. (g(\lambda x. fx)))).$$

<sup>12</sup> Questo dimostra anche che, nel  $\lambda$ -calcolo, ogni termine ha sempre almeno un punto fisso, quello calcolato da  $Y$ .

<sup>13</sup> Ovviamente non è l'unico; per un altro operatore, leggermente più complesso di  $Y$  ma che soddisfa una relazione più forte, si veda l'Esercizio 7.

Per motivi di efficienza, le implementazioni dei linguaggi funzionali non usano la tecnologia degli operatori di punto fisso per realizzare la ricorsione. Gli operatori di punto fisso giocano tuttavia un ruolo assai importante nella teoria di questi linguaggi.

**Espressività del  $\lambda$ -calcolo** Questo semplicissimo sistema formale, costruito attorno alle sole nozioni di applicazione e astrazione, non è poi così rudimentale come sembra, se è in grado di esprimere ricorsioni arbitrarie. In effetti, è possibile mostrare che il  $\lambda$ -calcolo è un formalismo Turing completo: in primo luogo è possibile codificare i numeri naturali come  $\lambda$ -termini (al numero  $n$  corrisponderà un opportuno  $\lambda$ -termine  $\underline{n}$ ) e, quindi, mostrare che ad ogni funzione calcolabile  $f$  corrisponde un  $\lambda$ -termine  $M_f$  che calcola  $f$ : se  $f(n) = m$  allora  $M_f \underline{n} \rightarrow \underline{m}$ .

## 13.7 Sommario del capitolo

Abbiamo presentato in questo capitolo il paradigma di programmazione funzionale. Nella sua forma *pura* si tratta di un modello di calcolo che non fa riferimento alla nozione di variabile modificabile: la computazione procede per riscrittura di termini che denotano funzioni. I principali temi che abbiamo discusso sono:

- l'*astrazione*, un meccanismo per passare da un'espressione che denota un valore ad un'espressione che denota una funzione;
- l'*applicazione funzionale*, un meccanismo duale rispetto all'astrazione col quale una funzione viene applicata ad un argomento;
- la computazione per *riscrittura* (o per *riduzione*), nel quale un'espressione viene ripetutamente semplificata sino al raggiungimento di una forma non ulteriormente riducibile;
- il *redex*, la struttura sintattica che viene semplificata durante la riduzione, costituita dall'applicazione di un'astrazione ad un'espressione;
- la centralità dell'*ordine superiore* in questo modello di calcolo;
- la nozione di *valore*, che corrisponde a quelle forme sintattiche non ulteriormente riducibili in corrispondenza delle quali la riduzione termina;
- le diverse *strategie di valutazione*: per valore (o *eager*), per nome, *lazy*.
- alcuni meccanismi che i linguaggi funzionali più comuni aggiungono al nucleo fondamentale; tra questi ricordiamo: un ambiente interattivo, un ricco sistema di tipi, il pattern matching, aspetti imperativi quali variabili ed assegnamenti;
- la *macchina SECD*, un prototipo di macchina astratta per linguaggi funzionali di ordine superiore con strategia per valore;
- un confronto tra il paradigma funzionale e quello imperativo, centrato sulla nozione di correttezza dei programmi;
- il  $\lambda$ -*calcolo*, un sistema formale semplice e potente che costituisce il nucleo formale di tutti i linguaggi funzionali.

## 13.8 Nota bibliografica

Su LISP è ancora una lettura didatticamente importante l'articolo originale di John McCarthy [65]. Un'introduzione elementare alla programmazione usando ML è [99], mentre [30] è un testo più avanzato (che usa il dialetto Caml); [70] è la definizione ufficiale del linguaggio, che è opportuno leggere assieme al suo commento [69].

L'implementazione dei linguaggi funzionali, con particolare riguardo alla valutazione lazy è trattata in [78]. La macchina SECD è introdotta nel lavoro pionieristico di Peter Landin [52].

La lezione di John Backus in occasione del conferimento del premio Turing [11] dovrebbe essere una lettura obbligata (almeno nella sua prima parte) per ogni studente di linguaggi di programmazione.

Il lambda calcolo è stato introdotto negli anni trenta da Alonzo Church; un riferimento originale è [25]. Il testo di riferimento moderno è [13], anche se [43] è un'introduzione più piana ed accessibile.

## 13.9 Esercizi

1. Si considerino le seguenti definizioni in ML:

```
fun K x y = x;
fun I x = x;
fun Omega x = Omega x;
```

Si dica qual è il risultato della valutazione dell'espressione  
 $K(I\ 3)\ (\Omega\ 1)$ .

Quale sarebbe il risultato se ML adottasse una strategia lazy?

2. Si scriva una funzione *map* che applichi il suo primo parametro (che sarà una funzione) a tutti gli elementi di una lista passata come secondo argomento.
3. Si descriva in modo formale il funzionamento della macchina SECD, dando una definizione usando la semantica operazionale strutturata che abbiamo introdotto nel Paragrafo 2.5.
4. In  $\lambda$ -calcolo, possiamo definire la codifica di un numero naturale  $n$  come

$$\underline{n} \stackrel{\text{def}}{=} \lambda f. \lambda x. f^n x \stackrel{\text{def}}{=} \lambda f. \lambda x. f(f \cdots (fx) \cdots),$$

dove nel corpo vi sono  $n$  occorrenze di  $f$ . Si dia la definizione di un termine *prod* tale che

$$\text{prod } \underline{n} \underline{m} \rightarrow \underline{n * m}$$

5. Si applichi la tecnica del calcolo del punto fisso per risolvere la seguente equazione nell'incognita *fatt*

$$\text{fatt} =_{\beta} \lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } n * \text{fatt}(n - 1)$$

e si descriva la riduzione che calcola il valore della sua soluzione applicata all'argomento 2.

6. Si applichi la tecnica del calcolo del punto fisso per risolvere la seguente equazione nell'incognita  $h$

$$h =_{\beta} \lambda x. \text{if } x > 100 \text{ then } x - 10 \text{ else } h(h(x + 11)).$$

Detta *Novantuno* la sua soluzione, si calcoli poi *Novantuno*(99).

7. Si considerino i  $\lambda$ -termini

$$\begin{aligned} A &= \lambda x. \lambda y. y(xxy) \\ \Theta &= AA. \end{aligned}$$

Si mostri che  $\Theta$  è un operatore di punto fisso per il quale vale la relazione

$$\Theta M \rightarrow \dots \rightarrow M (\Theta M).$$

Questa relazione è più forte di quella soddisfatta da  $Y$ , per il quale vale solo  $YM =_{\beta} M(YM)$ . Il termine  $\Theta$  è detto operatore di punto fisso di Turing.

## Il paradigma logico

In questo capitolo analizziamo l'altro paradigma che, insieme a quello funzionale, permette la programmazione dichiarativa: il paradigma logico che include sia linguaggi teorici sia linguaggi effettivamente implementati ed usati, dei quali il più noto è sicuramente PROLOG. Anche se vi sono differenze abbastanza rilevanti di ordine pragmatico e, per certi versi, teorico, fra i vari linguaggi logici, questi condividono l'idea di intendere la computazione come deduzione logica.

In questo capitolo vedremo dunque queste nozioni, cercando di limitare la parte teorica, adottando anche per questo paradigma l'approccio che ha caratterizzato tutto il testo. Non intendiamo quindi insegnare a programmare in PROLOG, anche se faremo vedere vari esempi di programmi reali, ma intendiamo fornire la base per poter comprendere e, in breve tempo, padroneggiare, questo ed altri linguaggi logici.

### 14.1 Deduzione come computazione

Un noto slogan, dovuto a R. Kowalski, così sintetizza la nozione alla base dell'attività di programmazione: Algoritmo = Logica + Controllo. Secondo questa "equazione" la specifica di un algoritmo, e quindi la sua formulazione in termini di un linguaggio di programmazione, può essere separata in due parti. Da un lato si specifica la logica della soluzione, ossia si definisce "cosa" debba essere fatto. Dall'altro invece si specificano gli aspetti relativi al controllo, e quindi si chiarisce "come" si deve arrivare alla soluzione desiderata. Il programmatore che usa un tradizionale linguaggio imperativo deve tener conto di entrambe queste componenti, usando i meccanismi che abbiamo diffusamente analizzato nei capitoli precedenti. La programmazione logica, invece, fu definita originariamente con l'idea di separare nettamente questi due aspetti. Al programmatore è richiesta solo, almeno in linea di principio, la specifica della parte logica. Tutto quanto riguarda il controllo è demandato alla macchina astratta: usando un meccanismo computazionale basato su una particolare regola di deduzione (la risoluzione), essa ricerca nello spazio delle possibili soluzioni quella specificata dalla "logica", definendo in questo modo la sequenza di operazioni necessarie ad arrivare al risultato finale.

Le basi di questa visione della computazione come deduzione logica possono essere ricondotte ai lavori di K. Gödel e J. Herbrand negli anni '30. In particolare, Herbrand anticipò, anche se in modo non completamente formale, alcune idee relative al processo di unificazione che, come vedremo, costituisce il meccanismo computazionale di base dei linguaggi logici.

Bisognerà attendere fino agli '60 per una definizione formale di tale processo (ad opera di A. Robinson) e solo dieci anni ancora più tardi (nei primi anni '70) ci rese conto che la dimostrazione automatica di formule di un tipo particolare poteva essere interpretata come un meccanismo computazionale. Nascevano così i primi linguaggi di programmazione del paradigma logico, tra cui PROLOG (il nome è un acronimo per PROGramming in LOGic, per ulteriori informazioni sulla storia del linguaggio si veda il Paragrafo 16.4).

Oggi ci sono molte versioni implementate di PROLOG ed esistono vari altri linguaggi logici reali (di particolare interesse, per le applicazioni, sono quelli con vincoli). Tutti questi linguaggi, PROLOG per primo, per motivi di efficienza permettono l'uso di costrutti che riguardano anche aspetti di specifica del controllo. Questi costrutti, non avendo una diretta interpretazione logica, rendono molto più complicata la semantica del programma e fanno perdere parte della natura puramente dichiarativa del paradigma logico. Nonostante ciò, anche in presenza di questi aspetti "impuri", si tratta pur sempre di linguaggi di programmazione che richiedono al programmatore poco più che formulare (o dichiarare) la specifica del problema da risolvere. In alcuni casi i programmi risultanti sono davvero sorprendenti per semplicità, brevità e chiarezza, come vedremo nel prossimo paragrafo.

**Nota terminologica** Nell'ambito dei linguaggi logici si distingue la programmazione logica (*logic programming*), che costituisce il formalismo teorico, dal PROLOG, un linguaggio esistente in molte implementazioni diverse del qual è anche stato definito recentemente uno standard. Le nozioni che introdurremo nel seguito, se non diversamente specificato, valgono per entrambi i formalismi. Le differenze importanti saranno esplicitamente sottolineate. Gli esempi di programmi che forniremo (al solito usando la fonte monospaziata) sono tutti codici PROLOG che dovrebbero "girare" su una qualsiasi implementazione del linguaggio. Le nozioni teoriche invece, anche nel caso in cui valgano per PROLOG, usano la notazione matematica con i caratteri in *corsivo*. Infine, in questo capitolo seguiranno la convenzione di PROLOG e indicheremo sempre le variabili usando stringhe di caratteri che iniziano con una lettera maiuscola.

### 14.1.1 Un esempio

Vediamo di sostanziare quanto detto nel paragrafo precedente con un esempio che, per il momento, rimane abbastanza informale, dato che non abbiamo ancora introdotto né sintassi né, tanto meno, semantica dei linguaggi logici.

Consideriamo dunque il seguente problema: si vogliono disporre tre 1, tre 2, tre 3, ..., tre 9 in una lista (che dunque conterrà 27 numeri) in modo tale che, per

ogni  $i \in [1, 9]$ , vi siano esattamente  $i$  numeri fra due occorrenze successive (nella lista) del numero  $i$ . Così, ad esempio,  $\dots 1, 2, 1, 8, 2, 4, 6, 2 \dots$  potrebbe essere una parte della soluzione finale, mentre  $\dots 1, 2, 1, 8, 2, 4, 2, 6 \dots$  non può esserlo (perché fra le ultime due occorrenze di 2 c'è un solo numero). Il lettore è invitato a provare a scrivere un programma che risolva questo problema, usando il suo linguaggio di programmazione imperativo preferito. Non si tratta di un esercizio proibitivo ma richiede comunque una qualche attenzione perché se "cosa" deve essere fatto è chiaro, "come" si debba ottenere la soluzione non è immediatamente evidente. Vi sono infatti da definire aspetti di controllo non immediati. Ad esempio, in modo molto naif (e assolutamente inefficiente) si potrebbe pensare di generare tutte le possibili permutazioni di una lista di 27 numeri contenente tre 1, tre 2, tre 3, ..., tre 9, per poi controllare se una delle permutazioni soddisfa la proprietà richiesta. Anche questa soluzione, probabilmente una delle più semplici, richiede comunque la specifica nel dettaglio degli aspetti di controllo necessari a generare le permutazioni di una lista.

Ragionando in modo dichiarativo, invece, possiamo procedere come segue. Innanzitutto ci serve una lista, chiamiamola  $Ls$ . Questa lista dovrà contenere 27 elementi, cosa che possiamo specificare usando un predicato (ossia un simbolo di relazione) unario<sup>1</sup> `lista_di_27`: se scriviamo `lista_di_27(Ls)` intendiamo dunque dire che  $Ls$  deve essere una lista di 27 elementi ossia, in altri termini, `lista_di_27(Ls)` definisce una relazione costituita da tutte le possibili liste di 27 elementi. Per ottenere questo scopo possiamo definire `lista_di_27` come segue

```
lista_di_27(Ls) :-  
  Ls = [_, _, _, _, _, _, _, _, _, _, _, _, _, _, _, _, _, _, _, _, _, _, _, _, _, _, _, _, _, _, _].
```

dove la parte a sinistra del simbolo `:`- indica quello che si deve definire e la parte a destra la definizione. Il simbolo `=` indica una nozione di uguaglianza per la quale, per il momento, ci accontentiamo di una interpretazione intuitiva (diversa però dall'assegnamento). Anticipando la notazione di PROLOG per le liste, qui scriviamo `[X1, X2, ..., Xn]` per indicare la lista che contiene le  $n$  variabili  $X_1, X_2, \dots, X_n$  (si veda il Paragrafo 14.4.5 per ulteriori dettagli sulle liste).

La lista  $Ls$ , oltre ad essere fatta da 27 elementi, per soddisfare le nostre specifiche dovrà contenere una sottolista<sup>2</sup> nella quale compare il numero 1 seguito da altro numero qualsiasi, da una nuova occorrenza del numero 1, da un altro numero qualsiasi ed infine un'ultima occorrenza del numero 1. Una tale sottolista può essere specificata da

```
[1, X, 1, Y, 1]
```

dove  $X$  e  $Y$  sono variabili. Ancora più efficacemente possiamo scrivere

```
[1, _, 1, _, 1]
```

<sup>1</sup>Ricordiamo che unario significa che ha un solo argomento mentre  $n$ -ario indica che vi sono  $n$  argomenti.

<sup>2</sup>Ricordiamo che  $L_i$  è una sottolista di  $Ls$  se  $Ls$  è ottenuta concatenando una lista (eventualmente vuota) con  $L_i$  e con un'altra lista (anch'essa eventualmente vuota).

dove `_` indica la variabile anonima, ossia una variabile della quale non ci interessa il nome e che è distinta da tutte le altre variabili presenti (incluse le altre variabili anonime). Supponendo di avere a disposizione un predicato binario `sublist(X, Y)` il cui significato è che il primo argomento (`X`) è una sottolista del secondo (`Y`)<sup>3</sup>, il nostro requisito può dunque essere espresso scrivendo

```
sublist([1, _, 1, _, 1], Ls)
```

Passando a considerare il requisito per il numero 2 e ragionando in modo analogo, otteniamo che la lista `Ls` deve contenere anche la sottolista

```
[2, _, _, 2, _, _, 2]
```

e dunque deve valere anche

```
sublist([2, _, _, 2, _, _, 2], Ls)
```

Possiamo ripetere questo ragionamento fino al numero 9, per cui il programma SEQUENZA che vogliamo ottenere può essere descritto come segue.

```
sol(Ls) :-  
    lista_di_27[Ls],  
    sublist([9, _, _, _, _, _, _, 9, _, _, _, _, _, _, 9], Ls),  
    sublist([8, _, _, _, _, _, 8, _, _, _, _, _, 8], Ls),  
    sublist([7, _, _, _, _, 7, _, _, _, _, 7], Ls),  
    sublist([6, _, _, _, 6, _, _, _, 6], Ls),  
    sublist([5, _, _, _, 5, _, _, 5], Ls),  
    sublist([4, _, _, _, 4, _, _, 4], Ls),  
    sublist([3, _, _, 3, _, _, 3], Ls),  
    sublist([2, _, _, 2, _, _, 2], Ls),  
    sublist([1, _, 1, _, 1], Ls).
```

```
lista_di_27(Ls) :-  
    Ls = [_, _, _, _, _, _, _, _, _, _, _, _, _, _, _, _, _, _].
```

dove, come prima, la parte a destra del simbolo `:-` è la definizione di quello che sta a sinistra. Quanto scritto sopra esprime che per trovare una soluzione al nostro problema (`sol(Ls)` in linea 1) devono essere soddisfatte tutte le proprietà descritte nelle linee 2-11. Le virgolette che separano i vari predicati sono da interpretarsi come “e”, ossia come congiunzioni in senso logico. Quella che abbiamo fornito dunque non è altro che una specifica che, nella sostanza, ripete in modo formale la formulazione del problema e che, come vedremo più avanti, può essere interpretata in termini puramente logici.

Questa lettura dichiarativa, per quanto elegante e compatta, non sarebbe di grande aiuto per la soluzione del nostro problema se non fosse accompagnata da un’interpretazione procedurale e dovesse quindi essere tradotta in un linguaggio di programmazione convenzionale. La caratteristica sostanziale del paradigma che stiamo considerando è proprio che le specifiche (logiche) che scriviamo sono a tutti gli effetti dei programmi eseguibili. Il codice descritto sopra dalle linee 1-14 (più la definizione di `sublist` data nel Paragrafo 14.4.5) è infatti un vero e

proprio programma PROLOG che, come tale, può essere valutato da un interprete per ottenere la soluzione richiesta.

In altri termini, la nostra specifica può anche essere letta in modo procedurale, come segue: la linea di codice (1) contiene la dichiarazione di una procedura, di nome `sol`, che ha un unico parametro formale `Ls`. Il corpo di tale procedura è quello definito nelle linee di codice da (2) a (11), dove troviamo le seguenti dieci chiamate di procedura: nella linea (2) viene chiamata la procedura `lista_di_27` con parametro attuale `Ls`, che è definita come abbiamo visto sopra e che quindi istanzia<sup>4</sup> il suo parametro attuale con una lista di 27 variabili anonime; quindi, nella linea (3) abbiamo la chiamata

```
sublist([9, _, _, _, _, _, 9, _, _, _, _, _, _, 9], Ls)
```

Assumiamo che questa chiamata faccia sì che la lista che compare come primo parametro sia una sottolista della lista che risulta legata al secondo parametro (`Ls`), eventualmente istanziando opportunamente la variabile `Ls`; analogamente per le altre chiamate fino a quella nella linea 11. Il passaggio dei parametri avviene in modo simile al passaggio per nome e, in PROLOG, l’ordine con cui le varie chiamate di procedura compaiono nel testo specifica l’ordine di valutazione (nel caso dei programmi logici puri, invece, non è specificato alcun ordine).

Data la precedente definizione della procedura `sol`, la chiamata `sol(Ls)` restituisce `Ls` istanziata con una soluzione del problema, quale ad esempio la sequenza

```
1, 9, 1, 2, 1, 8, 2, 4, 6, 2, 7, 9, 4, 5, 8, 6, 3, 4, 7, 5, 3, 9, 6, 8, 3, 5, 7.
```

Chiamate successive della stessa procedura permettono di ottenere anche le altre soluzioni del problema.

Con questa interpretazione procedurale si ha dunque a disposizione un vero linguaggio di programmazione, che permette di esprimere in modo compatto e relativamente semplice programmi che risolvono problemi anche molto complessi. Per questa potenza del linguaggio c’è tuttavia da pagare un prezzo da pagare in termini di efficienza. Nel programma precedente, nonostante la semplicità esteriore, la computazione svolta dalla macchina astratta del linguaggio è molto complessa, dato che l’interprete si deve occupare di provare le varie combinazioni di sottoliste possibili, fino a trovare quella che soddisfa tutte le condizioni. In questo processo di ricerca si usa il meccanismo del *backtracking*: quando la computazione arriva ad un punto nel quale non può procedere, viene “disfatta” la computazione effettuata per tornare ad un punto di scelta, se questo esiste, dove si sceglie un’alternativa diversa da quelle già seguite (se tale alternativa non esiste la computazione termina con un fallimento). Non è difficile immaginare che, in generale, un tale processo di ricerca può avere complessità esponenziale.

<sup>4</sup>Anche se la terminologia dovrebbe essere intuitivamente chiara, con istanziare una variabile intendiamo il sostituire ad essa un altro oggetto sintattico. Vedremo una definizione precisa nel Paragrafo 14.3.

<sup>3</sup>La definizione di `sublist` è rimandata al Paragrafo 14.4.5.

## 14.2 Sintassi

I programmi logici sono insiemi di formule logiche di una particolare forma. Inizieremo quindi con alcune nozioni di base necessarie per definirne la sintassi.

La logica della quale ci occupiamo è quella del prim'ordine, detta anche calcolo dei predicati. La terminologia si riferisce al fatto che si usano dei simboli per esprimere (o, come si dice con una terminologia un poco obsoleta “predicare”) proprietà di elementi che fanno parte di un dominio del discorso  $\mathcal{D}$  prefissato. Logiche più espansive (del secondo ordine, del terzo ordine ecc.) permettono anche di avere predicati che come argomenti hanno, invece che elementi di  $\mathcal{D}$ , oggetti più complicati quali insiemi e funzioni su  $\mathcal{D}$  (second'ordine), insiemi di funzioni (terz'ordine) ecc.

### 14.2.1 Il linguaggio della logica del prim'ordine

Innanzitutto, come per ogni altro sistema formale, per poter parlare del calcolo dei predicati (e quindi dei programmi logici) dobbiamo definirne il linguaggio. Un *linguaggio del prim'ordine* consiste di tre componenti:

1. un *alfabeto*;
2. i *termini* definiti su tale alfabeto;
3. le *formule ben formate* definite su tale alfabeto.

Vediamo in ordine i vari componenti di questa definizione.

**Alfabeto** L'alfabeto, al solito, è un insieme<sup>5</sup> di simboli. In questo caso consideriamo tale insieme partizionato in due sottoinsiemi disgiunti: l'insieme dei *simboli logici*, comuni a tutti i linguaggi del prim'ordine, e l'insieme dei *simboli non logici*, specifici di un qualche dominio di interesse. Ad esempio, tutti i linguaggi del prim'ordine, verosimilmente, useranno un simbolo (logico) per indicare la congiunzione; se però stiamo considerando degli ordinamenti su di un insieme, probabilmente fra i simboli non logici del linguaggio vorremo avere anche il simbolo  $<$ .

L'insieme dei *simboli logici* contiene i seguenti elementi

- i connettivi logici  $\wedge$  (congiunzione),  $\vee$  (disgiunzione),  $\neg$  (negazione),  $\rightarrow$  (implicazione) e  $\leftrightarrow$  (doppia implicazione);
- le costanti proposizionali *true* e *false*;
- i quantificatori  $\exists$  (esiste) e  $\forall$  (per ogni);
- alcuni simboli di interpunkzione quali le parentesi  $(, )$  e la virgola  $,$  ;
- un insieme infinito (numerabile)  $V$  di *variabili*, indicate con  $X, Y, Z, \dots$

I simboli *non logici* sono definiti da una *segnatura con predicati*  $\langle \Sigma, \Pi \rangle$ : si tratta di una coppia nella quale il primo elemento  $\Sigma$  è la *segnatura delle funzioni*, cioè un insieme di simboli di funzione, considerati ognuno con la propria arietà<sup>6</sup>. Il secondo elemento della coppia  $\Pi$  è la *segnatura dei predicati*, un insieme di simboli di predicato insieme alle loro arietà. Le funzioni di arietà 0 sono dette *costanti* e sono indicate con le lettere  $a, b, c, \dots$ . I simboli di funzione di arietà positiva sono di solito indicati da  $f, g, h, \dots$ , mentre i simboli di predicato sono indicati da  $p, q, r, \dots$ . Assumiamo che gli insiemi  $\Sigma$  e  $\Pi$  abbiano intersezione vuota e siano anche disgiunti dagli altri insiemi di simboli elencati in precedenza. La differenza fra simboli di funzione e di predicato è che i primi devono essere interpretati come funzioni, mentre i secondi devono essere interpretati come relazioni. Questa distinzione sarà più chiara quando parleremo di formule.

**Termini** La nozione di termine, fondamentale nella logica matematica, nell'ambito dell'informatica è usata implicitamente in molti contesti. Ad esempio, un'espressione aritmetica in sostanza è un termine ottenuto applicando degli operatori (aritmetici) a degli operandi. Anche altri tipi di costrutti, quali le stringhe, gli alberi binari, le liste ecc. possono essere convenientemente viste come termini, ottenuti a partire da opportuni costruttori.

Nel caso più semplice un termine è ottenuto applicando un simbolo di funzione a variabili e costanti in modo tale da rispettare l'arietà. Ad esempio, se  $a$  e  $b$  sono costanti,  $X$  e  $Y$  sono variabili e  $f$  e  $g$  hanno arietà 2,  $f(a, b)$  e  $g(a, X)$  sono termini. Nulla però vieta di usare termini come argomenti di una funzione, purché sia sempre rispettata l'arietà. Possiamo ad esempio scrivere  $g(f(a, b), Y)$  o anche  $g(f(a, f(X, Y)), X)$  e così via.

Nel caso più generale possiamo definire i termini come segue.

**Definizione 14.1 (Termini)** I termini sulla segnatura  $\Sigma$  (e sull'insieme di variabili  $V$ ) sono definiti induttivamente<sup>7</sup> come segue:

- una variabile (in  $V$ ) è un termine;
- se  $f$  (in  $\Sigma$ ) è un simbolo di funzione di arietà  $n$  e  $t_1, \dots, t_n$  sono termini, allora  $f(t_1, \dots, t_n)$  è un termine.

Come caso particolare del secondo punto, una costante è un termine: stando alla lettera, un termine che corrisponde ad una costante si dovrebbe scrivere con le parentesi:  $a(), b(), \dots$ ; stabiliamo, per convenzione, che nel caso di simboli di funzione di arietà 0 le parentesi sono omesse. I termini senza variabili sono detti termini *ground*. I termini usualmente sono denotati dalle lettere  $s, t, u, \dots$ . Si noti che nei termini non compaiono predicati: questi compariranno nelle formule (per esprimere proprietà dei termini).

<sup>5</sup>Tutti gli insiemi che considereremo nel seguito sono finiti o numerabili.

<sup>6</sup>Ricordiamo che, come già detto, l'arietà indica il numero di argomenti di una funzione o di una relazione.

<sup>7</sup>Si veda il riquadro di pag. 233 per le definizioni induttive.

**Formule** Le *formula ben formate* (*formule* in breve) del linguaggio ci permettono di esprimere proprietà dei termini che poi, dal punto di vista semantico, sono proprietà di un particolare dominio di interesse. Ad esempio, se abbiamo il predicato  $>$ , interpretato nel modo usuale, scrivendo  $>(3, 2)$  vogliamo esprimere il fatto che il termine 3 corrisponde ad un valore che è maggiore di quello associato al termine 2. I predicati possono poi essere usati per costruire espressioni complesse mediante i simboli logici. Ad esempio, la formula  $>(X, Y) \wedge >(Y, Z) \rightarrow >(X, Z)$  esprime la transitività di  $>$ , in quanto afferma che se vale  $>(X, Y)$  e vale anche  $>(Y, Z)$ , allora vale  $>(X, Z)$ .

Volendo definire con precisione le formule, abbiamo innanzitutto le formule atomiche (o atomi), costruite applicando un predicato a termini in modo tale da rispettare l'arietà. Ad esempio, se  $p$  ha arietà 2, usando due termini introdotti prima possiamo scrivere  $p(f(a, b), f(a, X))$ . Usando i connettivi logici ed i quantificatori, possiamo costruire formule complesse a partire dalle formule atomiche. Al solito, possiamo usare una definizione induttiva (oppure, equivalentemente, una grammatica libera, si veda l'Esercizio 1).

**Definizione 14.2 (Formule)** Le formule (*ben formate*) del linguaggio sulla segnatura con termini  $(\Sigma, \Pi)$  sono definite induttivamente come segue:

1. se  $t_1, \dots, t_n$  sono termini sulla segnatura  $\Sigma$  e  $p \in \Pi$  è un simbolo di predicato di arietà  $n$ , allora  $p(t_1, \dots, t_n)$  è una formula;
2. true e false sono formule;
3. se  $F$  e  $G$  sono formule allora  $\neg F$ ,  $(F \wedge G)$ ,  $(F \vee G)$ ,  $(F \rightarrow G)$  e  $(F \leftrightarrow G)$  sono formule;
4. se  $F$  è una formula e  $X$  è una variabile, allora  $\forall X.F$  e  $\exists X.F$  sono formule.

### 14.2.2 I programmi logici

Una formula della logica del prim'ordine può avere una struttura molto complessa, che influisce sulla difficoltà di determinare una dimostrazione per essa. Nell'ambito della dimostrazione automatica di teoremi e, poi, nel contesto della programmazione logica, sono state identificate particolari classi di formule, dette *clausole*, che si prestano ad essere manipolate più efficientemente, in specie usando una particolare regola di dimostrazione detta *risoluzione*. A noi interessano, in particolare, delle versioni ristrette della nozione di clausola (le *clausole definite*) e della regola di risoluzione (la *risoluzione SLD*, si veda l'Approfondimento 14.1) per le quali il procedimento di ricerca di una dimostrazione non solo è particolarmente semplice, ma permette di calcolare esplicitamente i valori delle variabili necessari alla dimostrazione. Questi valori possono essere considerati come il risultato della computazione, dando luogo ad un modello di calcolo interamente basato sulla deduzione logica. Vedremo meglio tale modello più avanti, per ora ci concentriamo sugli aspetti sintattici.

**Definizione 14.3 (Programma logico)** Siano  $H, A_1, \dots, A_n$  formule atomiche. Una clausola definita (per noi semplicemente "clausola") è una formula della

forma

$$H : -A_1, \dots, A_n.$$

Se  $n = 0$  la clausola è detta unitaria, o fatto, ed il simbolo  $:$  è omesso (ma non il punto terminale). Un programma logico è un insieme di clausole, mentre un programma PROLOG puro è una sequenza di clausole. Una query (o goal) è una sequenza di atomi  $A_1, \dots, A_n$ .

Chiariamo alcuni punti nella precedente definizione. Innanzitutto il simbolo  $:$  -, che non avevamo introdotto nel nostro alfabeto, è semplicemente un sostituto per l'implicazione (rovesciata)  $\leftarrow$  ed è quello comunemente usato nei linguaggi logici reali<sup>8</sup>.

La virgola in una clausola o in una query dal punto di vista logico è da intendersi come congiunzione. La notazione " $H : -A_1, \dots, A_n.$ " è dunque una abbreviazione per " $H \leftarrow A_1 \wedge \dots \wedge A_n.$ ". Si noti che il punto fa parte della notazione ed è importante, in quanto segnala all'eventuale interprete o compilatore quando termina la clausola che si sta considerando.

La parte a sinistra di  $:$  - è detta *testa* della clausola mentre quella a destra è detta *corpo*. Un fatto è dunque una clausola con corpo vuoto. Un programma è un insieme di clausole, nel caso del formalismo teorico. Nel caso di PROLOG invece un programma è considerato come una sequenza perché, come vedremo, in questo caso l'ordine delle clausole ha rilevanza. Qui abbiamo usato la terminologia semplificata usata in molti testi recenti (ad esempio [9]). Per la terminologia più precisa si veda il riquadro successivo. L'insieme delle clausole che contengono nella testa il simbolo di predicato  $p$  è detto la *definizione* di  $p$ . Le variabili che occorrono nel corpo di una clausola e non nella testa sono dette variabili *locali*.

## 14.3 Teoria dell'unificazione

Il meccanismo fondamentale di computazione nella programmazione logica è la soluzione di equazioni fra termini, realizzata mediante il processo di unificazione. In tale processo vengono calcolate delle sostituzioni che permettono di legare (o istanziare) delle variabili a dei termini. La composizione delle varie sostituzioni ottenute nel corso della computazione fornisce il risultato del calcolo. Prima di vedere nel dettaglio il modello computazionale del paradigma logico dobbiamo quindi analizzare alcuni aspetti riguardanti l'unificazione, cosa che facciamo in questo paragrafo.

### 14.3.1 La variabile logica

Prima di entrare nel dettaglio del processo di unificazione, è importante chiarire che la nozione di variabile che qui stiamo considerando è diversa da quella vista

<sup>8</sup>Si usa  $:$  - per motivi pragmatici: sulle tastiere, soprattutto al tempo in cui vennero introdotti i linguaggi logici, era molto più semplice realizzare un  $:$  che una freccia a sinistra.

**Clausole**

Il lettore familiare con la logica del prim'ordine avrà riconosciuto nella nozione di clausola che abbiamo dato (Definizione 14.3) un caso particolare di quella che si usa in logica. Un clausola, in senso generale, è infatti una formula del tipo

$$\forall X_1, \dots, X_m (L_1 \vee L_2 \vee \dots \vee L_n)$$

dove  $L_1, \dots, L_n$  sono *letterali*, ovvero atomi oppure atomi negati, e  $X_1, \dots, X_n$  sono tutte le variabili che occorrono in  $L_1, \dots, L_n$ . Per maggiore chiarezza, se pariamo gli atomi negati dagli altri e dunque vediamo una clausola come una formula della forma

$$\forall X_1, \dots, X_m (A_1 \vee A_2 \vee \dots \vee A_m, \neg B_1 \vee \neg B_2 \vee \dots \vee \neg B_k).$$

Usando la nota equivalenza della logica che permette di esprimere l'implicazione in termini di una disunione, possiamo esprimere una tale formula nella seguente forma speciale, equivalente alla precedente

$$A_1, \dots, A_m \leftarrow B_1, \dots, B_k. \quad (14.1)$$

Una *clausola di programma*, detta anche *clausola definita*, è una clausola che ha un solo atomo non negato; nella forma espressa dalla (14.1) una clausola definita ha sempre  $m = 1$ , che è proprio la nozione da noi introdotta con la Definizione 14.3. Un fatto è dunque un clausola definita senza atomi negativi. Infine una *clausola negativa*, detta anche query o goal, è una clausola in forma (14.1) nella quale  $m = 0$ .

nel Paragrafo 8.2.1. Qui consideriamo la cosiddetta *variabile logica* che, analogamente alla variabile di cui abbiamo già parlato, costituisce un'incognita che può assumere i valori di un insieme prefissato. Nel nostro caso tale insieme è quello dei termini definiti sull'alfabeto dato. Questo, unitamente all'uso che della variabile logica viene fatto nel paradigma logico, fa sì che vi siano tre importanti differenze fra questa nozione e la variabile modificabile dei linguaggi imperativi:

- la variabile logica può essere legata una sola volta, nel senso che se una variabile è legata ad un termine questo legame non può essere distrutto (può essere eventualmente modificato il termine, come discutiamo più avanti). Ad esempio, se in un programma logico leghiamo la variabile  $X$  alla costante  $a$ , tale legame non può poi essere sostituito da un altro che lega  $X$  alla costante  $b$ . Questo invece, ovviamente, è possibile nei linguaggi imperativi mediante il comando di assegnamento. Il fatto che il legame di una variabile non possa essere eliminato non significa però che non si possa modificare il valore della variabile. Questa apparente contraddizione merita di essere considerata con qualche attenzione, come facciamo nel punto seguente.

- Il valore di una variabile logica può essere definito parzialmente (o indefinito) per essere poi ulteriormente specificato in seguito. Questo perché un termine legato ad una variabile può contenere, a sua volta, altre variabili logiche. Ad esempio, se la variabile  $X$  è legata al termine  $f(Y, Z)$ , successivi legami delle variabili  $Y$  e  $Z$  modificano anche il valore della variabile  $X$  (se  $Y$  viene legata ad  $a$  e  $Z$  viene legata a  $g(W)$ , il valore di  $X$  diverrà il termine  $f(a, g(W))$ ), e il discorso potrebbe continuare modificando il valore della  $W$ ). Questo meccanismo di specifica del valore di una variabile per approssimazioni successive, per così dire, è tipico dei linguaggi logici ed è molto diverso da quello che si ha nel contesto imperativo, dove un valore assegnato ad una variabile non può essere parzialmente definito.
- Una terza importante differenza riguarda la natura bidirezionale dei legami nel caso delle variabili logiche. Se  $X$  è legata al termine  $f(Y)$  e successivamente proviamo a legare  $X$  al termine  $f(a)$ , l'effetto che produciamo è quello di legare la variabile  $Y$  alla costante  $a$ . Questo non contraddice il primo punto, dato che il legame di  $X$  al termine  $f(Y)$  non viene distrutto, ma viene solo ulteriormente specificato il valore  $f(Y)$  per mezzo del legame effettuato per la  $Y$ . Quindi, non solo possiamo modificare il valore di una variabile modificando il termine ad essa legato, ma possiamo anche modificare tale termine fornendo un altro legame per la variabile stessa; ovviamente tale secondo legame deve essere consistente con il primo, cioè se  $X$  è legata al termine  $f(Y)$  non possiamo cercare di legare  $X$  ad un termine della forma  $g(Z)$ .

L'ultimo punto, fondamentale, è quello che permette di usare in molti modi diversi, come vedremo, uno stesso programma logico. In sostanza si tratta di una caratteristica dovuta alla presenza del meccanismo di unificazione, del quale parleremo fra un attimo. Prima però dobbiamo introdurre la nozione di sostituzione.

**14.3.2 Sostituzione**

Il legame fra variabili e termini è realizzato mediante la nozione di sostituzione, che, come dice il nome stesso, permette di "sostituire" un termine a l posto di una variabile. Una sostituzione, indicata usualmente con le lettere greche  $\vartheta, \sigma, \rho, \dots$  può essere definita come segue.

**Definizione 14.4 (Sostituzione)** *Un sostituzione è una funzione da variabili a termini tale che il numero di variabili che non sono mappate in se stesse è finito. Indichiamo una sostituzione  $\vartheta$  usando la notazione*

$$\vartheta = \{X_1/t_1, \dots, X_n/t_n\}$$

*dove  $X_1, \dots, X_n$  sono variabili diverse,  $t_1, \dots, t_n$  sono termini e dove assumiamo che  $t_i$  sia diverso da  $X_i$ , per  $i = 1, \dots, n$ .*

Nella precedente definizione una coppia  $X_i/t_i$  è detta legame (binding); nel caso in cui tutti i  $t_1, \dots, t_n$  siano termini ground allora  $\vartheta$  è detta sostituzione ground;

$\varepsilon$  denota la sostituzione vuota. Per  $\vartheta$  rappresentata come nella Definizione 14.4, definiamo il dominio, il codominio e le variabili della sostituzione come segue:

$$\begin{aligned} \text{Dominio}(\vartheta) &= \{X_1, \dots, X_n\}, \\ \text{Codominio}(\vartheta) &= \{Y \mid Y \text{ è una variabile in } t_i, \text{ per qualche } t_i, 1 \leq i \leq n\}, \end{aligned}$$

Una sostituzione può essere applicata ad un termine o, più in generale, ad una qualsiasi espressione sintattica, per modificare il valore delle variabili presenti nel dominio della sostituzione. Più precisamente, se consideriamo una generica espressione  $E$  (che, può essere un termine, un letterale, una congiunzione di atomi ecc.) il risultato dell'applicazione di  $\vartheta = \{X_1/t_1, \dots, X_n/t_n\}$  ad  $E$ , indicato con  $E\vartheta^9$ , è ottenuto rimpiazzando simultaneamente tutte le occorrenze di  $X_i$  in  $E$  mediante il corrispondente  $t_i$ , per ogni  $1 \leq i \leq n$ . Così, ad esempio, se applichiamo la sostituzione  $\vartheta = \{X/a, Y/f(W)\}$  al termine  $g(X, W, Y)$  otteniamo il termine  $g(X, W, Y)\vartheta$  e cioè  $g(a, W, f(W))$ . Si noti che l'applicazione è simultanea: ad esempio, se applichiamo la sostituzione  $\sigma = \{Y/f(X), X/a\}$  al termine  $g(X, Y)$  otteniamo  $g(a, f(X))$  (e non  $g(a, f(a))$ ).

La *composizione*  $\vartheta\sigma$  di due sostituzioni  $\vartheta = \{X_1/t_1, \dots, X_n/t_n\}$  e  $\sigma = \{Y_1/s_1, \dots, Y_m/s_m\}$  è definita come la sostituzione ottenuta rimuovendo dall'insieme

$$\{X_1/t_1\sigma, \dots, X_n/t_n\sigma, Y_1/s_1, \dots, Y_m/s_m\}$$

le coppie  $X_i/t_i\sigma$  tali che  $X_i$  è eguale a  $t_i\sigma$  e le coppie  $Y_i/s_i$  tali che  $Y_i \in \{X_1, \dots, X_n\}$ . La composizione è associativa e non è difficile rendersi conto che, per una qualsiasi espressione  $E$ , vale che  $E(\vartheta\sigma) = (E\vartheta)\sigma$ : l'effetto dell'applicazione della composizione è lo stesso che si ottiene applicando successivamente le due sostituzioni che vogliamo comporre.

Ad esempio, componendo

$$\vartheta_1 = \{X/f(Y), W/a, Z/X\} \text{ e } \vartheta_2 = \{Y/b, W/b, X/Z\}$$

otteniamo la sostituzione

$$\vartheta = \vartheta_1\vartheta_2 = \{X/f(b), W/a, Y/b\}.$$

Se applichiamo quest'ultima sostituzione al termine  $g(X, Y, W)$  otteniamo il termine  $g(f(b), b, a)$ , così come lo stesso termine si ottiene applicando a  $g(X, Y, W)$  prima  $\vartheta_1$  e poi  $\vartheta_2$ . Si noti che, nel risultato  $\vartheta$  della composizione, la  $Y$  presente in  $\vartheta_1$  viene istanziata a  $b$  per effetto del legame presente in  $\vartheta_2$ ; la  $X$  presente in  $Z/X$  viene istanziata a  $Z$  per effetto del legame  $X/Z$  presente in  $\vartheta_2$ , dopo di che

<sup>9</sup>Si noti che sia per quanto riguarda la notazione usata per le sostituzioni sia per quella relativa all'applicazione di una sostituzione, vi sono versioni opposte: la notazione per la sostituzione di un termine  $N$  al posto della variabile  $X$  usata a pagina 441 è  $N/X$ , in accordo allo standard del paradigma funzionale. Qui invece tale legame è indicato con  $X/N$ . In generale, qui usiamo la notazione più comune nell'ambito della programmazione logica.

il legame  $Z/Z$  viene eliminato dalla sostituzione risultante (perché realizza un'identità); i legami  $W/b$  e  $X/Z$  presenti in  $\vartheta_2$  infine scompaiono da  $\vartheta$  perché sia  $W$  che  $Z$  compaiono nel dominio di (ossia, a sinistra di un legame in)  $\vartheta_1$ .

Un particolare tipo di sostituzioni è costituito da quelle che semplicemente ridenominano le variabili. Ad esempio, la sostituzione  $\{X/W, W/X\}$  altro non fa che cambiare di nome alle variabili  $X$  e  $W$ . Tali sostituzioni sono dette ridenominazioni (o renaming) e possono essere definite come segue.

**Definizione 14.5 (Ridenominazione)** Una sostituzione  $\rho$  è una ridenominazione se esiste la sua sostituzione inversa  $\rho^{-1}$  tale che  $\rho\rho^{-1} = \rho^{-1}\rho = \varepsilon$ .

Si noti che la sostituzione  $\{X/Y, W/Y\}$  non è un renaming: essa infatti non solo cambia nome a due variabili, ma fa diventare uguali due variabili che in precedenza erano diverse.

Infine ci sarà utile definire un preordine  $\leq$  sulle sostituzioni dove  $\vartheta \leq \sigma$  è letto come  $\vartheta$  è più generale di  $\sigma$ . Definiamo dunque  $\vartheta \leq \sigma$  se (e solo se) esiste una sostituzione  $\gamma$  tale che  $\vartheta\gamma = \sigma$ . Analogamente, date due espressioni  $t$  e  $t'$ , definiamo  $t \leq t'$  ( $t$  è più generale di  $t'$ ) se e solo se esiste  $\vartheta$  tale che  $t\vartheta = t'$ . La relazione  $\leq$  è un preordine e l'equivalenza da esso indotta<sup>10</sup> è detta varianza:  $t$  e  $t'$  sono dunque varianti se  $t$  è un'istanza di  $t'$  e, viceversa,  $t'$  è un'istanza di  $t$ . Non è difficile vedere che questa definizione è equivalente a dire che  $t$  e  $t'$  sono varianti se esiste un renaming  $\rho$  tale che  $t$  è sintatticamente identico a  $t'\rho$ . Queste definizioni possono essere estese a generiche espressioni in modo ovvio. Infine, se  $\vartheta$  è una sostituzione che ha come dominio l'insieme di variabili  $V$  e  $W$  è un sottoinsieme di  $V$ , la *restrizione* di  $\vartheta$  alle variabili in  $W$  è la sostituzione ottenuta considerando solo i legami per le variabili in  $W$ , ossia è la sostituzione definita come segue

$$\{Y/t \mid Y \in W \text{ e } Y/t \in \vartheta\}.$$

Un confronto con il paradigma imperativo può aiutare a comprendere meglio le nozioni che stiamo qui considerando. Come abbiamo visto nel Paragrafo 2.5, nel paradigma imperativo la semantica può essere espressa facendo riferimento ad una nozione di stato che associa ad ogni variabile un valore<sup>11</sup>. Un'espressione contenente delle variabili è valutata rispetto ad uno stato per ottenere un valore completamente definito.

Nel paradigma logico l'associazione dei valori alle variabili è realizzata mediante le sostituzioni. L'applicazione di una sostituzione ad un termine (o ad una espressione più complessa) può essere vista come la valutazione del termine, valutazione che restituisce un altro termine e quindi, in generale, un valore parzialmente definito.

<sup>10</sup>Dato un preordine  $\leq$ , la relazione di equivalenza indotta da  $\leq$  è definita come  $t \equiv t'$  se (e solo se)  $t \leq t'$  e  $t' \leq t$ .

<sup>11</sup>A voler essere precisi, come abbiamo visto nel riquadro di pag. 214, nei linguaggi reali tale associazione è realizzata mediante due funzioni, ambiente e memoria. Questo comunque non cambia la sostanza di quello che stiamo dicendo.

### 14.3.3 L'unificatore più generale

Il meccanismo di computazione di base del paradigma logico è la valutazione di equazioni della forma  $s = t^{12}$ , dove  $s$  e  $t$  sono termini e  $=$  è un simbolo di predicato interpretato come uguaglianza sintattica sull'insieme di tutti i termini ground, insieme detto anche *universo di Herbrand*<sup>13</sup>. Vediamo di chiarire meglio il senso di queste uguaglianze.

Se in un programma logico scriviamo  $X = a$  intendiamo dire che la variabile  $X$  deve essere legata alla costante  $a$ . La sostituzione  $\{X/a\}$  costituisce dunque una soluzione per tale equazione dato che, applicando questa sostituzione all'equazione, otteniamo  $a = a$  che è un'equazione evidentemente soddisfatta. L'analogia sintattica con l'assegnamento dei linguaggi imperativi non deve trarre in inganno, dato che si tratta di una nozione completamente diversa. Infatti, a differenza di quello che possiamo fare in un linguaggio imperativo, qui possiamo anche scrivere  $a = X$  invece di  $X = a$ , ed il significato non cambia (l'uguaglianza che consideriamo è simmetrica, come tutte le relazioni di uguaglianza). Anche l'analogia con l'uguaglianza delle espressioni aritmetiche può essere, per certi versi, fuorviante, come illustrato dal seguente esempio. Supponiamo di avere un simbolo di funzione binario  $+$  che, intuitivamente, esprime la somma di due numeri naturali e consideriamo l'equazione  $3 = 2 + 1$ , dove, per comodità, usiamo la notazione infissa per il  $+$  e rappresentiamo nel modo usuale i numeri naturali. Dato che l'equazione  $3 = 2 + 1$  non contiene variabili, essa può essere o vera (ossia, già risolta) o falsa (cioè non risolvibile). Contrariamente a quello che l'intuizione aritmetica ci suggerirebbe, in un programma logico (puro) tale equazione non è risolvibile. Questo perché, come abbiamo detto, il simbolo  $=$  è interpretato come uguaglianza sintattica sull'insieme dei termini ground (universo di Herbrand): è evidente che, dal punto di vista sintattico, la costante  $3$  è diversa dal termine  $2 + 1$  e trattandosi di termini ground (cioè completamente istanziati) non c'è alcun modo di renderli sintatticamente eguali. Analogamente, l'equazione  $f(X) = g(Y)$  non ha soluzioni (non è risolvibile) perché comunque si istanzino le variabili  $X$  e  $Y$  non possiamo rendere eguali i due diversi simboli di funzione  $f$  e  $g$ . Si noti che anche l'equazione  $f(X) = f(g(X))$  non ha soluzioni, perché la variabile  $X$  nel termine di sinistra dovrebbe essere istanziata con  $g(X)$  e quindi l'eventuale soluzione dovrebbe contenere la sostituzione  $\{X/g(X)\}$ ; però l'applicazione di tale sostituzione al termine di destra istanzierebbe la  $X$ , producendo il termine  $f(g(g(X)))$ , dunque la  $X$  nel termine di sinistra dovrebbe essere istanziata a  $g(g(X))$  invece che a  $g(X)$  e così via, senza mai giungere ad una

<sup>12</sup>Si faccia attenzione alla notazione che sovraccarica, come usuale, il simbolo “ $=$ ”: scrivendo  $\vartheta = \{X_1/t_1, \dots, X_n/t_n\}$  intendiamo dire che  $\vartheta$  è la sostituzione  $\{X_1/t_1, \dots, X_n/t_n\}$ , mentre scrivendo  $s = t$  indichiamo un'equazione.

<sup>13</sup>Il simbolo  $=$  usualmente usa la notazione infissa per migliorare la leggibilità. I vari linguaggi logici reali possono avere delle leggere differenze sintattiche al riguardo e usare anche vari simboli di uguaglianza, con significati diversi. Qui facciamo riferimento alla programmazione logica “pura”.

soluzione<sup>14</sup>. In generale dunque l'equazione  $X = t$  non è risolvibile se  $t$  contiene la variabile  $X$  (ed è diverso da  $X$ ).

Se invece consideriamo l'equazione  $f(X) = f(g(Y))$  questa è risolvibile: *una* soluzione è la sostituzione  $\vartheta = \{X/g(Y)\}$  perché se questa è applicata ai due termini dell'equazione li rende sintatticamente identici: infatti  $f(X)\vartheta$  è identico a  $f(g(Y))$  che è identico  $f(g(Y))\vartheta$ . Detto in termini più formali, la sostituzione  $\vartheta$  *unifica* i due termini dell'equazione ed è pertanto detta *unificatore*. Si noti che abbiamo detto “una soluzione” perché vi sono molte (infinte) sostituzioni che sono unificatori di  $X$  e  $g(Y)$ : basta istanziare  $Y$  nella definizione di  $\vartheta$ . Così, ad esempio, sono unificatori anche le sostituzioni  $\{X/g(a), Y/a\}$ ,  $\{X/g(f(Z)), Y/f(Z)\}$ ,  $\{X/g(f(a)), Y/f(a)\}$  e così via. Tutte queste sostituzioni sono però *meno generali* di  $\vartheta$  secondo il preordine che abbiamo definito in precedenza: ognuna di esse, cioè, si può ottenere componendo  $\vartheta$  con un'opportuna sostituzione. Ad esempio,  $\{X/g(a), Y/a\}$  è eguale a  $\vartheta\{Y/a\}$  (indichiamo la composizione di sostituzioni con la giustapposizione, come già visto). In questo senso, diciamo che  $\vartheta$  è l'*unificatore più generale*, o l'm.g.u., (*most general unifier*) di  $X$  e  $g(Y)$ .

Prima di passare alle definizioni generali, si noti un ultimo particolare importante: il processo di soluzione di un'equazione, e cioè di unificazione, può creare dei legami di natura bidirezionale, ossia non è specificata una direzione nella quale si devono realizzare le associazioni. Ad esempio, una soluzione dell'equazione  $f(X, a) = f(b, Y)$  è data dalla sostituzione  $\{X/b, Y/a\}$ , dove si lega una variabile a sinistra ed una a destra del simbolo  $=$ . Invece nell'equazione  $f(X, a) = f(Y, a)$  possiamo scegliere se legare la variabile a sinistra (usando  $\{X/Y\}$ ) oppure quella a destra (usando  $\{Y/X\}$ ) o anche tutte e due (usando  $\{X/Z, Y/Z\}$ ) nella soluzione dell'equazione.

Quest'aspetto, sul quale ritorneremo, è importante perché permette di realizzare un meccanismo di passaggio dei parametri bidirezionale e, caratteristica unica del paradigma logico, di usare uno stesso programma in molti modi diversi, facendo diventare gli argomenti di input argomenti di output e vice versa, senza alcuna modifica al programma stesso.

Vediamo ora le definizioni formali.

**Definizione 14.6 (M.g.u.)** *Dato un insieme di equazioni  $E = \{s_1 = t_1, \dots, s_n = t_n\}$ , dove  $s_1, \dots, s_n$  e  $t_1, \dots, t_n$  sono termini, la sostituzione  $\vartheta$  è un unificatore per  $E$  se le sequenze  $(s_1, \dots, s_n)\vartheta$  e  $(t_1, \dots, t_n)\vartheta$  sono sintatticamente identiche. Un unificatore di  $E$  è detto unificatore più generale (m.g.u.) se è più generale di ogni altro unificatore di  $E$ , ossia se per ogni altro unificatore  $\sigma$  di  $E$  vale che  $\sigma$  è eguale a  $\vartheta\tau$  per un'opportuna sostituzione  $\tau$ .*

La precedente nozione di unificatore può essere estesa ad altri oggetti sintattici nel modo ovvio. In particolare, diremo che  $\vartheta$  è un unificatore di due atomi  $p(s_1, \dots, s_n)$  e  $p(t_1, \dots, t_n)$  se  $\vartheta$  è un unificatore per  $\{s_1 = t_1, \dots, s_n = t_n\}$ .

<sup>14</sup>Si noti che ammettendo termini infiniti potremmo invece trovare una soluzione.

#### 14.3.4 Un algoritmo di unificazione

Un risultato importante, dovuto a Robinson nel 1965, dimostra che è decidibile il problema di verificare se un insieme di equazioni fra termini è unificabile. La dimostrazione è costruttiva, nel senso che fornisce un *algoritmo di unificazione* che, per ogni insieme di equazioni, produce il loro m.g.u. se l'insieme è unificabile e riporta un fallimento in caso contrario<sup>15</sup>.

Si può anche dimostrare che un m.g.u. è unico a meno di renaming. L'algoritmo di unificazione che adesso vediamo non è quello originale di Robinson, ma è quello di Martelli e Montanari, del 1982, che riprende alcune idee presenti già nella tesi di Herbrand del 1930.

**Algoritmo di unificazione di Martelli e Montanari** Dato un insieme di equazioni

$$E = \{s_1 = t_1, \dots, s_n = t_n\},$$

l'algoritmo produce o un fallimento, o un insieme di equazioni in forma risolta del tipo

$$\{X_1 = r_1, \dots, X_m = r_m\},$$

dove  $X_1, \dots, X_m$  sono variabili tutte diverse che non compaiono nei termini  $r_1, \dots, r_m$ . L'insieme di equazioni è equivalente all'insieme di partenza  $E$  e da esso possiamo ottenere un m.g.u. per  $E$  semplicemente interpretando ogni uguaglianza come un legame; dunque l'm.g.u. cercato è la sostituzione

$$\{X_1/r_1, \dots, X_m/r_m\}.$$

L'algoritmo è non deterministico nel senso che quando vi sono più azioni possibili ne viene scelta una in modo casuale, senza alcuna priorità fra le diverse azioni<sup>16</sup>. L'algoritmo è descritto dai seguenti passi.

1. Scegli in modo non deterministico un'equazione all'interno dell'insieme  $E$ .
2. A seconda del tipo dell'equazione scelta, esegui, se possibile, una delle operazioni specificate come segue, dove a sinistra del simbolo ":" indichiamo il tipo d'equazione e a destra l'azione associata:

- (i)  $f(l_1, \dots, l_k) = f(m_1, \dots, m_k)$ : elimina questa equazione dall'insieme  $E$  e aggiungi ad  $E$  le equazioni  $l_1 = m_1, \dots, l_k = m_k$ ;
- (ii)  $f(l_1, \dots, l_k) = g(m_1, \dots, m_h)$ : se  $f$  è diverso da  $g$  termina con fallimento;
- (iii)  $X = X$ : elimina questa equazione dall'insieme  $E$ ;

<sup>15</sup>L'algoritmo originale di Robinson considera l'unificazione di due soli termini ma questo, ovviamente, non è riduttivo dato che l'unificazione di  $\{s_1 = t_1, \dots, s_n = t_n\}$  può essere vista come l'unificazione di  $f(s_1, \dots, s_n)$  e  $f(t_1, \dots, t_n)$ .

<sup>16</sup>Ritorneremo sul non determinismo in breve, quando parleremo della semantica operazionale dei programmi logici.

(iv)  $X = t$ : se  $t$  non contiene la variabile  $X$  e tale variabile appare in un'altra equazione dell'insieme  $E$ , applica la sostituzione  $\{X/t\}$  a tutte le altre equazioni dell'insieme  $E$ ;

(v)  $X = t$ : se  $t$  contiene la variabile  $X$  termina con fallimento;

(vi)  $t = X$ : se  $t$  non è una variabile elimina questa equazione dall'insieme  $E$  e aggiungi ad  $E$  l'equazione  $X = t$ .

3. se nessuna delle operazioni precedenti è possibile, termina con successo ( $E$  contiene la forma risolta); se invece è stata eseguita un'operazione diversa dalla terminazione con fallimento torna a (1).

Si tratta, come si vede, di un algoritmo molto semplice; discutiamo in dettaglio le varie operazioni (i)-(vi).

Il primo caso è quello di due termini che concordano nel simbolo di funzione più esterno. In questo caso per unificare i due termini dovremo procedere all'unificazione degli argomenti, cosa che facciamo rimpiazzando l'equazione originaria con le equazioni ottenute uguagliando gli argomenti nelle stesse posizioni. Si noti che questo caso include anche le equazioni fra costanti del tipo  $a = a$  (dove  $a$  è un simbolo di funzione di arietà 0) che vengono eliminate senza aggiungere niente.

Il secondo caso produce un fallimento dato che, essendo  $f$  e  $g$  diversi, come abbiamo visto i due termini non sono unificabili.

L'equazione  $X = X$  è eliminata in quanto soddisfatta dalla sostituzione identica che quindi non produce alcun cambiamento alle altre equazioni.

Il quarto caso è quello più interessante. Un'equazione  $X = t$  è già in forma risolta (perché, per ipotesi di questo caso,  $t$  non contiene  $X$ ): in altre parole, la sostituzione  $\{X/t\}$  è l'm.g.u. di questa equazione. Perché l'effetto di tale m.g.u. sia combinato con quelli prodotti dalle altre equazioni, dobbiamo applicare la sostituzione  $\{X/t\}$  a tutte le altre equazioni dell'insieme  $E$ .

Nel caso in cui invece  $t$  contenga la variabile  $X$ , come abbiamo già detto, l'equazione non ha soluzione e quindi l'algoritmo termina con fallimento. È importante notare che questo controllo, detto *occur check* è assente in molte implementazioni di PROLOG per motivi di efficienza. Dunque, molte implementazioni di PROLOG usano un algoritmo di unificazione non corretto!

L'ultimo caso infine serve solo ad ottenere una forma risolta, nella quale le variabili compaiono a sinistra e i termini a destra del simbolo  $=$ .

È facile convincersi che l'algoritmo termina, dato che la profondità dei termini di partenza è finita. È inoltre possibile dimostrare che l'algoritmo produce un m.g.u., ottenuto dall'interpretazione della forma risolta finale delle equazioni come sostituzione.

Considerando con attenzione l'algoritmo, ci si rende conto che il calcolo dell'unificatore più generale avviene in modo incrementale, risolvendo equazioni sempre più semplici fino ad arrivare alla forma risolta. È possibile esprimere questo processo anche in termini di composizione di sostituzioni, come in effetti avviene nel modello operazionale dei linguaggi logici. Vediamo questo punto con un esempio, che costituisce anche un esempio di applicazione dell'algoritmo d'unificazione appena visto. Per semplicità selezioneremo sempre la prima

equazione a sinistra (il risultato finale, tuttavia, non dipende da tale assunzione: una qualsiasi altra regola di selezione porterebbe allo stesso risultato, a meno di renaming).

Consideriamo l'insieme di equazioni

$$E = \{f(X, b) = f(g(Y), W), h(X, Y) = h(Z, W)\}.$$

Selezionando la prima equazione a sinistra, mediante l'operazione descritta al punto (i) l'insieme  $E$  è trasformato in

$$E_1 = \{X = g(Y), b = W, h(X, Y) = h(Z, W)\};$$

usando l'operazione in (iv) otteniamo quindi

$$E_2 = \{X = g(Y), b = W, h(g(Y), Y) = h(Z, W)\};$$

usando (vi) e poi di nuovo (iv) sulla seconda equazione otteniamo infine

$$E_3 = \{X = g(Y), W = b, h(g(Y), Y) = h(Z, b)\},$$

che già contiene la forma risolta della prima equazione dell'insieme  $E$ : difatti la sostituzione

$$\vartheta_1 = \{X/g(Y), W/b\}$$

è un m.g.u. per  $f(X, b) = f(g(Y), W)$ .

Proseguendo con la seconda equazione dell'insieme originario, opportunamente istanziata dalle sostituzioni sin qui calcolate, dall'operazione (i) otteniamo

$$E_4 = \{X = g(Y), W = b, g(Y) = Z, Y = b\};$$

quindi da (vi) otteniamo

$$E_5 = \{X = g(Y), W = b, Z = g(Y), Y = b\}.$$

Infine da (iv), applicata all'ultima equazione, abbiamo

$$E_6 = \{X = g(b), W = b, Z = g(b), Y = b\},$$

che costituisce la forma risolta dell'insieme  $E$  e quindi fornisce anche l'm.g.u. dell'insieme iniziale, nella forma della sostituzione

$$\vartheta = \{X/g(b), W/b, Z/g(b), Y/b\}.$$

Ci sono due osservazioni, importanti, da fare. Innanzitutto si noti come il valore di alcune variabili possa essere prima specificato parzialmente, per poi essere ulteriormente definito in un tempo successivo. Ad esempio,  $\vartheta_1$  (m.g.u. della prima equazione di  $E$ ) ci dice che  $X$  ha come valore il termine  $g(Y)$  e solo risolvendo anche la seconda equazione vediamo che  $Y$  ha come valore  $b$ , e quindi che  $X$  ha valore  $g(b)$  (come in effetti risulta nell'm.g.u. finale).

Inoltre, osserviamo che se consideriamo  $\{h(g(X), Y) = h(Z, W)\}\vartheta_1$  (la seconda equazione di  $E$  istanziata dall'm.g.u. della prima), otteniamo l'equazione

$$h(g(Y), Y) = h(Z, b)$$

per la quale la sostituzione

$$\vartheta_2 = \{Z = g(b), Y = b\}$$

è un m.g.u. Usando la definizione di composizione di sostituzione è facile verificare che  $\vartheta = \vartheta_1\vartheta_2$ : l'm.g.u. dell'insieme  $E$  si può dunque ottenere componendo l'm.g.u. della prima equazione con quello della seconda, alla quale sia applicato il primo m.g.u. Questo, come già dicevamo, è quello che in effetti avviene normalmente nelle implementazioni dei linguaggi logici, dove invece che accumulare tutte le equazioni per poi risolverle alla fine, ad ogni passo di computazione viene calcolato un m.g.u. e viene composto con quelli precedentemente ottenuti.

## 14.4 Il modello computazionale

Il paradigma logico, realizzando l'idea di "computazione come deduzione", ha un modello computazionale sostanzialmente diverso da tutti quelli che abbiamo sin qui visto. Volendo sintetizzare, possiamo individuare le seguenti differenze principali rispetto agli altri paradigmi.

1. Gli unici valori possibili, almeno nel modello puro, sono i termini su una data segnatura.
2. I programmi possono avere una lettura dichiarativa, interamente logica, o una lettura procedurale di tipo più operazionale.
3. La computazione avviene istanziando le variabili che appaiono nei termini (e quindi nei goal) con altri termini, usando il meccanismo dell'unificazione.
4. Il controllo, interamente gestito dalla macchina astratta (salvo alcune annotazioni possibili in PROLOG) è basato sul meccanismo del backtracking automatico.

Nel seguito cercheremo dunque di illustrare il modello computazionale del paradigma logico analizzando questi quattro punti. Discuteremo esplicitamente le differenze fra programmazione logica e PROLOG.

### 14.4.1 L'universo di Herbrand

Nell'ambito della programmazione logica i termini costituiscono un elemento fondamentale. L'insieme di tutti i possibili termini su una data segnatura è detto universo di Herbrand e costituisce il dominio sul quale viene effettuata la computazione da un programma logico. Vi sono alcune particolarità da notare al riguardo.

- Relativamente ai simboli non logici l’alfabeto sul quale sono definiti i programmi non è prefissato ma può variare. Spesso è determinato dai simboli che compaiono nel particolare programma in questione, ma altre volte contiene anche altri simboli.
- Come (parziale) conseguenza del precedente punto, a differenza di quanto avviene nei linguaggi imperativi, nessun significato predefinito è associato ai simboli (non logici) dell’alfabeto. Ad esempio, un programma può usare il simbolo  $+$  per denotare la somma aritmetica mentre un altro con tale simbolo può indicare la concatenazione di stringhe. Fanno eccezione il predicato (binario) d’uguaglianza, alcuni altri predicatori detti “built-in” presenti in PROLOG<sup>17</sup>.
- Come ulteriore conseguenza, nessun sistema di tipi è presente nei linguaggi logici (almeno nel formalismo classico): l’unico tipo presente è quello dei termini con i quali possiamo rappresentare espressioni aritmetiche, liste ecc.

Dal punto di vista teorico, il fatto che non vi siano tipi e che la computazione avvenga sull’universo di Herbrand non è limitativo, anzi permette di esprimere in modo elegante e, tutto sommato, semplice, la semantica formale dei programmi logici. Ad esempio, già con due soli simboli di funzione, 0 (costante zero) e  $s$  (successore, di arietà 1) riusciamo ad esprimere i numeri naturali con i termini  $0, s(0), s(s(0)), s(s(s(0)))$  ecc. Con qualche attenzione possiamo esprimere le normali operazioni aritmetiche in termini di questa rappresentazione a due simboli.

Dal punto di vista pratico, invece, la mancanza di tipi è un problema serio e difatti, sia in PROLOG che in alcuni linguaggi logici più recenti sono stati introdotti alcuni tipi elementari (ad esempio, gli interi con le relative operazioni aritmetiche). I linguaggi di questo paradigma risultano comunque sempre carenti dal punto di vista dei tipi.

#### 14.4.2 Interpretazione dichiarativa e procedurale

Come abbiamo già accennato, una clausola, e quindi un programma logico, può avere due interpretazioni diverse: una *dichiarativa* ed una *procedurale*.

Dal punto di vista *dichiarativo*, una clausola  $H : -A_1, \dots, A_n.$  è una formula che, sostanzialmente, esprime che se  $A_1$  e  $A_2 \dots$  e  $A_n$  sono veri allora è vero anche  $H$ . Una query (o goal) è anch’essa una formula per la quale vogliamo dimostrare che, se opportunamente istanziata, è una conseguenza logica del programma, ossia vale in tutte le interpretazioni nelle quali vale il programma<sup>18</sup>. Questa interpretazione può essere sviluppata, usando gli strumenti della logica (in particolare alcune nozioni elementari di teoria dei modelli) in modo tale da dare un significato ad un programma in termini puramente dichiarativi, senza alcun riferimento ad un processo di calcolo. Per questa interpretazione, per altro interessante, rimandiamo alla letteratura specializzata citata alla fine del capitolo.

<sup>17</sup> Ed anche i predicatori predefiniti nei linguaggi con vincoli, che però noi qui non consideriamo.

<sup>18</sup> Ci accontentiamo qui di un’idea intuitiva di questi concetti. Il lettore interessato può vedere una qualsiasi testo di logica per maggiori dettagli.

L’interpretazione *procedurale*, invece, permette di leggere una clausola

$$H : -A_1, \dots, A_n.$$

come segue: per dimostrare  $H$  devi prima dimostrare  $A_1, \dots, A_n$ , o anche, per calcolare  $H$  devi prima calcolare  $A_1, \dots, A_n$ . In questo senso possiamo vedere un predicato come un nome di procedura: le clausole che lo definiscono costituiscono il suo corpo. In questa interpretazione possiamo leggere un atomo nel corpo di una clausola, o in un goal, come una chiamata di procedura. Un programma logico è dunque un insieme di dichiarazioni e un goal non è altro che l’equivalente del “main” di un programma imperativo, dato che contiene tutte le chiamate di procedura che si vogliono valutare. La virgola nel corpo delle clausole e nei goal, in PROLOG (ma non nella programmazione logica pura), può essere vista come l’analogo del “;” dei linguaggi imperativi.

Precisi teoremi di corrispondenza permettono di riconciliare la visione dichiarativa e quella procedurale, dimostrando che i due approcci sono equivalenti.

Da un punto di vista formale, l’interpretazione procedurale è supportata dalla cosiddetta risoluzione SLD, una regola di derivazione logica che discuteremo nell’Approfondimento 14.1. Ma è anche possibile descrivere l’interpretazione procedurale in modo più informale, usando solo l’analogia che abbiamo appena espresso con le chiamate di procedura ed il passaggio dei parametri. È questo l’approccio che useremo nel seguito.

#### 14.4.3 La chiamata di procedura

Consideriamo per il momento una definizione semplificata di clausola, dove assumiamo che nella testa tutti gli argomenti del predicato siano variabili distinte. Una generica clausola di questo tipo ha dunque la forma

$$p(X_1, \dots, X_n) : -A_1, \dots, A_m.$$

e, come abbiamo anticipato, può essere vista come la dichiarazione della procedura  $p$  con  $n$  parametri formali  $X_1, \dots, X_n$ . Un atomo  $q(t_1, \dots, t_n)$  può essere visto come la chiamata della procedura  $q$  con gli  $n$  parametri attuali  $t_1, \dots, t_n$ . Nella definizione di  $p$ , dunque, il corpo è costituito dall’invocazione delle  $m$  procedure che costituiscono gli atomi  $A_1, \dots, A_m$ .

In accordo a questa visione e analogamente a quanto avviene nei linguaggi imperativi, la valutazione della chiamata  $p(t_1, \dots, t_n)$  causa la valutazione del corpo della procedura, dopo aver effettuato il passaggio dei parametri. Questo avviene con una modalità simile al passaggio per nome, rimpiazzando nel corpo della procedura il parametro formale  $X_i$  con il corrispondente attuale  $t_i$ . Inoltre, dato che le variabili che compaiono nel corpo della procedura sono da considerarsi locali, esse devono essere considerate distinte da tutte le altre variabili. Nei linguaggi a blocchi questo avviene implicitamente, dato che il corpo della procedura è considerato come un blocco con un suo ambiente locale. Qui invece, non essendo presente la nozione di blocco, per evitare possibili conflitti tra i nomi di variabile, supponiamo che, prima di usare una clausola, tutte le variabili

che occorrono in essa siano state ridenominate opportunamente in modo da essere diverse da tutte le altre variabili presenti.

Riassumendo in termini più precisi quanto detto sopra, possiamo dire che la valutazione della chiamata  $p(t_1, \dots, t_n)$ , con la definizione di  $p$  vista poc' anzi, causa la valutazione delle  $m$  chiamate di procedura

$$(A_1, \dots, A_m)\vartheta$$

presenti nel corpo di  $p$ , opportunamente istanziate dalla sostituzione

$$\vartheta = \{X_1/t_1, \dots, X_n/t_n\}$$

che effettua il passaggio dei parametri. Nel caso in cui il corpo della clausola sia vuoto (ossia  $m = 0$ ), la chiamata di procedura termina immediatamente. Altrimenti la computazione prosegue con la valutazione delle nuove chiamate. Usando la terminologia della programmazione logica, questo si esprime dicendo che la valutazione del goal  $p(t_1, \dots, t_n)$  produce il nuovo goal

$$(A_1, \dots, A_m)\{X_1/t_1, \dots, X_n/t_n\}$$

che, a sua volta, dovrà essere valutato. Quando tutte le chiamate generate con questo processo sono state valutate (senza che vi sia stato alcun fallimento) la computazione termina con successo e il risultato finale è costituito dalla sostituzione che associa alle variabili presenti nella chiamata iniziale ( $X_1, \dots, X_n$  nel nostro caso) i valori calcolati nel corso della computazione. Questa sostituzione è detta *risposta calcolata* per il goal iniziale nel programma dato. Vedremo una definizione più precisa di questa nozione più avanti, per ora vediamo un semplice esempio. Consideriamo la procedura `lista_di_2` definita qui sotto ed identica alla procedura `lista_di_27` del Paragrafo 14.1.1, salvo il minor numero di variabili anonime.

```
lista_di_2(Ls) :- Ls = [_,_].
```

La valutazione della chiamata `lista_di_2(Lxs)` causa la valutazione del corpo della clausola, istanziato dalla sostituzione  $\{Ls/Lxs\}$ , e cioè

```
Lxs = [_,_]
```

Questa è una chiamata particolare, perché `=` è un predicato predefinito che, come abbiamo visto nel paragrafo precedente, è interpretato come uguaglianza sintattica sull'universo di Herbrand e operazionalmente corrisponde al processo di unificazione. La chiamata precedente dunque si riduce al tentativo (fatto dall'interprete del linguaggio) di risolvere l'equazione usando il meccanismo dell'unificazione. Nel nostro caso, ovviamente, tale tentativo ha successo e produce l'm.g.u.  $\{Lxs/[_,_]\}$  che è il risultato della computazione: la precedente sostituzione è dunque risposta calcolata per il goal `lista_di_2` nel programma logico costituito dall'unica linea (1) sopra.

**Valutazione di un goal non atomico** Quanto visto nel paragrafo precedente deve essere generalizzato e precisato meglio perché sia chiaro il modello computazionale del paradigma logico. Nel seguito, per conformarci alla terminologia corrente, parleremo di goal atomico e di goal, invece che di chiamata di procedura e di sequenza di chiamate. Rimane valida tuttavia l'analogia con le procedure dei paradigmi convenzionali.

Nel caso in cui si debba valutare un goal non atomico il meccanismo computazionale è analogo a quello visto sopra, salvo che adesso dobbiamo selezionare una delle chiamate possibili usando un'opportuna *regola di selezione*. Mentre nel caso della programmazione logica pura non è specificata alcuna regola, PROLOG adotta la regola che seleziona sempre l'atomo più a sinistra. È comunque possibile dimostrare che le risposte calcolate sono sempre le stesse, indipendentemente da quale sia la regola adottata (si vedano comunque anche gli Esercizi 13 e 14 alla fine del capitolo).

Supponendo, per semplicità, di adottare la regola di PROLOG, possiamo descrivere come segue il processo di valutazione. Sia

$$B_1, \dots, B_k$$

con  $k \geq 1$  il goal da valutare. Distinguiamo i seguenti casi a seconda della forma dell'atomo selezionato  $B_1$ .

1. se  $B_1$  è un'equazione della forma  $s = t$  allora si cerca di calcolarne un m.g.u. (usando l'algoritmo di unificazione). Abbiamo due possibilità:
  - (a) se l'm.g.u. esiste ed è la sostituzione  $\sigma$ , allora il risultato della valutazione è il goal
 
$$(B_2, \dots, B_k)\sigma$$
 ottenuto dal precedente eliminando l'atomo selezionato e applicando l'm.g.u. calcolato. Se  $k = 1$  (e dunque  $(B_2, \dots, B_k)\sigma$  è vuoto), la computazione termina con successo.
  - (b) se l'm.g.u. non esiste (ossia l'equazione non ha soluzioni) allora si ha un fallimento;
2. se invece  $B_1$  ha la forma  $p(t_1, \dots, t_n)$  abbiamo i seguenti due casi:
  - (a) se esiste nel programma una clausola della forma
 
$$p(X_1, \dots, X_n) : -A_1, \dots, A_m.$$
 (che consideriamo ridenominata per evitare cattura di variabili), allora il risultato della valutazione è un nuovo goal
 
$$(A_1, \dots, A_m)\vartheta, B_2, \dots, B_k$$
 dove  $\vartheta = \{X_1/t_1, \dots, X_n/t_n\}$ . Se  $k = 1$  (ossia avevamo un goal atomico) e  $m = 0$  (il corpo della clausola è vuoto) allora la computazione termina con successo;

- (b) se nel programma non esiste alcuna clausola che definisce il predicato  $p$ , allora si ha un fallimento.

Per poter definire con esattezza i risultati delle computazioni (le risposte calcolate) abbiamo bisogno di chiarire alcuni aspetti relativi al controllo, cosa che facciamo nel Paragrafo 14.4.4.

**Teste con termini generici** Sin qui abbiamo supposto che le teste delle clausole contenessero solo variabili distinte. Abbiamo fatto questa scelta per conservare la similarità con le procedure dei linguaggi tradizionali, tuttavia i programmi logici reali usano anche termini generici come argomenti dei prediciati nelle teste, come abbiamo visto nell'esempio del Paragrafo 14.1.1. L'Approfondimento 14.1 fornisce la regola di valutazione per tale caso più generale. Si noti, comunque, che la nostra assunzione non è in alcun modo limitativa (a parte, forse, la convenienza grafica): infatti, come appare evidente da quanto detto nel riquadro e dalla precedente trattazione, una clausola della forma

$$p(t_1, \dots, t_n) : -A_1, \dots, A_m.$$

può essere vista come un'abbreviazione per la clausola

$$p(X_1, \dots, X_n) : -X_1 = t_1, \dots, X_n = t_n, A_1, \dots, A_m.$$

Negli esempi seguenti useremo spesso la notazione con generici termini nelle teste.

#### Approfondimento

14.1

#### 14.4.4 Controllo: il non determinismo

Nella valutazione di un goal abbiamo due gradi di libertà: la selezione dell'atomo da valutare e la scelta della clausola da applicare.

Per il primo abbiamo detto che possiamo fissare una regola di selezione, senza che questo influenzi i risultati finali delle computazioni che terminano con successo.

Per la scelta della clausola invece la cosa è più delicata. Dato che un predicato può essere definito da più clausole e dobbiamo usarne una sola alla volta, potremmo pensare di fissare una qualche regola di scelta delle clausole, analogamente a quanto fatto con la selezione degli atomi. Il seguente esempio mostra però un problema. Consideriamo il seguente programma, che chiamiamo  $P_a$ :

```
p(X) :- p(X).
p(X) :- X=a.
```

1  
2

e supponiamo di scegliere le clausole dall'alto in basso, seguendo l'ordine testuale del programma. È facile vedere che adottando tale regola, la valutazione del goal  $p(Y)$  non termina mai e quindi non abbiamo alcuna risposta calcolata dal programma. Infatti, usando la clausola (1) e la sostituzione  $\{X/Y\}$ , il goal iniziale, dopo un passo di computazione, diviene il goal  $p(X)\{X/Y\}$  che è di nuovo il goal di partenza. A questo goal, secondo la regola di scelta della clausola, dobbiamo applicare di nuovo la clausola (1) e così via. È però evidente che usando la clausola (2) otterremmo immediatamente una computazione che termina, producendo come risposta la sostituzione  $\{Y/a\}$ . Si noti che fissare un altro ordine, ad esempio dal basso verso l'alto, non risolve in generale il problema. Alla luce di questo esempio riconsideriamo con attenzione la regola di valutazione di un goal vista nel precedente paragrafo e, in particolare, soffermiamoci sul punto 2(a), dove sta scritto "se esiste una clausola" senza specificare come questa debba essere scelta. Così facendo abbiamo dunque introdotto nel modello di computazione una forma di non determinismo: nel caso in cui vi siano più clausole per lo stesso predicato dobbiamo sceglierne una in modo non deterministico, senza fissare alcuna regola. Questa forma di non-determinismo è detta "*don't know*", in quanto non sappiamo qual è la clausola "giusta" che ci permette di terminare con successo la computazione. Il modello teorico della programmazione logica mantiene questo non-determinismo, in quanto vengono considerate tutte le possibili scelte della clausole e quindi tutti i possibili risultati delle varie computazioni prodotte in conseguenza di tali scelte. Il risultato della valutazione di un goal  $G$  nel programma  $P$  è dunque un insieme di risposte calcolate, dove ognuna di tali risposte è la sostituzione ottenuta dalla composizione di tutti gli m.g.u. che si incontrano in una specifica computazione (con specifiche scelte di clausole), ristretta alle variabili presenti in  $G$ . Per una definizione più precisa di questa nozione, così come dell'intero processo di valutazione dei goal, si veda l'Approfondimento 14.1.

Tornando al nostro esempio precedente, l'unica risposta calcolata per il goal  $p(Y)$  nel programma  $P_a$  è la sostituzione  $\{Y/a\}$  mentre, sempre per lo stesso programma, il goal  $p(b)$  non ha alcuna risposta calcolata dato che tutte le sue computazioni o terminano con fallimento (quando si usa seconda clausola) oppure non terminano (quando si usa solo la prima clausola).

**Il backtracking in PROLOG** Quando dal modello teorico si passa ad un linguaggio implementato, quale PROLOG, il non-determinismo ad un qualche livello deve essere trasformato in determinismo, dato che le macchine fisiche che usiamo come calcolatori sono deterministiche. Questo può ovviamente essere fatto in vari modi e in linea di principio non causa la perdita di soluzioni: ad esempio, potremmo pensare di far partire  $k$  computazioni parallele quando si abbiano  $k$  possibili clausole per un predicato<sup>19</sup> e quindi considerare i risultati di tutte le possibili computazioni.

<sup>19</sup>Ovviamente, su una macchina con un solo processore, le  $k$  computazioni parallele devono essere opportunamente "schedolate" per poter essere eseguite in modo sequenziale, analogamente a quanto avviene con i processi in un sistema operativo multitasking.

In PROLOG tuttavia, per motivi di semplicità e di efficienza dell'implementazione, viene usata la strategia vista prima: le clausole vengono usate secondo l'ordine testuale con cui appaiono nel programma (dall'alto in basso). Abbiamo visto con il precedente esempio che questa strategia è *incompleta*, dato che non permette di trovare tutte le possibili risposte calcolate. Questo limite tuttavia è aggirabile dal programmatore che, conoscendo questa caratteristica di PROLOG, può ordinare le clausole del programma nel modo più conveniente (tipicamente mettendo prima quelle relative ai casi terminali e poi quelle induttive). Si noti però che questo, almeno in linea di principio, elimina parte della dichiaratività del linguaggio, in quanto al programmatore è richiesta la specifica di un aspetto di controllo.

Oltre alle computazioni infinite, vi è un secondo aspetto, più importante, da considerare adottando il modello deterministico di PROLOG e riguarda la gestione dei fallimenti. Vediamo prima un esempio. Consideriamo il programma *Pb*

```
p(X) :- X=f(a).  
p(X) :- X=g(a).
```

e consideriamo la valutazione del goal  $p(g(Y))$ . Secondo la strategia di PROLOG viene scelta la clausola (1), che (usando la sostituzione  $\{X/g(Y)\}$ ) da luogo al nuovo goal  $g(Y) = f(a)$ . Questo fallisce, dato che i due termini dell'equazione non sono unificabili. Tuttavia, dato che vi è ancora una clausola da usare, non sarebbe accettabile terminare la computazione riportando un fallimento. Viene dunque fatto un "backtracking" tornando alla scelta della clausola per  $p(g(Y))$  e quindi si prosegue la computazione provando la clausola (2). In questo modo si arriva al successo con risposta calcolata  $\{Y/a\}$ .

In generale dunque, quando si arriva ad un fallimento, la macchina astratta PROLOG fa "backtracking" fino al precedente punto di scelta nel quale vi siano altre possibilità, ossia altre clausole da provare. In questo processo di backtracking gli eventuali legami che erano stati calcolati nella computazione precedente devono essere disfatti. Giunti al punto di scelta, viene provata una nuova clausola e la computazione prosegue nel modo visto. Se al precedente punto di scelta non vi sono altre possibilità, si risale al punto di scelta ancora precedente e se non ve ne sono la computazione termina con un fallimento. Si noti che tutto questo è gestito direttamente dalla macchina astratta PROLOG ed è completamente invisibile al programmatore (salvo l'uso di costrutti particolari, quali il cut che introdurremo nel Paragrafo 14.5.1).

È anche facile rendersi conto come questo procedere per tentativi, che sostanzialmente corrisponde ad una ricerca in profondità in un albero che rappresenta tutte le possibili computazioni, può essere molto pesante dal punto di vista computazionale. La soluzione del problema visto nel Paragrafo 14.1.1, ad esempio, richiede un uso esteso del backtracking ed è abbastanza dispendiosa dal punto di vista del tempo di calcolo.

Vediamo un altro esempio. Consideriamo il programma *Pc*

```
p(X) :- X=f(Y), q(X).  
p(X) :- X=g(Y), q(X).  
q(X) :- X=g(a).
```

$q(X) :- X=g(b)$ .

ed analizziamo la valutazione del goal  $p(Z)$ . Usando la clausola (1) otteniamo il goal  $Z=f(Y), q(Z)$ ; la valutazione dell'equazione produce l'm.g.u.  $\{Z/f(Y)\}$  e rimane il goal  $q(f(Y))$ . Usando la clausola (3) otteniamo il goal  $f(Y) = g(a)$ , che fallisce. A questo punto dobbiamo tornare<sup>20</sup> al precedente punto di scelta, ossia alla scelta della clausola per il predicato  $q$ . In questo caso non vi sono legami da "disfare" e dunque proviamo la clausola (4), ottenendo così il goal  $f(Y) = g(b)$  che fallisce anch'esso. Torniamo di nuovo al punto di scelta per  $q$ , vediamo che non vi sono altre possibili clausole e dunque torniamo al precedente punto di scelta che è quello per il predicato  $p$ . Facendo questo dobbiamo disfare il legame  $\{Z/f(Y)\}$  che era stato calcolato dalla clausola (1) e dunque ritorniamo alla situazione iniziale, dove la variabile  $Z$  non è istanziata. Usando la clausola (2) otteniamo il goal  $Z=g(Y), q(Z)$  e quindi, dalla valutazione dell'equazione, otteniamo l'm.g.u.  $\{Z/g(Y)\}$  ed il nuovo goal  $q(g(Y))$ . A questo punto, usando la clausola (3) otteniamo il goal  $g(Y) = g(a)$  che ha successo e produce l'm.g.u.  $\{Y/a\}$ . Dato che non rimangono altri goal da valutare, la computazione termina con successo. Il risultato della computazione è prodotto componendo gli m.g.u. calcolati  $\{Z/g(Y)\}\{Y/a\}$  e restringendo la sostituzione  $\{Z/g(a), Y/a\}$  ottenuta da tale composizione all'unica variabile presente nel goal iniziale: otteniamo così la risposta calcolata  $\{Z/g(a)\}$ . Si noti che vi è un'altra risposta calcolata  $\{Z/g(b)\}$ , che possiamo ottenere usando la clausola (4) invece che la (3). Nelle implementazioni PROLOG si possono ottenere le risposte successive alla prima usando il comando ";". Infine, il lettore può facilmente verificare che il goal  $p(g(c))$  nel programma *Pc* termina con un fallimento, ottenuto dopo aver provato tutte e quattro le combinazioni di clausole possibili.

#### 14.4.5 Alcuni esempi

In questo paragrafo faremo riferimento al linguaggio PROLOG, del quale seguiremo anche la sintassi. La notazione  $[h \mid t]$  è usata per indicare la lista che ha come testa  $h$  e come coda  $t$ . Ricordiamo che la testa è il primo elemento della lista, mentre la coda è la lista costituita dai rimanenti elementi, una volta tolto il primo. La lista vuota è denotata da  $[]$  mentre la scrittura  $[a, b, c]$  è un'abbreviazione per  $[a \mid [b \mid [c \mid []]]]$  (la lista costituita dai tre elementi  $a$ ,  $b$  e  $c$ ). Si noti che in PROLOG, così come nella programmazione logica pura, non esiste il tipo lista, per cui il simbolo di funzione binaria  $[ \mid ]$  può essere usato anche per termini che non sono liste: ad esempio, possiamo anche scrivere  $[a \mid f(a)]$  che non è una lista (perché  $f(a)$  non è una lista).

Come primo esempio consideriamo il seguente programma MEMBER che verifica se un elemento appartiene ad una data lista:

```
member(X, [X | Xs]).
```

<sup>1</sup>

<sup>20</sup>Usiamo il plurale per convenzione antropomorfa: tutto questo è fatto, ovviamente, dalla macchina astratta PROLOG.

```
member(X, [Y | Xs]) :- member(X, Xs).
```

La lettura dichiarativa del precedente programma è immediata: la clausola (1) costituisce il caso terminale, in cui l'elemento che stiamo cercando (il primo argomento del predicato `member`) è la testa della lista che abbiamo (il secondo argomento di `member`). La clausola (2) fornisce invece il caso induttivo e ci dice che `X` è un elemento della lista `[Y | Xs]` se è un elemento della lista `Xs`.

Così formulato, il programma MEMBER è simile a quello che possiamo scrivere in un qualsiasi linguaggio che supporti la ricorsione. Tuttavia notiamo che, a differenza di quanto avviene nei paradigmi funzionale e imperativo, il precedente programma può essere usato in vari modi diversi.

Il modo più convenzionale è quello di usarlo come test: in un sistema PROLOG, una volta immesso il precedente programma, abbiamo

```
?- member(pippo, [pluto, pippo, ciccio]).  
Yes
```

dove `?-` è il prompt della macchina astratta, al quale abbiamo fatto seguire il goal di cui si chiede la valutazione. La linea successiva riporta la risposta dell'interprete. In questo caso abbiamo una semplice risposta "booleana", che esprime l'esistenza di una computazione di successo per il nostro goal. Tuttavia, come sappiamo, possiamo anche usare il programma per calcolare. Ad esempio possiamo chiedere la valutazione di

```
?- member(X, [pluto, pippo, ciccio]).  
X = pluto
```

La macchina astratta restituisce `{X/pluto}` come una risposta calcolata. Possiamo anche ottenere le riposte successive usando il comando `;`. Quando non vi sono più risposte il sistema risponde "no".

Infine, anche se si tratta di un uso poco naturale, possiamo usare il primo argomento per istanziare la lista nel secondo argomento. Ad esempio abbiamo

```
?- member(pluto, [X, pippo, ciccio]).  
X = pluto
```

Questa possibilità di usare gli stessi argomenti come input o output, a seconda di come sono istanziati, è unica del paradigma logico ed è dovuta alla presenza del meccanismo di unificazione nel modello di calcolo.

Possiamo chiarire ulteriormente questo punto considerando il seguente programma APPEND che permette di concatenare due liste.

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

Anche in questo caso la lettura dichiarativa è immediata: se la prima lista è vuota, allora il risultato è costituito dalla seconda lista (clausola (1)). Altrimenti (induttivamente) se `Zs` è il risultato della concatenazione di `Xs` e `Ys`, il risultato della concatenazione di `[X|Xs]` e `Ys` è ottenuto aggiungendo `X` in testa alla lista `Zs`, come indicato in `[X|Zs]`.

L'uso normale del precedente programma è quello illustrato dal seguente goal

2

```
?- append([pluto, pippo], [ciccio, qui], Zs).  
Zs = [pluto, pippo, ciccio, qui]
```

Tuttavia, possiamo usare APPEND anche per sapere come possiamo suddividere una lista in due sottoliste, cosa che non è possibile con una versione funzionale o imperativa del programma:

```
?- append(Xs, Ys, [pluto, pippo]).  
Xs = []  
Ys = [pluto, pippo];  
Xs = [pluto]  
Ys = [pippo];  
Xs = [pluto, pippo]  
Ys = [];  
no
```

dove, come ricordato poc'anzi, il comando `;` permette di ottenere nuove soluzioni.

Come terzo esempio diamo la definizione del predicato sublist, che abbiamo usato nel Paragrafo 14.1.1. Se `Xs` è una sottolista di `Ys` allora esistono altre due liste (possibilmente vuote) `As` e `Bs` tali che `Ys` è la concatenazione di `As`, `Xs` e `Bs`. Osserviamo che questo significa che `Xs` è il suffisso di un prefisso di `Ys` per cui, usando APPEND, possiamo definire sublist come segue:

```
sublist(Xs, Ys) :- append(As, XsBs, Ys), append(Xs, Bs, XsBs).
```

Il precedente programma, unitamente al programma APPEND e al programma SEQUENZA del Paragrafo 14.1.1, permette di risolvere il problema enunciato nel Paragrafo 14.1.1 stesso. Si noti la concisione e semplicità del programma risultante. Ovviamente altre definizioni di sublist sono possibili (si veda l'esercizio 8).

Come ultimo esempio consideriamo un programma che risolve il classico problema delle torri di Hanoi. Si ha una torre (in termini informatici diremmo una pila) costituita da  $n$  dischi forati (di diametri diversi), disposti su di un'asta in ordine di diametro decrescente e si hanno altre due aste vuote. Il problema consiste nello spostare la torre ad un'altra asta, ricreando l'ordinamento iniziale dei dischi e rispettando le seguenti regole: i dischi possono essere spostati solo da un'asta ad un'altra asta; non si può spostare più di un disco alla volta da un'asta all'altra e si deve prelevare sempre il disco in cima ad un'asta; un disco non può mai essere posto sopra un altro disco di diametro più piccolo.

Secondo la leggenda, questo problema fu assegnato dalla Divinità ai monaci di un monastero nei pressi di Hanoi, con tre aste e 64 dischi d'oro. La soluzione del problema avrebbe segnato la fine del mondo. Dato che la soluzione ottima richiede un tempo esponenziale nel numero dei dischi, anche se la leggenda si dovesse avverare possiamo stare tranquilli ancora per un bel pezzo:  $2^{64}$  è un numero sufficientemente grande.

Il seguente programma risolve il problema per un generico numero di dischi `N` e tre aste che chiamiamo A, B e C. Usiamo la codifica dei numeri naturali in termini di 0 e di suoi successori, già vista in precedenza.

```
hanoi(s(0), A, B, C [sposta(A, B)]).
```

1

2

```
hanoi(s(N), A, B, C, Mosse) :-  
    hanoi(N, A, C, B, Mosse1),  
    hanoi(N, C, B, A, Mosse2),  
    append(Mosse1, [sposta(A, B) | Mosse2], Mosse).
```

La chiamata `hanoi(n, A, B, C, Mosse)`, dove `n` è un termine che rappresenta un numero naturale, risolve il problema di spostare la torre dei dischi da `A` a `B` usando `C` come asta ausiliaria. La soluzione è contenuta nella variabile `Mosse` che, al termine della computazione, è istanziata alla lista contenente le mosse che portano alla soluzione. Ogni mossa è rappresentata da un termine della forma `sposta(X, Y)` per indicare lo spostamento del disco in cima all'asta `X` all'asta `Y`.

Questo programma, di semplicità sorprendente, mostra appieno la potenza della programmazione logica ed anche del ragionamento ricorsivo. Al solito, vediamone la lettura dichiarativa. La prima clausola è chiara: se abbiamo un solo disco, tutto quello che dobbiamo fare è spostarlo da `A` a `B`. Anche la lettura della seconda clausola è abbastanza intuitiva. Se `Mosse1` è la lista di mosse che risolve il problema di spostare una torre di `N` dischi da `A` a `C` usando `B` come asta ausiliaria, e `Mosse2` è la lista di mosse per spostare la torre di `N` dischi da `C` a `B` usando `A` come asta ausiliaria, per risolvere il nostro problema nel caso di `N+1` (ossia `s(N)`) dischi evidentemente dovremo fare quanto segue: prima eseguire tutte le mosse in `Mosse1`, quindi spostare da `A` a `B` (con la mossa `sposta(A, B)`) l'`N+1`-esimo disco di `A` che, essendo quello più grande, dovrà essere l'ultimo (in basso) anche in `B`; infine eseguire tutte le mosse in `Mosse2`. Questo è esattamente quello che viene realizzato dal predicato `append` che dunque fornirà nella variabile `Mosse` la soluzione del nostro problema.

## 14.5 Estensioni

Sin qui abbiamo visto la programmazione logica pura ed alcuni aspetti, molto parziali, di PROLOG. In quest'ultimo paragrafo accenniamo brevemente ad alcune delle numerose estensioni del formalismo puro. Ognuna di esse richiederebbe un intero capitolo per cui non possiamo che essere molto superficiali nella trattazione. Nella nota bibliografica sono indicati alcuni riferimenti per chi volesse saperne di più.

### 14.5.1 PROLOG

PROLOG è un linguaggio molto più ricco di quello che potrebbe apparire da quanto detto sin qui. Ricordiamo che, come già visto, questo linguaggio si differenzia dalla programmazione logica per l'adozione di precise regole per la selezione dell'atomo da riscrivere (da sinistra a destra) e per la scelta della clausola da usare (dall'alto in basso, seguendo il testo del programma).

Oltre a queste vi sono altre importanti differenze con il formalismo teorico. Ne elenchiamo alcune, senza la pretesa di essere esaustivi.

**Aritmetica** Un linguaggio reale non può permettersi il lusso di usare un'aritmetica completamente simbolica, nella quale i numeri naturali sono rappresentati come successori dello zero e non vi sono operatori predefiniti.

In PROLOG esistono dunque gli interi ed i reali (in virgola mobile) come strutture dati predefinite (built-in) e vari operatori per operare su di essi. Questi includono:

- gli usuali operatori aritmetici `+, -, *, //` (divisione intera);
- operatori aritmetici di confronto quali `<, <=, >, >=, =:=` (uguale), `=\=` (diverso);
- un operatore di valutazione detto `is`.

A differenza degli altri simboli (di funzione e di predicato) usati in PROLOG, questi usano la notazione infissa, per comodità dell'utente. Vi sono tuttavia numerosi aspetti delicati che richiedono una conoscenza approfondita per evitare errori banali, nei quali il programmatore abituato ai linguaggi imperativi può facilmente incorrere.

Innanzitutto gli operatori di confronto richiedono sempre che gli operandi siano espressioni aritmetiche ground (senza variabili). Così, mentre abbiamo<sup>21</sup>

```
?- 3*2 =:= 1+5  
Yes  
?- 4 > 5+2  
no
```

se usiamo termini che non sono espressioni aritmetiche o che contengono delle variabili abbiamo degli errori:

```
?- 3 > a  
error in arithmetic expression: a is not a number  
?- X =:= 3+5  
instantiation fault.
```

L'ultimo esempio disturba particolarmente: quello che vorremo esprimere è che la valutazione dell'espressione aritmetica `3+5` produce il valore 8 che viene quindi legato alla variabile `x`. Questo non lo possiamo fare usando il `=:=`, come visto sopra, e neanche usando l'uguaglianza fra termini considerata nei paragrafi precedenti: se scriviamo `X=3+5` infatti l'effetto è quello di legare `x` al termine `3+5` (invece che al valore 8). Per ovviare a questo problema è stato introdotto il valutatore di espressioni `is` che permette di ottenere l'effetto desiderato: `s is t` infatti *unifica* `s` con il *valore* dell'espressione aritmetica ground `t` (se `t` non è un'espressione aritmetica ground abbiamo un errore). I seguenti sono alcuni esempi di utilizzo di `is`

```
?- X is 3+5;  
X = 8  
?- 8 is 3 + 5
```

<sup>21</sup> Come detto prima, la linea con `?-` riporta la query che vogliamo valutare e quella successiva è la risposta dell'interprete PROLOG.

```

Yes
?- 6 is 3 * 3
no
?- X is Y* 2
error in arithmetic expression: Y*2 is not a number

```

L'uso dell'`is`, necessario per poter valutare un'espressione, rende l'uso dell'aritmetica in PROLOG abbastanza macchinoso. Ad esempio, dato il programma seguente

```

valuta(0, 0).
valuta(s(X), Val+1) :- valuta(X, Val)

```

il goal `valuta(s(s(s(0))), X)` non calcola per `X` il valore 3 come forse ci saremmo aspettati, ma il termine `0+1+1+1` (si veda anche l'esercizio 9 alla fine del capitolo).

**Cut** PROLOG offre vari costrutti per interagire con l'interprete della macchina astratta in modo tale da modificare il normale flusso del controllo. Fra questi, uno dei più significativi (e dei più discussi) è il *cut*. Si tratta di un predicato senza argomenti, indicato con un punto esclamativo, che permette di eliminare parte delle possibili alternative nel processo di valutazione, con lo scopo di aumentare l'efficienza dell'esecuzione. È usato quando siamo sicuri che, al verificarsi di una condizione, le altre clausole del programma non serviranno. Ad esempio, nel seguente programma che calcola il minimo tra due valori, se una condizione di confronto è vera, l'altra è necessariamente falsa, quindi possiamo usare il `!` per esprimere il fatto che una volta che si è usata la prima clausola non occorre provare ad usare la seconda:

```

minimo(X, Y, X) :- X < Y, !.
minimo(X, Y, Y) :- X > Y.

```

Oppure, se ci interessa solo verificare se un valore compare almeno una volta in una lista, possiamo usare la seguente modifica del programma MEMBER già visto:

```

member(X, [X | Xs]) :- !.
member(X, [Y | Xs]) :- member(X, Xs).

```

In generale il significato del cut è il seguente. Se abbiamo  $n$  clausole

```

p(S1) :- A1.
...
p(Sk) :- B, !, C.
...
p(Sn) :- An.

```

che definiscono il predicato `p` e nella valutazione del goal `p(t)` ci troviamo ad usare la clausola  $k$ -esima della lista, abbiamo i seguenti due casi:

1. se la valutazione di  $B^{22}$  fallisce, allora si prosegue provando la clausola  $k + 1$ -esima;
2. se invece la valutazione di  $B$  ha successo allora viene valutato il `!`, che ha successo, e si prosegue con `C`. In caso di backtracking, tuttavia, tutti i modi alternativi per calcolare  $B$  sono scartati, così come sono eliminate tutte le alternative date dalle clausole dalla  $k$ -esima alla  $n$ -esima per valutare  $p(t)$ .

Non c'è bisogno di dire che il `cut`, oltre a non essere di facile utilizzo, elimina buona parte della dichiaratività di un programma.

**Disgiunzione** Se vogliamo esprimere la disgiunzione di due goal  $G_1$  e  $G_2$ , ossia il fatto che è sufficiente che almeno uno dei due abbia successo, possiamo usare due clausole della forma

```

p(X) :- G1.
p(X) :- G2.

```

ed il goal `p(X)`. Si può ottenere lo stesso effetto scrivendo  $G_1;G_2$ , dove abbiamo usato il predicato predefinito (o built-in) `;` che rappresenta, appunto, la disgiunzione.

**If-then-else** Il tradizionale costrutto dei linguaggi imperativi

```

if B then C1 else C2

```

è offerto in PROLOG come un built-in dalla sintassi `B ->C1;C2`. L'`if then else` si può realizzare usando il `cut` come segue

```

if_then_else(B, C1, C2) :- B, !, C1.
if_then_else(B, C1, C2) :- C2.

```

ed in effetti questa è la definizione interna del costrutto `B ->C1;C2`.

**Negazione** Sin qui abbiamo visto che nel corpo di una clausola si possono usare solo formule atomiche "positive", ossia non negate<sup>23</sup>. Spesso tuttavia può essere utile usare anche formule atomiche negate. Ad esempio, consideriamo il seguente programma Voli

```

volo_diretto(bologna, parigi).
volo_diretto(bologna, amsterdam).
volo_diretto(parigi, bombay).
volo_diretto(amsterdam, mosca).
volo(X, Y) :- volo_diretto(X, Y).
volo(X, Y) :- volo_diretto(X, Z), volo(Z, Y).

```

<sup>22</sup>Qui  $B$ , così come  $A_i$  e  $C$ , è considerato come un generico goal, non necessariamente atomico.

<sup>23</sup>Qui facciamo riferimento, ovviamente, alla notazione  $H : -A_1, \dots, A_n$ . Se invece consideriamo una clausola come disgiunzione di letterali gli atomi nel corpo sono negati, dato che la rappresentazione precedente equivale a  $\neg A_1 \vee \neg A_2 \vee \dots \vee \neg A_n \vee H$ .

dove abbiamo una serie di fatti che definiscono il predicato `volo_diretto` che indica l'esistenza di una connessione senza scali fra due destinazioni. Quindi abbiamo il predicato `volo` che definisce un volo, anche con scali intermedi, fra due località: questo sarà o un volo diretto (clausola 5) oppure un volo diretto per uno scalo diverso da quello di destinazione, seguito da un volo da tale scalo alla destinazione (clausola 6). Con tale programma possiamo verificare l'esistenza di un volo, oppure chiedere quali sono le destinazioni raggiungibili a partire da un dato aeroporto:

```
?- volo(bologna, bombay)
Yes
?- volo_diretto(bologna, mosca)
no
?- volo(bologna, X)
X = parigi
```

Tuttavia, non riusciamo ad esprimere il fatto che esiste solo un volo non diretto, perché non sappiamo come esprimere il fatto che *non* esiste un volo diretto. Per fare questo abbiamo bisogno della negazione. Se scriviamo

```
volo_indiretto(X, Y) :- volo(X, Y), not volo_diretto(X, Y).
```

intendiamo dire esiste un volo indiretto fra `X` e `Y` se esiste un volo fra le due località e non esiste (limitatamente ai fatti nel nostro programma) un volo diretto. Con tale definizione abbiamo

```
?- volo_indiretto(bologna, bombay)
Yes
?- volo_indiretto(bologna, parigi)
no
```

Questi risultati sono spiegabili come segue. L'interprete PROLOG valuta un goal quale `not G` provando a valutare il goal `non negato G`: se la valutazione di questo goal termina (dopo gli eventuali backtracking) con un fallimento allora il goal `not G` ha successo. Se invece il goal `G` ha una computazione che termina con successo allora il goal `not G` fallisce. Infine, se la valutazione di `G` non termina, anche quella di `not G` non termina. Questo tipo di negazione è detto "negazione come fallimento"<sup>24</sup> in quanto, appunto, interpreta la negazione di un goal in termini del fallimento del goal non negato. Si noti che, a causa delle computazioni infinite<sup>25</sup>, questo tipo di negazione è diverso dalla negazione classica della logica, dato che il *non successo* di `G` non equivale al successo della versione negata `not G`. Ad esempio, né il goal `p` né il goal `not p` hanno successo nel programma `p :- p`.

<sup>24</sup> A voler essere precisi, la negazione come fallimento, così come definita nel modello teorico della programmazione logica, ha un comportamento leggermente diverso da quella qui descritta a causa dell'incompletezza dell'interprete PROLOG.

<sup>25</sup> Ed anche dei goal non ground, anche se qui non affrontiamo il problema.

### 14.5.2 Programmazione logica e basi di dati

Il precedente programma `Voli` mostra indirettamente una possibile applicazione della programmazione logica nel contesto delle basi di dati.

Un insieme di clausole unitarie, quali quelle che definiscono il predicato `volo_diretto`, è infatti, a tutti gli effetti, la definizione esplicita (o estensionale) della relazione individuata da tale predicato. In questo senso, tale insieme di clausole unitarie può essere visto come l'analogo di una relazione nel modello relazionale dei dati. Per esprimere un'interrogazione, mentre nel modello relazionale useremmo l'algebra relazionale con gli usuali operatori di selezione, proiezione, join ecc., oppure, più convenientemente, un linguaggio di manipolazione dei dati quale SQL (nella sua componente DML), nel paradigma logico possiamo usare l'usuale meccanismo di computazione già visto. Ad esempio, per sapere se esiste un volo diretto che arriva a Parigi possiamo usare la query `volo_diretto(X, parigi)`. I prediciati definiti da clausole non unitarie, come `volo`, definiscono delle relazioni in modo implicito (o intensionale): essi infatti permettono di calcolare nuove relazioni, che non sono memorizzate esplicitamente, usando il meccanismo di inferenza già visto. Usando la terminologia delle basi di dati diremo che questi prediciati definiscono delle "viste" (o relazioni virtuali).

Si noti, inoltre, come nel programma `Voli` non si siano usati simboli di funzione: in effetti, se vogliamo solo manipolare relazioni, come facciamo in algebra relazionale, i simboli di funzione non servono.

Queste considerazioni sono state cristallizzate nella definizione di *Datalog*, un linguaggio logico per basi di dati. Nella sua forma più semplice è una versione semplificata della programmazione logica nella quale:

1. non vi sono simboli di funzione;
2. si distinguono prediciati estensionali, prediciati intensionali e prediciati di confronto;
3. i prediciati estensionali non possono comparire nella testa di clausole con corpo non vuoto;
4. se una variabile compare nella testa, allora compare anche nel corpo di una clausola;
5. un predicato di confronto può comparire solo nel corpo di una clausola; le variabili che compaiono in un predicato di confronto devono comparire anche in un altro atomo del corpo della stessa clausola.

Le ultime due condizioni hanno a che fare con la specifica regola di valutazione che Datalog adotta<sup>26</sup> e qui saranno ignorate. La distinzione fra prediciati estensionali e intensionali corrisponde all'idea intuitiva già discussa: i prediciati estensionali, come risulta dalla condizione (3), possono essere definiti solo da fatti.

<sup>26</sup> Datalog adotta un meccanismo di valutazione dei goal di tipo bottom-up, diverso da quello top-down che abbiamo visto per la programmazione logica. I risultati ottenuti sono comunque gli stessi.

Il precedente programma Voli può dunque essere considerato a tutti gli effetti come un programma Datalog.

Il lettore che conosca SQL o l'algebra relazionale non avrà difficoltà a comprendere come la presenza della ricorsione renda il potere espressivo di Datalog superiore a quello di questi formalismi. Il solito programma Voli fornisce un esempio in questo senso: la query `volo(bologna, X)` permette di trovare tutte le destinazioni raggiungibili da Bologna con un numero arbitrario di scali intermedi. Una tale query non è esprimibile in algebra relazionale o SQL (nella sua versione iniziale) in quanto il numero di join che dovremmo effettuare dipende dal numero di scali intermedi e in generale, non sapendo quanti questi siano (non facciamo ipotesi su come sono fatte le relazioni), non è definibile a priori.

### 14.5.3 Programmazione logica con vincoli

Per concludere accenniamo brevemente alla programmazione logica con vincoli, o CLP (*Constraint Logic Programming*), un'estensione della programmazione logica che unisce al modello computazionale già visto meccanismi di soluzione di vincoli, anche molto sofisticati. Il paradigma risultante è di sicuro interesse per le applicazioni pratiche ed è oggi usato in vari campi applicativi.

Un vincolo (*constraint* in inglese) non è altro che una particolare formula della logica del prim'ordine (normalmente si tratta di una congiunzione di formule atomiche) che usa solo predici di significato predefinito. Un esempio di vincolo lo abbiamo già incontrato: se scriviamo in un programma logico `X=t` o anche `X=t, Y = f(a)` abbiamo usato dei vincoli, dove il simbolo di predicato `=`, come sappiamo, è interpretato come uguaglianza sintattica sull'universo di Herbrand<sup>27</sup> mentre la virgola indica la congiunzione logica.

L'idea della programmazione logica con vincoli è quella di rimpiazzare l'universo di Herbrand con un diverso dominio di computazione, simbolico ma anche numerico in alcuni casi, adatto alla specifica classe di applicazioni che interessa. I vincoli, invece che relazioni fra termini ground, definiscono relazioni sui valori del nuovo dominio considerato. Corrispondentemente, il meccanismo di calcolo di base non sarà più la soluzione di equazioni fra termini (mediante unificazione), ma sarà costituito da uno specifico risolutore di vincoli, ossia da un opportuno algoritmo per determinare la soluzione dei vincoli considerati. Ad esempio, potremmo considerare come dominio della computazione i numeri reali, come vincoli congiunzioni di equazioni lineari, e come risolutore potremmo usare il metodo di Gauss-Jordan.

Il vantaggio principale di questo approccio è evidente: possiamo integrare nel paradigma logico (in modo, fra l'altro, semanticamente molto pulito) meccanismi di calcolo molto potenti sviluppati anche in altri contesti (quali la programmazione lineare, la ricerca operativa ecc.). Domini particolarmente interessanti per

<sup>27</sup>Ricordiamo che questo è l'insieme dei termini ground.

le applicazioni pratiche sono i già menzionati numeri reali<sup>28</sup> e, soprattutto, i domini finiti, nei quali le variabili possono assumere un numero finito di valori. In quest'ultimo caso si parla anche di problemi soddisfabilità di vincoli o CSP (*Constraint Satisfaction Problem*) ed esistono numerosi algoritmi specifici per la soluzione di questo tipo di problemi.

Per avere un'idea delle possibilità della programmazione logica con vincoli si consideri il problema di definire l'ammontare delle rate di un mutuo. Le quantità coinvolte in questo problema sono le seguenti: `Fin` è il finanziamento richiesto, ossia la somma iniziale presa in prestito; `NumR` è il numero delle rate; `Int` è il tasso d'interesse; `Rata` è l'importo della singola rata; infine `Deb` è il debito rimanente. La relazione fra queste variabili è intuitivamente la seguente. Nel caso in cui non vi sia alcuna rata rimborsata (ossia `NumR = 0`) evidentemente abbiamo che il debito è eguale al finanziamento iniziale:

$$\text{Deb} = \text{Fin}.$$

Nel caso del pagamento di una singola rata invece abbiamo la seguente relazione:

$$\text{Deb} = \text{Fin} + \text{Fin} * \text{Int} - \text{Rata}$$

ossia il debito rimanente è l'importo iniziale al quale abbiamo aggiunto gli interessi passivi maturati e sottratto quanto rimborsato con una rata. Nel caso di due rate la cosa è più complicata perché gli interessi vanno calcolati sul debito rimanente e non sull'importo iniziale. Il debito rimanente, cioè, diventa il nuovo valore del finanziamento. Usando le variabili `NuFin1` e `NuFin2` per indicare tale nuovo valore del finanziamento, in questo caso abbiamo dunque le relazioni

$$\begin{aligned}\text{NuFin1} &= \text{Fin} + \text{Fin} * \text{Int} - \text{Rata} \\ \text{NuFin2} &= \text{NuFin1} + \text{Fin} * \text{Int} - \text{Rata} \\ \text{Deb} &= \text{NuFin2}\end{aligned}$$

Nel caso generale, al solito, possiamo ragionare ricorsivamente. Abbiamo dunque il seguente programma con vincoli che chiamiamo MUTUO

```
mutuo(Fin, NumR, Int, Rata, Deb) :-  
    NumR = 0,  
    Deb = Fin.  
  
mutuo(Fin, NumR, Int, Rata, Deb) :-  
    NumR >= 1,  
    NuFin = Fin+Fin*Int-Rata,  
    NuNumR = NumR-1,  
    mutuo(NuFin, NuNumR, Int, Rata, Deb).
```

la cui comprensione, alla luce di quanto detto sopra, non dovrebbe porre difficoltà.

Un tale programma può essere usato in molti modi, analogamente a quanto avviene con i programmi logici. Ad esempio, possiamo usarlo per sapere qual è il debito rimanente, avendo preso un finanziamento di 1000 euro e avendo pagato

<sup>28</sup>Ovviamente, come sempre quando si tratta di calcolatori, quello che viene effettivamente rappresentato è un sottoinsieme dei reali.

10 rate di 150 euro l'una, con un tasso d'interesse del 10%. La query per risolvere questo problema è

```
mutuo(1000, 10, 10/100, 150, Deb)
```

che se valutata produce il risultato  $Deb = 203.13$ .

Possiamo anche usare lo stesso programma in "senso inverso", per così dire, ossia per sapere qual è il finanziamento che avevamo chiesto, sapendo che abbiamo pagato le stesse 10 rate di 150 euro l'una di prima, sempre con un tasso d'interesse del 10%, ma con debito residuo 0. Il goal in questo caso è

```
mutuo(Fin, 10, 10/100, 150, 0)
```

che dà il risultato  $Fin = 921.685$ .

Infine potremmo anche lasciare più di una variabile nel goal, ad esempio per sapere la relazione che esiste fra il finanziamento, la rata e il debito, sapendo che paghiamo 10 rate con tasso d'interesse del 10%. Possiamo quindi formulare la query

```
mutuo(Fin, 10, 10/100, Rata, Deb)
```

che (con opportune ipotesi sul risolutore<sup>29</sup>) produce il risultato

```
Fin = 0.3855 * Deb + 6.1446 * Rata.
```

ossia, cosa unica nei vari paradigmi sin qui visti, permette di ottenere, come risultato, una relazione su domini numerici.

## 14.6 Pregi e difetti del paradigma logico

I pochi esempi mostrati in questo capitolo dovrebbero essere comunque sufficienti per mostrare che i linguaggi logici richiedono uno stile di programmazione significativamente diverso da quelle dei linguaggi tradizionali ed hanno anche un diverso potere espressivo. Ovviamente, dato che tutti i linguaggi (o quasi) che sono stati nominati in questo testo costituiscono dei formalismi Turing completi, non è che PROLOG, o CLP, o un qualsiasi altro linguaggio logico permetta di "calcolare più cose" di un più comune linguaggio imperativo. Il potere espressivo al quale facciamo qui riferimento è di tipo pragmatico, come chiariamo meglio in questo paragrafo

Innanzitutto giova ripetere che linguaggi logici, analogamente ai linguaggi funzionali, permettono di esprimere in modo "puramente dichiarativo", e quindi estremamente semplice e compatto, soluzioni di problemi anche molto complessi. Nel nostro caso specifico questi aspetti sono ulteriormente rafforzati da tre caratteristiche uniche del paradigma logico: (i) la possibilità di usare in più modi diversi

<sup>29</sup>Il programma MUTUO è scritto usando la sintassi di CLP(R). L'adattamento ad altri sistemi CLP, anche se definiti sui reali, può richiedere modifiche abbastanza sostanziali anche se l'idea di fondo rimane la stessa. In particolare, la possibilità di ottenere i risultati discussi dipende molto dalla potenza del risolutore di vincoli adottato dal linguaggio CLP. I risultati qui riportati sono derivati da [63].

uno stesso programma, trasformando argomenti di input in argomenti di output e viceversa; (ii) la possibilità di ottenere come risultato una relazione complessa fra variabili, che esprime, in modo implicito, un'infinità di soluzioni (date da tutti i valori delle variabili che soddisfano la relazione); (iii) la possibilità di leggere un programma direttamente come una formula logica.

La prima caratteristica deriva dal fatto che il meccanismo di computazione di base, l'unificazione, è intrinsecamente bidirezionale. Questo non avviene ne con l'assegnamento, nei linguaggi imperativi, ne con il pattern matching, nei linguaggi funzionali, perché entrambi questi meccanismi presuppongono una direzione per i legami delle variabili. Questa flessibilità evidentemente permette di utilizzare al meglio ogni programma, evitando duplicazioni del codice per modifiche che spesso sono solo di ordine sintattico.

La seconda peculiarità del paradigma logico è evidente nel caso dei linguaggi logici con vincoli, come abbiamo visto con l'ultimo programma della sezione precedente. Questo aspetto è particolarmente importante in tutti quei campi applicativi nei quali una singola soluzione, intesa come un insieme di specifici valori per le variabili di interesse, o non è ottenibile (perché i dati non contengono abbastanza informazioni) o non interessa, oppure è particolarmente difficile da ottenere dal punto di vista computazionale.

Il terzo aspetto infine, almeno in linea di principio, facilita la verifica di correttezza rispetto al caso imperativo, dove una lettura logica dei programmi è possibile ma richiede strumenti molto più espressivi (e complicati). Come già sottolineato nel Paragrafo 13.5, quest'aspetto è essenziale per poter ottenere degli strumenti formali di verifica di correttezza effettivamente usabili per programmi di dimensioni significative.

Questi tre aspetti, che possiamo riassumere nel principio della "computazione come deduzione", nel loro complesso permettono di esprimere in modo molto naturale forme di ragionamento tipiche dei problemi che si incontrano nell'ambito dell'intelligenza artificiale e della rappresentazione della conoscenza (ad esempio, nel contesto del, cosiddetto, Semantic Web). Il fatto (nei linguaggi implementati) che il controllo sia gestito interamente dalla macchina astratta mediante il meccanismo del backtracking, permette ad esempio di esprimere con grande facilità problemi che coinvolgono aspetti di ricerca in uno spazio di soluzioni.

I linguaggi logici possono essere anche usati con profitto come strumenti per la prototipazione rapida di sistemi: usando PROLOG si riesce a descrivere in breve tempo ed in modo compatto sistemi anche molto complessi, con il vantaggio rispetto ai linguaggi di specifica di avere dei programmi eseguibili.

L'utilizzo di risolutori di vincoli sofisticati ed anche di ottimizzatori (ad esempio basati sull'algoritmo del simplex) rende i linguaggi logici con vincoli competitivi rispetto ad altri formalismi in molte applicazioni commerciali dove siano richieste soluzioni di problemi combinatori, soluzioni di problemi di ottimizzazione e più in generale soluzioni di problemi esprimibili come relazioni su opportuni domini. Esempi in questo senso includono problemi di scheduling e altri problemi tipici delle ricerche operativa; analisi, sintesi e diagnosi di circuiti elettrici; matematica finanziaria e planning finanziario; ingegneria civile ed altro (per ulteriori esempi si veda il testo citato nella nota bibliografica del capitolo).

Fin qui i pregi. Accanto a questi, tuttavia, dobbiamo ricordare vari aspetti negativi dei linguaggi logici, almeno nelle versioni classiche attualmente esistenti.

Innanzitutto, come già ricordato in questo capitolo, il meccanismo di gestione del controllo basato sul backtracking permette un'efficienza limitata. Va detto che vari accorgimenti possono essere usati, sia a livello di costrutti usabili nei programmi, sia a livello di implementazione, per migliorare questo aspetto. Ad esempio, sono state definite macchine astratte specifiche per i linguaggi logici quali la WAM (Warren Abstract Machine) e varie ottimizzazioni. Tuttavia l'efficienza di un programma PROLOG rimane abbastanza limitata.

L'assenza di tipi e di moduli costituisce un secondo punto dolente, anche se in questo senso alcuni linguaggi più recenti hanno proposto soluzioni parziali (ad esempio, il linguaggio logico Mercury). Da quanto detto nel Capitolo 10 è chiaro che la mancanza di un sistema di tipi adeguato rende il controllo di correttezza dei programmi abbastanza difficile, nonostante la possibile lettura dichiarativa.

Anche i costrutti aritmetici e, in generale, i built-in esistenti in PROLOG non facilitano certo la correttezza dei programmi, in quanto sono di uso non facile a causa di varie sottigliezze semantiche. Questo è particolarmente evidente per quanto riguarda i costrutti per il controllo, il che spesso costringe a scegliere fra un programma chiaro ma poco efficiente ed un programma efficiente che però ha una semantica assai complicata.

Infine, per vari motivi (non ultimo un interesse commerciale limitato) purtroppo i linguaggi logici hanno quasi sempre un ambiente di programmazione a dir poco spartano, dove sono presenti pochissime delle caratteristiche disponibili, ad esempio, nei sofisticati ambienti disponibili per i linguaggi a oggetti.

Tutti questi aspetti negativi hanno sino ad oggi limitato la diffusione dei linguaggi logici, anche se rimane vivo l'interesse per questo paradigma di programmazione.

## 14.7 Sommario del capitolo

In questo capitolo abbiamo presentato le principali caratteristiche del paradigma logico, un paradigma relativamente recente (risale alla metà degli anni '70) che realizza la vecchia aspirazione di vedere la computazione come un processo interamente governato da leggi logiche. Con un unico capitolo a disposizione ci siamo dovuti limitare agli aspetti introduttivi, tuttavia quanto presentato ha la sufficiente precisione formale per poter illustrare senza ambiguità il modello computazionale dei linguaggi logici. In particolare, abbiamo visto:

- Alcune nozioni sintattiche relative alla logica del prim'ordine, necessarie per poter definire le clausole e dunque i programmi logici.
- Il processo di unificazione che costituisce il meccanismo computazionale di base; per far questo sono state necessarie alcune nozioni relative ai termini ed alle sostituzioni; abbiamo anche visto uno specifico algoritmo di unificazione.
- Come avviene la computazione nei programmi logici, mediante una regola di deduzione detta risoluzione SLD. Nel testo abbiamo scelto un approccio ispi-

rato ad una lettura procedurale delle clausole che evidenzia la similitudine con le "normali" procedure dei linguaggi imperativi. Due riquadri introducono le nozioni più formali.

- La differenza, a livello di modello computazionale, fra programmazione logica e PROLOG.

Dopo alcuni esempi abbiamo visto alcune estensioni significative del formalismo puro, in particolare:

- Alcune caratteristiche specifiche del linguaggio PROLOG. Rimangono tuttavia esclusi alcuni aspetti relativi alla meta-programmazione, alla programmazione di ordine superiore e ai costrutti per manipolare il programma durante l'esecuzione che sono molto importanti.
- L'idea dell'uso dei linguaggi logici nell'ambito delle basi di dati ed il Datalog.
- I linguaggi logici con vincoli, sostanzialmente attraverso un unico esempio.

Sia per gli aspetti mancati di PROLOG che, soprattutto, per un'introduzione più significativa a Datalog ed ai linguaggi logici con vincoli si rimanda alla letteratura suggerita qui sotto.

## 14.8 Nota bibliografica

Le idee sull'unificazione di Herbrand sono presenti nella sua tesi di dottorato del 1930 [42]. La definizione della regola di risoluzione e la prima formalizzazione dell'algoritmo di unificazione sono contenute nel lavoro di A. Robinson [84]. L'algoritmo di unificazione che abbiamo presentato è stato introdotto in [64]. Maggiori dettagli riguardanti la teoria dell'unificazione possono essere reperiti in vari articoli e testi, fra cui [37].

L'articolo "storico" di R. Kowalski che introduceva la risoluzione SLD e quindi la teoria della programmazione logica è [51]. I risultati di K.L. Clark sulla correttezza e completezza sono in [26].

Vi sono molti testi sia di teoria della programmazione logica che di programmazione in PROLOG. Fra i primi ricordiamo il testo di Lloyd [59] e quello più recente e più dettagliato di K. Apt [9]. Per la programmazione in PROLOG e per numerosi esempi di programmi anche non banali, consigliamo il classico testo di L. Sterling e E. Shapiro [94], dal quale abbiamo preso il programma che risolve il problema delle torri di Hanoi, e anche il libro di H. Coelho e J. C. Cotta [36], dal quale abbiamo preso il programma dell'esempio nel Paragrafo 14.1.1.

Infine, per quanto riguarda il linguaggio Datalog e, più in generale, l'uso della programmazione logica per le basi di dati, si può consultare [23], mentre [63] offre una buona introduzione ai linguaggi con vincoli.

## 14.9 Esercizi

- Si fornisca una grammatica libera che definisce le formule proposizionali (cioè quelle ottenute considerano solo predicati di arietà 0 e senza simboli di quantificazione).
- Supponiamo di rappresentare i naturali usando 0 per lo zero e  $s(n)$  per il successore di  $n$ . Si dica qual è la risposta calcolata dal seguente programma logico per un generico goal  $p(s, t, x)$  dove  $s$  e  $t$  sono termini che rappresentano numeri naturali.

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

- Si dica quale sono le risposte calcolate dal seguente programma logico per il goal  $p(X)$

```
p(0).
p(s(X)) :- q(X).
q(s(X)) :- p(X).
```

- Dato il seguente programma logico

```
member(X, [X|Xs]).
member(X, [Y|Xs]) :- member(X, Xs).
```

si dica qual è il risultato della valutazione del goal

```
member(f(X), [1, f(2), 3]).
```

- È dato il seguente programma PROLOG (ricordiamo che  $X$  e  $Y$  sono variabili mentre  $a$  e  $b$  sono costanti):

```
p(b) :- p(b).
p(X) :- r(b).
p(a) :- p(a).
r(Y).
```

Si dica se il goal  $p(a)$  termina o meno, giustificando la risposta.

- Si consideri il seguente programma logico

```
p(X) :- q(a), r(Y).
q(b).
q(X) :- p(X).
r(b).
```

Si dica se il goal  $p(b)$  termina o meno, giustificando la risposta.

- Supponendo di rappresentare i naturali usando 0 per lo zero e  $s(n)$  per il successore di  $n$  e di usare una primitiva  $write(x)$  per stampare il termine  $t$ , si scriva un programma logico che stampa tutti i numeri naturali
- Si definisca il predicato `sublist` in modo diretto, senza usare `append`.
- Si scriva in PROLOG un programma che calcoli la lunghezza (intesa come numero di elementi) di una lista e restituisca tale valore sotto forma di numero. (Suggerimento: si consideri una definizione induttiva della lunghezza e si usi l'operatore `is` per incrementare il valore nel caso induttivo).

- Si elenchino le principali differenze fra un programma logico ed un programma PROLOG.
- Se in un programma logico si cambia l'ordine degli atomi nel corpo di una clausola cambia la semantica del programma? Motivare la risposta.
- Se in PROLOG venisse cambiata la regola di scelta delle clausole (ad esempio scegliendo sempre quella più in basso invece che quella più in alto) cambierebbe la semantica del linguaggio? Motivare la risposta.
- Si fornisca un esempio di un programma logico  $P$  e di un goal  $G$  tali che la valutazione di  $G$  in  $P$  produca un effetto diverso quando si usano due regole di selezione diverse. (Suggerimento: dato che abbiamo visto che per le risposte calcolate non vi è differenza nell'uso di regole di selezione diverse, si consideri cosa accade con le computazioni che non terminano e che falliscono).
- Si descriva informalmente una regola di selezione che permetta di ottenere il minor numero possibile di computazioni che non terminano. (Suggerimento: le computazioni che non terminano, cambiando regola di selezione, potrebbero divenire fallimenti finiti. Si consideri una regola che garantisca che tutti gli atomi nel goal siano valutati).

## La programmazione concorrente

In questo capitolo affrontiamo un argomento sostanzialmente diverso da quelli considerati in precedenza. Sin qui, difatti, abbiamo considerato linguaggi *sequenziali*, nei quali in ogni momento dell'esecuzione di un programma vi è un unico contesto attivo. Tuttavia una parte molto significativa dei programmi esistenti ai nostri giorni, dal livello delle funzionalità del sistema operativo sino a quello dei servizi sul web, non rispetta tale limitazione in quanto vi sono più contesti d'esecuzione, attivi contemporaneamente, che *concorrono* a realizzare le funzionalità che ci interessano. Questo tipo di programmazione, detta appunto *concorrente*, è l'argomento di questo capitolo. Dopo una breve introduzione e qualche cenno di natura storica, necessario per capire il vasto panorama attuale, vedremo le principali problematiche che si hanno passando dal sequenziale al concorrente e le relative soluzioni. Esporremo solo le idee principali, dato che una trattazione esauriente richiederebbe un ulteriore volume. Seguendo il principio informatore di tutto questo testo, per i principi generali non faremo riferimento ad alcun linguaggio specifico, mentre cercheremo di concretizzare alcune nozioni esaminando il caso di Java.

### 15.1 Thread e processi

Nell'esecuzione di un programma possiamo efficacemente visualizzare il flusso del controllo immaginando un *thread*<sup>1</sup> che, nella struttura testuale del programma, colleghi tutti i comandi raggiunti dal controllo secondo l'ordine dato dall'esecuzione. Un thread del controllo, o più semplicemente thread, identifica dunque la sequenza di comandi di un programma eseguiti in una specifica computazione. In un programma *sequenziale* in ogni momento dell'esecuzione è attivo un unico thread, mentre in un programma *concorrente* vi sono più thread attivi. Spesso invece che di thread si parla di processi, task, programmi concorrenti, sempre intendendo il caso in cui si abbiano diversi contesti, di varia granularità, attivi

<sup>1</sup>“Thread” significa filo, trama. Qui abbiamo preferito adottare il termine originale inglese perché più significativo.

contemporaneamente. Tuttavia la terminologia non è consistente, dato che alcuni linguaggi chiamano processi quelli che noi abbiamo chiamato thread, mentre altri (Ada ad esempio) li chiamano task.

Qui cercheremo di impiegare una terminologia uniforme e quindi useremo il termine thread con il significato visto sopra, mentre il termine *processo* sarà usato per indicare un generico insieme di istruzioni in esecuzione, con il proprio spazio di indirizzamento, come di solito avviene nell'ambito dei sistemi operativi<sup>2</sup>. Secondo questa visione dunque un processo può essere visto come costituito da più thread diversi che ne condividono le risorse (ad esempio la memoria)<sup>3</sup>. Di solito i sistemi operativi distinguono i processi *pesanti* (heavyweight), che hanno un proprio spazio di indirizzamento, da quelli *leggeri* (lightweight), che possono condividere uno spazio comune. Questa terminologia a volte si usa anche per i thread che comunque per noi saranno sempre leggeri.

Infine, sempre riguardo alla terminologia, notiamo che quella inglese aiuta più di quella italiana. Infatti, mentre “concurrent” in inglese significa, appunto, “che avviene o è fatto allo stesso tempo”, in italiano “concorrente” non ha questa accezione specifica: concorrere (dal latino concurrere, ossia correre insieme) significa, secondo i dizionari attuali, cooperare, contribuire, convergere, competere, gareggiare. Tuttavia nell’uso informatico comune il termine “concorrente” oramai ha assunto il significato di “concurrent” e, come spesso avviene, in breve anche questa accezione sarà riportata dai dizionari e quindi passerà nella lingua ufficiale.

## 15.2 Una breve panoramica storica

Come molte altre aree importanti dell’informatica la concorrenza è nata da una innovazione nell’architettura hardware: dall’introduzione, negli anni sessanta, dei *controllori dei dispositivi* (device controller). Si tratta di meccanismi hardware che permettono di gestire l’ingresso e l’uscita di dati nei dispositivi periferici in modo autonomo, senza l’intervento diretto della CPU. Prima dell’avvento di tali dispositivi la gestione delle periferiche era estremamente inefficiente: la CPU, per poter eseguire un’operazione di ingresso o uscita, inviava un comando al dispositivo periferico e quindi si metteva in attesa del completamento del comando. Dato che le periferiche erano (e sono) dispositivi meccanici, con tempi di funzionamento relativamente molto lunghi, è evidente che in questo modo si aveva uno spreco di risorse (si pensi che un lettore di schede poteva leggere circa quattro schede al secondo, mentre un calcolatore già negli anni sessanta poteva eseguire circa quarantamila istruzioni al secondo). Con i controllori dei dispositivi ed i relativi

<sup>2</sup>Va notato che, nell’ambito della teoria della concorrenza e della cosiddetta algebra dei processi, il termine “processo” è usato per indicare un’entità computazionale astratta, spesso identificabile con un vero e proprio programma.

<sup>3</sup>Dal punto di vista implementativo tuttavia i thread possono essere realizzati usando le primitive del nucleo del sistema operativo per gestire i processi.

meccanismi di comunicazione basati su *interruzioni* si elimina questa inefficienza in quanto la CPU, per effettuare un’operazione di ingresso o uscita, invia un comando all’opportuno controllore del dispositivo e poi prosegue con l’esecuzione di altre istruzioni. Appena il dispositivo periferico ha terminato la propria esecuzione il relativo controllore segnala questo fatto alla CPU con un’interruzione: un segnale che comunica alla CPU di interrompere quello che sta facendo per eseguire il gestore dell’interruzione, che si occuperà della periferica (ad esempio, per trasferire i dati letti).

Le interruzioni introdussero nella programmazione nuove problematiche che, come dicevamo, furono fra le prime cause della nascita di meccanismi per la gestione della concorrenza. In particolare le interruzioni causano le cosiddette *race condition*, situazioni che si verificano in presenza di accesso a dati condivisi, quando il risultato della computazione può dipendere dall’ordine relativo delle operazioni. Difatti, dato che il momento in cui è lanciata un’interruzione non è prevedibile, parti diverse del programma (in particolare il programma principale ed il gestore dell’interruzione) potrebbero essere eseguite secondo un ordinamento temporale non prevedibile, con conseguenti risultati diversi. Si pensi ad esempio a due thread *P1* e *P2* di un programma che vogliono entrambi incrementare il valore di una variabile condivisa *x*, copiandolo prima dalla memoria centrale a un registro. Evidentemente se le operazioni vengono eseguite con il seguente ordinamento

```
leggi x in reg1;
leggi x in reg2;
incrementa reg2;
scrivi reg2 in x;
incrementa reg1;
scrivi reg1 in x;
```

dove le operazioni indentate sono quelle di *P2*, il risultato finale non sarà un incremento di due del valore della variabile, come invece ci aspetteremmo da una computazione corretta. Questo tipo di problemi può essere evitato sincronizzando opportunamente le varie operazioni e difatti contestualmente alle interruzioni nacquero i primi strumenti per la sincronizzazione e la gestione della concorrenza, quali sezioni critiche e meccanismi di mutua esclusione. I primi sistemi multiprogrammati, di tipo batch, permettevano una concorrenza molto limitata, essenzialmente solo a livello di sistema operativo (un processo cedeva la CPU ad un altro solo in occasione di operazioni di ingresso e uscita). Successivamente, con l’avvento dei sistemi in timesharing, l’esecuzione concorrente apparve anche al livello delle applicazioni e quindi vennero introdotti, nei linguaggi di alto livello, i primi costrutti per la comunicazione e la sincronizzazione.

Un secondo, importante, impulso allo sviluppo di tecniche e modelli per la concorrenza venne, alla fine degli anni sessanta, dalla realizzazione delle prime architetture multiprocessore. La presenza di più processori in queste architetture permette l’esecuzione contemporanea (o parallela) di più programmi diversi o anche di più parti di uno stesso programma. Questa possibilità pose problematiche nuove, sia algoritmiche che metodologiche; in effetti la *programmazione parallela*

la, sviluppatasi soprattutto per applicazioni di calcolo scientifico, interessa anche campi diversi da quelli solitamente collegati ai linguaggi di programmazione.

Un terzo elemento rilevante per lo sviluppo della programmazione concorrente è arrivato negli anni settanta con le prime reti di calcolatori. La possibilità di far “dialogare” macchine diverse, situate anche a grande distanza fra di loro, ha aperto la strada ad un modello di concorrenza ancora diverso: il *calcolo distribuito* nel quale i programmi che “girano” su macchine distinte, anche lontane, possono cooperare usando meccanismi basati sullo scambio di messaggi implementati da un’opportuna infrastruttura di comunicazione.

### 15.3 Tipi di programmazione concorrente

Come appare dalla questa breve nota storica, nell’ambito della programmazione concorrente, o concorrenza per brevità, possiamo distinguere varie tipologie.

Innanzitutto possiamo distinguere la *concorrenza fisica* dalla *concorrenza logica*: nella prima c’è una reale esecuzione simultanea, o *parallela*, di due thread (o processi, o anche programmi), come ad esempio nel caso di macchine con più processori; nel secondo caso invece la concorrenza c’è solo a livello logico, cioè al livello percepito dal programmatore e dalle applicazioni software; a livello fisico le varie attività concorrenti possono essere mappate in esecuzioni sequenziali, come nel caso di più utenti che contemporaneamente usino una macchina con un solo processore (il cui sistema operativo permetta la multiprogrammazione). Dal punto di vista concettuale e dei linguaggi di programmazione questa distinzione non ha molta rilevanza, salvo nel caso, importante per molte applicazioni, in cui si sia interessati principalmente alle prestazioni del sistema in termini di tempo di esecuzione. Difatti, se per risolvere un problema con un programma sequenziale impieghiamo tempo  $T_s$ , per risolvere lo stesso problema con  $n$  processori normalmente impiegheremo un tempo  $T_p < T_s$ <sup>4</sup>.

**Programmazione parallela** La programmazione che usa concorrenza “fisica” è detta usualmente *programmazione parallela* e coinvolge importanti aspetti algoritmici e metodologici, necessari per poter risolvere efficacemente un problema in parallelo. In questo ambito possiamo distinguere il *parallelismo sui dati*, nel quale la principale fonte di parallelismo deriva dall’applicazione di una stessa operazione a dati diversi, ed il *parallelismo sulle operazioni*, nel quale invece si tende ad applicare in parallelo più operazioni diverse ad uno stesso insieme di dati. Entrambi questi tipi di parallelismo hanno applicazioni pratiche molto im-

<sup>4</sup>Quanto minore dipende da molti fattori. In generale l’aumento delle prestazioni, detto anche *speedup*, è dato dalla quantità  $T_s/T_p$ : se questa è eguale a  $n$  si ha uno speedup lineare (detto anche perfetto) ma di solito lo speedup è strettamente minore di  $n$ . Vi sono anche casi in cui lo speedup è maggiore di  $n$  (o superlineare): questo avviene, ad esempio, se la computazione ha bisogno di una cache sensibilmente più grande di quella di un singolo processore, per cui le prestazioni sequenziali risultano particolarmente penalizzate rispetto a quelle concorrenti.

### Service Oriented e Cloud Computing

Con l’avvento del Web la programmazione distribuita ha trovato un nuovo impulso e si sono sviluppate forme sempre più avanzate di cooperazione fra un numero sempre maggiore di macchine diverse. In particolare, negli ultimissimi anni si è sviluppato il cosiddetto *Service Oriented Computing (SOC)* che, nell’ambito del Web, può essere considerato come una metodologia di sviluppo di applicazioni basata sulla composizione di componenti presenti sulla rete. Ogni componente realizza un servizio diverso ed è reperibile mediante le descrizioni, pubblicate sulla rete stessa, dell’interfaccia e delle funzionalità specifiche del componente stesso. Nuovi servizi possono essere quindi creati dinamicamente componendo servizi già esistenti secondo una metodologia che prevede un accoppiamento molto lasco fra i vari componenti. Ad esempio i servizi on-line di una compagnia aerea, di una catena di alberghi e di un noleggiatore di auto potrebbero essere combinati per creare un servizio che pianifica un viaggio. Più in generale, le *Service Oriented Architectures (SOA)* si stanno diffondendo, non solo nell’ambito del web, come sistemi distribuiti formati da un insieme di applicazioni indipendenti costruite mediante servizi. Sono stati proposti vari linguaggi specifici per programmare, comporre e “orchestrare” i diversi servizi. Fra tutti ricordiamo WS-BPEL (Web Services Business Process Execution Language), un linguaggio originariamente sviluppato da IBM e Microsoft che è attualmente uno standard OASIS, e JOLIE, un linguaggio sviluppato in Italia.

Un ulteriore sviluppo di questo tipo di metodologie è il cosiddetto *cloud computing*, che pur nell’attuale vaghezza del termine, indica un insieme di tecnologie (infrastrutture di comunicazione, modelli di comunicazione e altro) che permettono lo sviluppo di applicazioni distribuite su scala planetaria con una granularità molto più fine di quella che si ha nel SOC. Ovvero, mentre nel SOC si compongono servizi che già possiedono un certo livello di complessità, nel cloud computing, almeno in linea di principio, potremmo comporre servizi elementari quali uso dello spazio disco, uso della CPU, uso di una periferica, presenti in locazioni diverse. Il cloud computing quindi si propone come un modello di calcolo che permetta un semplice e facile accesso, di tipo on-demand, ad un insieme di risorse di calcolo condivise, configurabili e distribuite sulla rete. Anche se, nel momento in cui scriviamo, quasi tutto è da fare, il senso e le motivazioni di questo approccio sono chiare: il migliore sfruttamento delle risorse di calcolo disperse sulla rete che, per buona parte del tempo, rimangono inattive. Ovviamente tutto questo pone problemi di sicurezza e privacy, nonché problemi tecnici non indifferenti.

portanti, soprattutto nel campo del calcolo scientifico, spesso usando architetture multiprocessore particolari come i calcolatori vettoriali. Tipiche applicazioni del calcolo parallelo riguardano la soluzione di equazioni differenziali, quali quelle dei modelli meteorologici; questi sistemi di equazioni possono essere risolti usando metodi iterativi basati su calcoli matriciali che sono efficacemente parallelizzabili. La divisione del lavoro fra i vari processori porta a significativi aumenti delle prestazioni, a patto che gli algoritmi e i metodi di programmazione usati

permettano il bilanciamento del carico tra i processori.

Anche se la programmazione parallela introduce specifici problemi algoritmici e metodologici, dal punto di vista linguistico essa non pone problemi particolarmente diversi da quelli della concorrenza logica. Difatti un programma parallelo può usare varie architetture sottostanti, da macchine multiprocessore con memoria condivisa a multicomputer<sup>5</sup> o reti di calcolatori con memoria separata. Per la comunicazione e la sincronizzazione un tale programma può quindi usare gli stessi meccanismi di base (memoria condivisa e scambio di messaggi) che vedremo nel contesto della concorrenza logica e per questo motivo nel seguito ci concentreremo principalmente su questa.

Nell'ambito della *concorrenza logica*, nel seguito detta semplicemente concorrenza, distinguiamo la *programmazione multithreaded* da quella *distribuita*.

**Programmazione multithreaded** Questa è quella che abbiamo introdotto poco sopra e riguarda la presenza di più thread o processi, ossia di più contesti d'esecuzione, attivi contemporaneamente nell'ambito di una applicazione che "gira" su di un'unica macchina che tipicamente, o ha un'architettura tradizionale con un solo processore oppure è un multiprocessore a memoria condivisa. La necessità e l'utilità di usare thread e processi concorrenti dipende dalla natura del problema che l'applicazione deve risolvere. Si pensi ad esempio ad un browser moderno: quando seguiamo un link per aprire una nuova pagina (e spesso una nuova finestra) il programma apre un nuovo thread (o anche un nuovo processo) i cui compiti sono quelli di gestire la comunicazione dei pacchetti relativi alla nuova pagina, di gestire la formattazione ed il rendering della stessa (possibilmente aprendo nuovi thread) e altro ancora. L'utilizzatore nel frattempo può usare altre funzionalità del browser, senza che l'esecuzione di operazioni lente (quali la visualizzazione di immagini) penalizzi quelle più veloci. Evidentemente, nel caso di una macchina con un solo processore, opportune tecniche implementative sono necessarie per garantire che nessun thread occupi inutilmente la CPU (ad esempio quando è in attesa del completamento di un'operazione lenta) oppure mantenga il possesso della CPU per un tempo eccessivo.

**Programmazione distribuita** Con questo termine si intende la realizzazione di programmi concorrenti pensati per essere eseguiti su macchine realizzate da strutture fisiche distribuite. Queste possono essere multicomputer con memoria distribuita ma anche vere e proprie reti di calcolatori con varie architetture e topologie. Il punto importante qui è che i vari nodi del sistema di calcolo sono distribuiti fisicamente e quindi non si può assumere la presenza di una memoria condi-

<sup>5</sup>Per chiarezza espositiva nell'ambito delle macchine con più processori useremo il termine *multiprocessore* per le macchine con memoria condivisa e *multicomputer* per quelle con memoria distribuita. La terminologia in questo campo non è unica, inoltre vi sono molte possibili variazioni sul tema. Ai fini della presente trattazione comunque quello che importa è la presenza o meno di una memoria condivisa fra i diversi processori.

visa, per cui, come vedremo in dettaglio più avanti, si devono introdurre opportuni meccanismi e primitive di comunicazione mediante scambio di messaggi.

Un secondo asse lungo il quale si possono distinguere vari aspetti della programmazione concorrente riguarda la granularità delle varie attività concorrenti: queste infatti possono essere istruzioni di basso livello, comandi di un linguaggio di alto livello, parti di programma oppure interi programmi separati.

I due casi estremi di questa gerarchia non sono particolarmente interessanti ai fini della presente trattazione in quanto più che aspetti linguistici specifici riguardano problematiche architettoniche e aspetti relativi ai protocolli di comunicazione. Anche la concorrenza a livello di singolo comando, pur importante in pratica, riguarda più aspetti legati alla compilazione parallela efficiente che non costrutti per la gestione esplicita della concorrenza. Infatti, in questo tipo di concorrenza tipicamente si usa un tradizionale linguaggio sequenziale esteso con particolari annotazioni che permettono di fornire indicazioni al compilatore su come distribuire i dati in memoria, su quanti processori utilizzare e così via. Il compilatore, usando queste informazioni, produce un codice oggetto parallelo ottimizzato che può sfruttare al meglio le caratteristiche della macchina fisica (tipicamente si tratta di multiprocessori con memoria condivisa). Un esempio di linguaggio che permette questo tipo di compilazione parallela è High Performance FORTRAN (HPF) un'estensione del linguaggio FORTRAN 90, proposta negli anni novanta, che permette di ottenere programmi con parallelismo sui dati e che è particolarmente usata per applicazioni scientifiche.

In questo testo dunque ci concentreremo sulla concorrenza a livello di parti di programma, siano esse sottoprogrammi, unità o altro. In questo caso infatti sono necessari esplicativi costrutti linguistici di alto livello per gestire la comunicazione e la sincronizzazione delle varie parti, come vedremo meglio nel prossimo paragrafo.

**Nota terminologica** La trattazione che segue, salvo il caso specifico di Java trattato nel Paragrafo 15.7, non fa un particolare distinzione fra il caso dei thread e quello dei processi. Per semplicità quindi, fino al Paragrafo 15.7 useremo solo il termine "processo", intendendo però che ad esso potremmo sostituire anche "thread".

### 15.3.1 Meccanismi di comunicazione

Nell'ambito della programmazione concorrente, di qualsiasi tipo essa sia, due aspetti fondamentali, che caratterizzano modelli e linguaggi diversi, sono i meccanismi di *comunicazione* e di *sincronizzazione* impiegati.

Per quanto riguarda i primi, è evidente la loro necessità visto che le varie attività diverse che, come abbiamo detto, concorrono a realizzare una qualche funzionalità, dovranno poter comunicare fra di loro per scambiarsi informazioni di vario tipo (ad esempio risultati parziali delle computazioni o comunicazioni di controllo). Nei meccanismi che prevedono due partner nella comunicazione possiamo

distinguere chi produce una qualche informazione da chi la riceve<sup>6</sup>. Concettualmente possiamo individuare due meccanismi primari di comunicazione a seconda di come sono realizzati produttore e ricevente della comunicazione: quelli basati su *memoria condivisa* e quelli invece che usano lo *scambio di messaggi*. Queste due tipologie in effetti individuano classi diverse di modelli di concorrenza che saranno analizzate in dettaglio nei Paragrafi 15.4 e 15.5. Qui ci interessa soltanto accennare alle loro caratteristiche di base.

**Comunicazione a memoria condivisa** Questo tipo di comunicazione è quello che abbiamo già visto in altre parti di questo libro ed è tipico del paradigma imperativo: due parti di uno stesso programma, che hanno accesso ad una stessa variabile condivisa, possono comunicare semplicemente scrivendo nella variabile e leggendo il valore della stessa. Più in generale, nel modello di programmazione concorrente a memoria condivisa, i due partner, ad esempio due processi diversi, hanno accesso ad uno stesso spazio di memoria nel quale possono scrivere e leggere dei valori. Questo spazio di memoria può essere condiviso con più partner e permettere quindi anche comunicazioni di tipo broadcast nelle quali la stessa informazione prodotta è ricevuta da più partner.

**Comunicazione a scambio di messaggi** Secondo questo modello di comunicazione i partner della comunicazione non condividono alcuno spazio di memoria: per comunicare il mittente deve eseguire esplicitamente una operazione di invio, o *send*, mentre, in modo duale, il destinatario dovrà eseguire un'operazione esplicita di ricezione, o *receive*. Un'opportuna struttura di comunicazione, usualmente detta *canale*, dovrà essere accessibile sia al mittente che al destinatario per fornire un percorso che, almeno dal punto di vista logico, colleghi i due partner. L'implementazione di tale struttura, così come quella delle operazioni di *send* e *receive*, può essere la più varia e può dipendere sia dal livello di astrazione dell'operazione di comunicazione che dalla struttura hardware sottostante. Tuttavia è bene ricordare che, in linea di principio, i meccanismi di comunicazione della macchina fisica e quelli del linguaggio implementato su di essa possono essere diversi e, al limite, opposti. Così, ad esempio, è possibile che un linguaggio con comunicazione a scambio di messaggi implementato su una macchina multiprocessore usi la memoria condivisa per simulare i canali e realizzare le operazioni di *send* e *receive*, così come è possibile, almeno in linea di principio, che un multicomputer con memoria distribuita e comunicazione a livello hardware a scambio di messaggi sia usato per implementare un linguaggio con memoria condivisa.

<sup>6</sup>Nell'ambito della teoria della comunicazione di solito questi due partner sono detti mittente e destinatario; tuttavia qui, per uniformità con la terminologia corrente, riserveremo questi termini al solo modello con scambio di messaggi.

## Blackboard

Un modello di comunicazione in qualche modo intermedio fra la memoria condivisa e lo scambio di messaggi è quello basato sull'architettura a *blackboard* (o a store condiviso) nella quale vari processi separati, che possono avere un proprio spazio di memoria privato, condividono una stessa zona di memoria detta blackboard (o store) e usata per la comunicazione. Il modello più noto in questo ambito è quello del linguaggio Linda nel quale la blackboard è detta *spazio delle tuple* ed è un'astrazione di una memoria associativa condivisa dai vari processi. Questi comunicano fra di loro usando opportune primitive di comunicazione per spedire e ricevere messaggi dallo spazio delle tuple. La comunicazione dunque, anche se usa lo scambio di messaggi, avviene in modo indiretto ed è mediata da una memoria condivisa. Questo modello, studiato nell'ambito dei cosiddetti linguaggi di coordinazione, per quanto interessante è sicuramente meno rilevante degli altri due e non verrà ulteriormente considerato in questo testo.

### 15.3.2 Meccanismi di sincronizzazione

Il secondo aspetto fondamentale nell'ambito dei vari modelli della concorrenza esistenti riguarda i meccanismi di *sincronizzazione* impiegati. Questi meccanismi permettono di controllare l'ordine relativo delle varie attività nei diversi processi e sono essenziali per la correttezza della computazione. Nell'ambito della comunicazione a memoria condivisa quest'aspetto è evidente e i meccanismi di sincronizzazione sono presenti in modo esplicito. Si pensi ad esempio al caso di due processi che comunicino usando una variabile condivisa *x*: ovviamente il processo ricevente deve prelevare il valore di *x* solo dopo che il processo produttore ha scritto in *x* il valore corretto. Più in generale, nella programmazione concorrente con memoria condivisa si usano due tipologie di sincronizzazione: la *mutua esclusione*, che permette di garantire che determinate regioni critiche del codice non siano accessibili contemporaneamente a più processi, e la *sincronizzazione su condizione* (o *condizionale*), che invece permette di sospendere l'esecuzione di un processo fino al verificarsi di una opportuna condizione.

Questi due tipi di sincronizzazione sono realizzati usando l'*attesa attiva* (busy waiting, detta anche spinning) e il *blocco* (detta anche *sincronizzazione basata sullo scheduler*). Nella prima il processo in attesa impiega attivamente la CPU eseguendo un ciclo nel quale testa continuamente una condizione. Nella seconda invece il processo che deve essere posto in attesa rilascia la CPU, perché questa possa essere usata da altri. Lo scheduler provvederà poi a riattivare il processo quando la condizione d'attesa sarà verificata. Vi sono vari meccanismi che permettono di realizzare questo tipo di sincronizzazione, in particolare semafori e monitor. Vedremo nel dettaglio tutte queste tecniche nel Paragrafo 15.4.

Anche nell'ambito della comunicazione a scambio di messaggi sono presenti meccanismi di sincronizzazione, tuttavia di solito questi sono impliciti nella definizione delle primitive di *send* e *receive*. Difatti certamente un messaggio prima di essere ricevuto deve essere stato spedito, tuttavia può essere la stessa

primitiva `receive` che implementa questa sincronizzazione, ad esempio sospendendo l'attività del processo fino a quando il messaggio non è stato spedito, senza la necessità di un ulteriore costrutto esplicito. Considerando dunque il tipo di sincronizzazione realizzato nelle primitive di invio e ricezione possiamo identificare un meccanismo di comunicazione *sincrono* e uno *asincrono*, che discutiamo qui di seguito.

**Comunicazione asincrona** In questo tipo di comunicazione l'invio e la ricezione di un messaggio avvengono in momenti diversi, come nel caso della posta elettronica. Il mittente fa un'operazione di `send` e quindi procede nella computazione senza aspettare che il destinatario abbia effettuato la corrispondente `receive`. Si dice in tal caso che l'operazione di `send` non è bloccante. La `receive` invece è bloccante: il processo destinatario dopo aver effettuato una `receive` viene bloccato e attende che il mittente effettui l'operazione di `send`. Solo dopo che questa è stata fatta il destinatario può proseguire nella computazione. La comunicazione asincrona presenta qualche difficoltà implementativa nella realizzazione del canale, dal momento che esso, in linea di principio, può contenere un numero arbitrario di messaggi spediti e non ancora ricevuti.

**Comunicazione sincrona** In questo caso la comunicazione avviene solo quando sia il mittente che il ricevente sono pronti ad effettuare la comunicazione mediante uno specifico canale, come nel caso di una telefonata (senza uso di segreterie telefoniche). Concettualmente quindi nella comunicazione sincrona la `send` e la `receive` vengono fatte allo stesso tempo. Ovviamente nella realtà implementativa le operazioni avvengono in momenti diversi, comunque in pratica questo significa che il mittente, dopo aver effettuato un'operazione di `send`, prima di proseguire nella sua computazione aspetta che il destinatario effettui l'operazione di `receive`. Il destinatario si comporta come nel caso della comunicazione asincrona. Quindi, nel caso della comunicazione sincrona, sia la `send` che la `receive` sono bloccanti. L'implementazione della comunicazione sincrona può usare un buffer di dimensione limitata per realizzare il canale di comunicazione, visto che non può essere spedito un nuovo messaggio se il precedente non è stato ricevuto.

Richiamiamo l'attenzione del lettore a non fraintendere la precedente terminologia: essa si riferisce ai meccanismi di comunicazione e non a quelli di esecuzione. Se consideriamo quest'ultimi, i linguaggi che noi trattiamo in questo capitolo sono tutti asincroni, nel senso che ogni processo, thread, task o programma viene eseguito con una propria velocità (dipendente ovviamente anche dal particolare processore sottostante) senza che questa sia direttamente collegata a quella di altri processi. Esistono anche linguaggi e modelli sincroni, nei quali, o per il tipo di architettura usato o per le assunzioni fatte sul modello astratto di computazione impiegato, i tempi di esecuzione di processi diversi possono essere sincronizzati. Questi linguaggi hanno notevole interesse nei sistemi real-time ed embedded, ma la loro trattazione va oltre gli scopi di questo testo.

## 15.4 Memoria condivisa

In questo paragrafo prendiamo in esame i meccanismi di sincronizzazioni usati nei modelli di programmazione concorrente che usano memoria condivisa. Come detto in precedenza, in questo caso la comunicazione fra due o più partner avviene semplicemente con l'accesso ad una stessa zona di memoria condivisa che, per semplicità, nel seguito identifieremo con una variabile. Perché la comunicazione sia corretta ovviamente servono specifiche tecniche di sincronizzazione per evitare race conditions ed altre situazioni scorrette. Come visto nel Paragrafo 15.3.1 le due principali forme di sincronizzazione nel caso della memoria condivisa sono la mutua esclusione e la sincronizzazione condizionale. La *mutua esclusione* permette di escludere che le sezioni del codice che consentono l'accesso a risorse condivise, le cosiddette *sezioni critiche*, siano eseguite contemporaneamente da più processi. In questo modo, in sostanza, si possono considerare intere sezioni del codice come atomiche dal punto di vista dell'esecuzione. La *sincronizzazione su condizione* (o *condizionale*) invece permette di sospendere l'esecuzione di un processo fino al verificarsi di una opportuna condizione, quale ad esempio un dato valore per una variabile<sup>7</sup>. Questi due tipi di sincronizzazione sono realizzati usando due tecniche: l'*attesa attiva* (busy waiting, detta anche spinning) e il *blocco* (detta anche sincronizzazione basata sullo scheduler).

### 15.4.1 Attesa attiva

Nell'attesa attiva il processo che deve attendere il proprio turno (per entrare in una sezione critica oppure e sia verificata una data condizione) esegue un ciclo nel quale valuta continuamente una condizione fino a quando questa non diventa vera. Nell'attesa viene impiegata attivamente la CPU e quindi questa tecnica non ha molto senso se impiegata su una macchina uniprocessore per sincronizzare processi diversi: ovviamente non possiamo usare la CPU per aspettare che si verifichi una condizione (ad esempio lo svuotamento di un buffer) che dovrebbe essere prodotta da un altro processo usando la stessa CPU. Si tratta invece una tecnica efficiente se usata su macchine multiprocessore (o anche se usata a livello hardware, ad esempio per sincronizzare il trasferimento di dati su bus e reti locali). Vi sono vari meccanismi di sincronizzazione che usano l'attesa attiva: i più importanti sono i *lock* (lucchetti) per realizzare la mutua esclusione e le *barriere* per realizzare la sincronizzazione su condizione. Vediamoli entrambi.

**Lock** Per realizzare un meccanismo di mutua esclusione innanzitutto si deve chiarire quali sono le istruzioni atomiche di cui possiamo disporre, ossia quali sono le istruzioni la cui indivisibilità è garantita dalla sottostante macchina. Di

<sup>7</sup> La mutua esclusione potrebbe apparire un caso particolare di sincronizzazione su condizione, nella quale la condizione è: "nessun altro processo è in una sezione critica". Questo tuttavia assumerebbe un controllo sullo stato globale del sistema che di solito nella sincronizzazione su condizione non si ha.

solito a livello hardware è presente un'istruzione atomica `test_and_set(B)` che restituisce il valore della variabile booleana `B` e quindi, in modo atomico, ovvero non interrompibile, assegna il valore `true` a tale variabile. Usando una tale istruzione possiamo realizzare la mutua esclusione come segue.

Innanzitutto, supponiamo che ogni processo che voglia accedere alla sezione critica abbia la seguente struttura (al solito, qui usiamo in modo abbastanza libero il nostro pseudo linguaggio):

```
process Pi {
    sezione non critica;
    acquisisci_lock(B);
    sezione critica;
    rilascia_lock(B);
    sezione non critica;
}
```

La sezione critica, come già detto, racchiude le strutture dati condivise per le quali vogliamo garantire la mutua esclusione, ossia per le quali vogliamo impedire accessi contemporanei. Supponiamo anche che la variabile condivisa `B`, di tipo booleano, sia inizializzata a `false`:

```
bool B = false;
```

Questa variabile realizza il cosiddetto lock, ossia il lucchetto da usarsi per l'accesso alla sezione critica. Quindi, usando l'istruzione `test_and_set(B)`, possiamo definire le funzioni di acquisizione e rilascio del lock come segue:

```
void acquisisci_lock(ref B: bool) {
    while test_and_set(B) do skip;
}

void rilascia_lock(ref B: bool) {
    B = false;
}
```

(`skip` qui è il comando che non produce alcun effetto: è stato inserito nel codice solo per maggior chiarezza espositiva). La procedura di acquisizione del lock consiste in un ciclo (detto anche *spin*, da cui il termine *spin lock* per questa tecnica) nel quale si testa la variabile condivisa `B` fino a quando questa non ha valore *falso*, il che denota la possibilità di accesso alla sezione critica. Il test è fatto nella stessa azione atomica che assegna a `B` il valore *vero*, in modo tale che solo uno fra i vari test provati contemporaneamente (da più processi) abbia successo. È evidente dunque che fra i vari processi che effettuano la `acquisisci_lock(B)` solo uno riuscirà ad accedere alla sezione critica a causa dell'atomicità dell'operazione di `test_and_set` e dunque è garantita la mutua esclusione. Si noti anche che se  $n$  processi tentano di accedere alla sezione critica almeno uno riuscirà a farlo. È garantita dunque anche l'assenza di situazioni nella quali ogni processo aspetta la conclusione delle attività di un altro senza che nessuno possa progredire. In altri termini, è garantita l'assenza di *deadlock*. La precedente implementazione della mutua esclusione presenta invece un grave problema di efficienza sulle macchine multiprocessore. Difatti, i vari processi che tentano di accedere alla sezione

critica, ripetendo l'operazione di `test_and_set` all'interno del ciclo `while` continuano ad effettuare delle operazioni di scrittura nella variabile condivisa. Tali operazioni, effettuate in modo atomico insieme al test del valore della variabile, causano una competizione (o contention, secondo la terminologia inglese) per l'uso della memoria e dei bus che finisce per degradare la performance del sistema.

Per evitare questo problema si può ricorrere alla tecnica del *test and test\_and\_set*, nella quale invece di eseguire una operazione di `test_and_set` sulla variabile condivisa, con conseguente scrittura, ad ogni ciclo, si esegue nel ciclo solo l'operazione di test (lettura). Quando tale operazione ha successo si passa quindi alla `test_and_set` per ottenere effettivamente l'accesso alla sezione critica. Secondo questa tecnica la funzione di acquisizione del lock è realizzata come segue (il rilascio avviene come prima):

```
void acquisisci_lock(ref B: bool) {
    while B do skip;
    while test_and_set(B) do
        {while B do skip; }
}
```

Sia questa che la precedente implementazione della mutua esclusione presentano un ulteriore problema: non è garantito che un processo che voglia accedere alla sezione critica prima o poi riesca a farlo. Potrebbe infatti accadere che quando la sezione critica viene rilasciata vi sia sempre un altro processo che riesce ad accedervi prima. Questo perché non facciamo alcuna ipotesi sulla gestione dei processi e, in particolare, sul loro scheduler. Anche se nella pratica un tale comportamento appare abbastanza improbabile, vi sono specifici algoritmi che assicurano che un processo in attesa per accedere alla sezione critica, o più in generale in attesa di procedere, prima o poi riuscirà soddisfare la sua richiesta. Queste proprietà vengono chiamate di "fairness" e la loro definizione precisa va oltre gli scopi di questo testo (vi sono infatti varie nozioni diverse di fairness, e in alcuni casi le differenze sono sottili). Ci accontenteremo dunque di un uso intuitivo di questo termine: assumendo che esistano più pretendenti per una risorsa comune, una politica di gestione degli accessi alla risorsa è (debolmente) *fair* se ogni pretendente che è abilitato all'accesso infinite volte riuscirà ad accedere alla risorsa. Una soluzione fair per la mutua esclusione può essere ottenuta seguendo l'idea che, nel caso in cui vi siano più processi che competono per l'accesso alla sezione critica, debbano essere stabiliti dei turni. Un algoritmo che implementa in modo intuitivamente semplice quest'idea è quello del ticket, che tutti abbiamo visto in funzione in un qualche ufficio per gestire l'accesso dei clienti ad uno sportello: vi è un distributore che eroga ai clienti dei numeri (ticket) ordinati in senso crescente e la risorsa viene concessa, in mutua esclusione, seguendo tale ordine. Un limite di questo algoritmo è nel fatto che i valori crescono in modo illimitato, almeno in linea di principio. Per evitare questo possono essere usati algoritmi specifici, quali il tie-breaker, per i quali si rimanda alla letteratura specializzata.

**Barriere** Le barriere sono un meccanismo di sincronizzazione su condizione tipicamente usato nella programmazione parallela. Qui spesso si hanno algoritmi

implementati da  $n$  processi diversi che concorrono a produrre la soluzione finale usando metodi iterativi (si pensi alla manipolazione di matrici). Alla fine di ogni iterazione ogni processo deve attendere il completamento delle attività di tutti gli altri processi per poter passare all'iterazione successiva. Questo si può ottenere introducendo nel codice un punto di sincronizzazione, detto *barriera*, nel quale ogni processo attende il verificarsi di una condizione globale prima di procedere. Ogni processo  $P_i$ , con  $i = 1 \dots n$ , dunque avrà uno schema di questo tipo

```
process Pi {
    while B true do {
        codice per compito i;
        attendi che ogni altro Pj abbia terminato;
        // barriera
    }
}
```

Per realizzare l'attesa, e quindi la barriera, possiamo usare un contatore globale, inizializzato a zero prima del ciclo, che ogni processo incrementa al termine del proprio compito e sul quale si attende il raggiungimento del valore  $n$ . Le operazioni di accesso al contatore devono essere fatte in modo atomico e l'attesa può essere di tipo attivo.

Vi sono molti modi diversi, anche abbastanza sofisticati, per realizzare le barriere. Trattandosi di meccanismi abbastanza specifici per la programmazione parallela qui non li prenderemo ulteriormente in considerazione.

#### 15.4.2 Sincronizzazione basata sullo scheduler

I meccanismi di attesa attiva che abbiamo visto nel precedente paragrafo presentano vari inconvenienti. Innanzitutto normalmente vi sono più processi che processori, per cui non è conveniente utilizzare il tempo di CPU per effettuare un'attesa attiva di un processo, mentre altri processi sono in attesa di usare la CPU<sup>8</sup>.

Inoltre le variabili usate per la sincronizzazione negli esempi che abbiamo visto, e nella pratica reale, sono variabili "normali", non strutturate in un particolare tipo di dato o comunque in un qualche costrutto linguistico. Questa situazione è da evitarsi, come abbiamo imparato dai tipi di dato e dagli oggetti, se vogliamo minimizzare la probabilità di commettere errori di programmazione.

Per ovviare a questi problemi sono stati introdotti dei meccanismi di sincronizzazione basati sullo scheduler, detti anche basati sul *blocco*, nei quali il processo che deve essere posto in attesa rilascia la CPU, perché questa possa essere usata da altri, associando però all'evento che il processo sta attendendo un'informazione che identifichi il processo stesso. Quando l'evento si verificherà lo scheduler potrà sapere che quel particolare processo è pronto per essere eseguito. Questi meccanismi di sincronizzazione di solito usano delle variabili che possono assumere valori definiti e delle specifiche operazioni su tali valori. Possiamo dunque

<sup>8</sup>Un'eccezione a questo principio si ha quando il tempo di attesa attiva è inferiore a quello necessario per fare il cambio di contesto necessario per eseguire un altro processo.

vederli, almeno a livello concettuale, come dei tipi di dato, anche se raramente essi compaiono esplicitamente come costrutti in linguaggi di alto livello.

Nel seguito vedremo i due principali meccanismi di questo tipo, ossia semafori e monitor, che permettono di realizzare sia la mutua esclusione sia la sincronizzazione su condizione.

**Semafori** Il meccanismo basato sui *semafori*, introdotto da Dijkstra negli anni 60, è stato il primo strumento esplicito di sincronizzazione per i sistemi a memoria condivisa e rimane tutt'oggi uno dei costrutti di sincronizzazione più importanti fra quelli implementati a livello di nucleo di sistema operativo. Presenti già in ALGOL 68, i semafori sono usati in quasi tutte le librerie per la programmazione concorrente e appaiono anche in alcuni linguaggi, ad esempio SR e Modula-3.

Un semaforo, come suggerisce il nome stesso, può essere visto come un costrutto che regola il "traffico" dei processi evitando "collisioni", in modo del tutto analogo allo strumento che regola il traffico sulle linee ferroviarie, dove un valore (colore) di una opportuna "variabile" indica se un tratto di binario è libero, e quindi permette o meno l'accesso del treno. Tale valore è manipolato da opportune operazioni effettuate quando un treno occupa (da verde a rosso) o rilascia (da rosso a verde) il tratto di binario<sup>9</sup>.

In termini più informatici un semaforo può essere visto come un tipo di dato, che chiamiamo `sem`, formato da:

- l'insieme dei valori, costituito dai numeri interi  $\geq 0$ ;
- due operazioni atomiche chiamate `P` e `V`<sup>10</sup>.

Una variabile `s` di tipo semaforo, dichiarata come al solito con  
`sem s`

fornisce una struttura condivisa che permette la sincronizzazione fra processi, sulla base del valore di `s`, mediante le due operazioni `P` e `V` dove, intuitivamente, la `P` permette l'accesso al semaforo e la `V` invece consente il rilascio dello stesso.

Più in dettaglio, se un processo `P` esegue l'operazione

`P(s)`

l'effetto è il seguente. Se il valore di `s` è  $> 0$  allora `s` è decrementata e l'operazione termina, per cui il processo può proseguire con l'esecuzione del codice che segue la `P(s)` (cioè con l'accesso alla sezione critica). Se invece quando è invocata la `P(s)` la variabile `s` vale 0 allora l'operazione stessa, e quindi il processo

<sup>9</sup>Qui abbiamo, per ovvi motivi, semplificato la visione ferroviaria che in realtà usa anche il giallo ed è più complessa.

<sup>10</sup>I nomi di queste due operazioni derivano dalle iniziali di due parole in olandese, la lingua di Dijkstra. `V` sta per "verhogen" che significa aumentare. `P` invece sta per "prolaag", una parola creata da Dijkstra stesso come abbreviazione di "probeer te verlagen", cioè "tentare di ridurre" ossia "tentare di decrementare". Il motivo di questa complicazione sta nel fatto che, in olandese, anche la parola "decrementare" inizia con una `v`.

$P_r$ , è bloccata fino a quando  $s$  non assume un valore  $> 0$ . Questo blocco è di solito gestito a livello di nucleo di sistema operativo, nel senso che il processo  $P_r$  che viene posto in attesa rilascia l'uso della CPU, che può quindi essere usata da altri. Il processo sarà poi "risvegliato" al momento in cui verrà segnalato l'evento (sempre a livello di sistema operativo) che la variabile  $s$  ha assunto un valore  $> 0$ <sup>11</sup>. Come accennato poc'anzi queste operazioni che implementano la  $P(s)$  devono essere pensate come eseguite in un'unica azione atomica. Questo significa che, ad esempio, se il valore di  $s$  è 1 e ci sono due processi che eseguono un'operazione  $P(s)$ , solo una delle operazioni avrà successo e permetterà l'accesso alla sezione critica (l'altra operazione sarà bloccata).

#### L'operazione $V(s)$

come detto, permette di segnalare che la risorsa che si era impegnata con una  $P(s)$  adesso torna ad essere libera. Per fare questo dunque la  $V(s)$  incrementa, con un'unica operazione atomica, il valore di  $s$ . Il processo che esegue la  $V(s)$  non viene mai bloccato.

La variabile  $s$  può assumere i valori 0 e 1, nel qual caso si parla di semaforo binario, oppure può assumere un generico valore non negativo, nel qual caso si ha un semaforo di tipo generale.

Dato che il "risveglio" delle operazioni  $P$  bloccate è gestito normalmente dallo scheduler dei processi, la fairness nella gestione di queste operazioni, e dunque la fairness nell'accesso dei processi alle zone critiche, dipende dalle ipotesi di fairness dello scheduler stesso. Normalmente le implementazioni dei semafori assicurano che la coda dei processi bloccati in attesa del completamento di un'operazione  $P$  sia gestita in modo FIFO: i primi processi bloccati sono i primi ad essere svegliati. Dunque, se vi sono abbastanza operazioni  $V$ , in questo caso la fairness è garantita.

Come esempio di uso dei semafori vediamo il problema dei filosofi a cena, un classico problema di sincronizzazione dovuto a Dijkstra. Cinque filosofi siedono ad un tavolo come mostrato in Figura 15.1 e mangiano da un piatto comune posto al centro del tavolo. Ogni filosofo alterna le attività di mangiare e pensare. Ogni filosofo può usare solo le forchette poste ai lati del suo piatto, inoltre per mangiare ogni filosofo ha bisogno di due forchette<sup>12</sup>. Dato che ci sono tante forchette quanti sono i filosofi, evidentemente se ogni filosofo prende una sola forchetta (ad esempio quella a destra del piatto) nessuno riesce a mangiare. In questo caso infatti si crea una situazione di *deadlock* nella quale ogni processo (filosofo) per poter procedere deve attendere il rilascio di una risorsa (forchetta) da parte di un altro processo, senza quindi che nessuno possa procedere. Nel caso del nostro problema per evitare situazioni di deadlock basta evitare un'attesa circolare (il primo

<sup>11</sup>In linea di principio si potrebbero implementare i semafori usando anche l'attesa attiva (almeno su una macchina multiprocessore). Tuttavia di solito i semafori sono implementati come detto sopra, di cui la scelta di metterli in questo paragrafo.

<sup>12</sup>I testi in lingua inglese attribuiscono questa necessità al fatto che il piatto contiene spaghetti, notoriamente difficili da maneggiare per i non italiani.

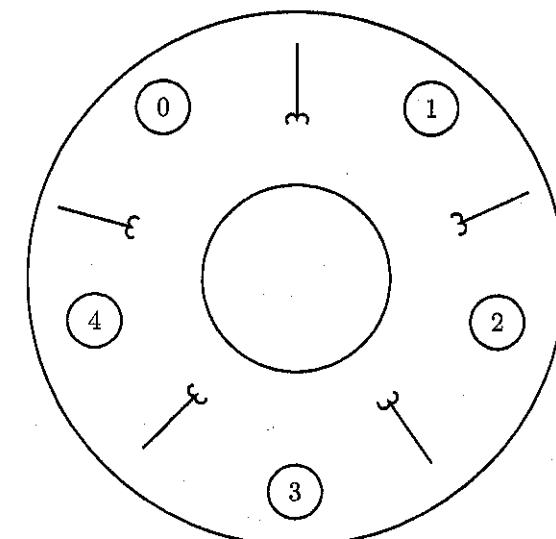


Figura 15.1 Il problema dei filosofi a cena.

processo che attende il secondo che attende il terzo e così via fino nuovamente al primo). Questo si può ottenere, ad esempio, assumendo che il primo filosofo scelga prima la forchetta di destra e poi quella di sinistra, mentre tutti gli altri prendano prima quella di sinistra e poi quella di destra. Una soluzione per questo problema di sincronizzazione è dunque la seguente. Assumiamo di avere un vettore *forchette* di 5 semafori, uno per ogni forchetta, che inizializziamo a 1. Inoltre indichiamo con *Filosofo*[i] il generico filosofo i-esimo (con  $i \in [0, 4]$  per analogia con gli indici dell'array: la forchetta numero  $i$  è quella a destra del filosofo  $i$ ). La selezione di una forchetta è realizzata da un'operazione  $P$  mentre il rilascio da una  $V$ .

```
sem forchette[5];
for (i=0, /i<=4, i+=1) {forchette[i] = 1};

process Filosofo[0] {
    while true {
        pensa;
        P(forchette[0]); // acquisisce forchetta di sinistra
        P(forchette[4]); // acquisisce forchetta di destra
        mangia;
        V(forchette[0]); // rilascia forchetta di sinistra
        V(forchette[4]); // rilascia forchetta di destra
    }
}

process Filosofo[i]{ % i = 1, 2, 3, 4
    while true {
        pensa;
```

```

P(forchette[i-1]); // acquisisce forchetta di destra
P(forchette[i]) // acquisisce forchetta di sinistra
mangia;
V(forchette[i-1]); // rilascia forchetta di destra
V(forchette[i]) // rilascia forchetta di sinistra
}
}

Filosofo[0] || Filosofo[1] || Filosofo[2] || Filosofo[3] || Filosofo[4]

```

L'ultima riga di codice indica l'esecuzione concorrente dei processi corrispondenti ai 5 filosofi ed è facile convincersi che produce la soluzione richiesta. Difatti, supponiamo che il filosofo numero 1 prenda la sua prima forchetta, che è quella alla sua destra. A questo punto se il filosofo numero 2 non prende la forchetta alla sua destra, il numero 1 può prendere anche la forchetta di sinistra e quindi mangiare. Altrimenti possiamo ripetere il ragionamento con il filosofo numero 2 e così via fino alla situazione in cui il numero 4 abbia preso la forchetta alla sua destra. A questo punto il numero 0, che usa un diverso ordinamento per la scelta delle forchette, può prendere la forchetta alla sua destra (e quindi alla sinistra di 4) solo se ha già preso la forchetta di sinistra (nel qual caso, può mangiare). Se invece la forchetta a sinistra di 0 è presa dal filosofo 1 (come prima forchetta) allora 0 non può prendere la sua forchetta di destra, che può quindi essere presa come forchetta di sinistra da 4 che può finalmente avere il sospirato piatto di spaghetti. Per quanto riguarda l'operatore `||` si veda quanto detto nella sezione 15.6.1.

**Monitor** Abbiamo visto in capitoli precedenti come i meccanismi di astrazione siano importanti anche nell'ambito delle strutture dati. In particolare, abbiamo analizzato in dettaglio l'idea di base dei tipi di dato astratto prima, e degli oggetti poi, cercando di capire perché è importante poter fornire dei costrutti che incapsulino sia la rappresentazione astratta di un dato che le operazioni possibili su di esso. Da questo punto di vista i semafori sono dunque deficitari, in quanto sono costrutti di basso livello che non offrono sufficienti meccanismi di strutturazione e astrazione. Un semaforo infatti è sostanzialmente una variabile condivisa fra più processi ed è compito del programmatore fare in modo che tale variabile sia usata correttamente, usando le P e le V, con tutti i rischi di correttezza conseguenti (uso del semaforo sbagliato, omissione di una P o V, uso di operazioni diverse da P e V eccetera). Inoltre si noti anche che i semafori possono essere usati sia per la mutua esclusione che per la sincronizzazione condizionale, cosa questa non completamente soddisfacente in quanto i due concetti sono diversi e, per la comprensione del programma, sarebbe bene che questa diversità apparisse anche a livello linguistico.

Per ovviare a queste carenze dei semafori, negli anni '70 sono stati introdotti i *monitor*, anch'essi proposti inizialmente da Dijkstra, successivamente sviluppati da Brinch Hansen e Hoare e poi usati in molti linguaggi di programmazione, incluso Java. Un monitor è un tipo di dato astratto che incapsula la rappresentazione di un oggetto astratto con stato e fornisce le operazioni permesse per agire su tale oggetto. Con più precisione, un monitor contiene delle variabili permanenti, che

rappresentano lo stato dell'oggetto, delle procedure, per agire su tali variabili, e dei comandi di inizializzazione, usati quando il monitor è creato.

Senza far riferimento ad alcun linguaggio possiamo dunque assumere che una dichiarazione di un monitor abbia la seguente struttura

```

monitor NomeMonitor {
    variabili;
    comandi init;
    procedure;
}

```

Le variabili permanenti esistono fino a quando esiste il monitor (ovviamente le procedure possono avere proprie variabili locali che hanno un tempo di vita inferiore). Fra di esse vi sono alcune variabili dette *condizionali* che, come vedremo, permettono la sincronizzazione condizionale. All'esterno del monitor sono visibili solo i nomi delle procedure (con le relative segnature) che dunque sono le uniche operazioni usabili dai processi (esterni) per agire sulle variabili permanenti, analogamente a quanto avviene con i campi degli oggetti definiti con visibilità *private*. Le variabili permanenti sono inizializzate, dai relativi comandi, quando il monitor è creato e prima dell'uso di qualsiasi procedura.

In un programma concorrente che usa i monitor possiamo distinguere componenti passivi e attivi. I primi, introdotti da dichiarazioni analoghe a quella vista sopra, sono i monitor e contengono le strutture dati condivise fra i vari processi. I secondi sono i processi e questi possono interagire usando le procedure (e quindi, indirettamente, le variabili) dei monitor.

Già da questa prima descrizione è evidente il vantaggio dei monitor rispetto ai semafori: il programmatore che usa un monitor può programmare un processo in modo relativamente indipendente, conoscendo solo l'interfaccia esterna del monitor, ignorando come le procedure di questo siano implementate ed anche, in parte, come tali procedure siano usate da altri processi. Questo è il solito vantaggio dei tipi di dato astratto e degli oggetti, che però qui assume una rilevanza ulteriore: dunque, a differenza di quanto avviene nella programmazione sequenziale, il monitor è condiviso da vari processi concorrenti che sono attivi contemporaneamente con tempi e velocità diverse. È dunque essenziale poter essere in grado di programmare ogni processo in modo relativamente indipendente, sia per la comprensione del programma che, soprattutto, per la sua correttezza. Ovviamente devono essere risolti i problemi di mutua esclusione e sincronizzazione ma questo, usando i monitor, è relativamente semplice.

La mutua esclusione infatti, usando i monitor, è garantita implicitamente dal costrutto stesso, senza che il programmatore debba fare niente: le procedure del monitor sono eseguite in mutua esclusione *per definizione*. Più precisamente, definiamo *attiva* una procedura del monitor che sia in esecuzione in seguito all'invocazione da parte di un processo. La specifica del monitor garantisce che in ogni istante non vi possano essere due diverse istanze di procedure del monitor attive contemporaneamente, sia che si tratti di istanze di procedure diverse che della stessa procedura. L'implementazione del monitor, ovviamente, dovrà garantire il rispetto di questa specifica usando opportuni meccanismi di livello più basso (lock, semafori o anche inibizione delle interruzioni).

Il programmatore invece, anche usando i monitor, deve programmare esplicitamente la *sincronizzazione condizionale*, usando le variabili condizionali del monitor, che permettono di sospendere l'esecuzione di un processo nell'attesa che si verifichi una determinata condizione. Tuttavia, anche in questo caso, l'accesso alle variabili può avvenire solo mediante una procedura definita all'interno del monitor e solo usando specifiche operazioni. Più in dettaglio, una variabile condizionale può essere dichiarata all'interno del monitor con un'opportuna sintassi, ad esempio

**cond** nomevar

Il valore di una tale variabile, concettualmente, può essere considerato come una coda di processi bloccati. Le più importanti operazioni del monitor possibili su una tale variabile sono le seguenti:

**empty** (nomevar)

che restituisce *vero* se la coda che rappresenta il valore di nomevar è vuota, *falso* altrimenti.

**wait** (nomevar)

Un processo che esegua questa operazione viene bloccato e viene inserito<sup>13</sup> alla fine della coda associata alla variabile nomevar. Allo stesso tempo, il processo rilascia l'uso del monitor, che quindi può essere usato da (altre procedure di) altri processi. Infine abbiamo l'operazione che permette il risveglio di un processo. Questa è la

**signal** (nomevar)

che risveglia il processo in testa alla coda nomevar, se questa non è vuota, mentre non ha alcun effetto altrimenti. In questo modo, un processo che era stato precedentemente bloccato con una **wait** può essere mandato nuovamente in esecuzione dalla **signal**, secondo una politica di gestione della coda FIFO: il primo processo che è stato bloccato è anche il primo ad essere risvegliato. Il lettore attento a questo punto avrà notato un problema: dato che sia la **wait** che la **signal** sono operazioni usabili solo da procedure definite all'interno del monitor (invocabili dall'esterno), sia il processo che esegue la **signal** che il processo risvegliato che aveva eseguito in precedenza la **wait** si trovano "all'interno del monitor". Questo, come abbiamo detto prima, non è possibile, dato che una sola procedura del monitor alla volta può essere in esecuzione. In generale per risolvere questo problema si usano due tecniche:

**Segnala e continua** Il processo che ha fatto la **signal** continua l'esecuzione, mentre l'altro è risvegliato solo "virtualmente", ossia il suo stato viene cambiato per indicare che può essere nuovamente messo in esecuzione quando potrà entrare nel monitor.

<sup>13</sup>Ovviamente la coda contiene solo dei riferimenti ai processi.

**Segnala e aspetta** In questo caso il processo risvegliato va in esecuzione immediatamente e acquisisce l'accesso al monitor. Il processo che ha fatto la **signal** invece viene bloccato (e in questo caso può essere messo in coda o in testa alla lista dei bloccati).

## 15.5 Scambio di messaggi

Come abbiamo detto in precedenza, nel caso di architetture distribuite, siano esse multicomputer o reti, il modello di comunicazione che più si usa è basato sullo scambio di messaggi. Qui analizzeremo in dettaglio questo modello focalizzandoci sugli aspetti più importanti che lo caratterizzano, ossia i meccanismi per l'identificazione dei partner (o meccanismi di naming) e i meccanismi di invio e ricezione, sia asincroni che sincroni.

### 15.5.1 Meccanismi di naming

I due partner della comunicazione, ovviamente, per potersi "parlare" devono avere a disposizione dei meccanismi che permettano di sapere a chi inviare e da chi ricevere i messaggi. Servono cioè dei meccanismi che permettano di designare, o nominare, i partner stessi. In questo paragrafo ne vedremo tre diversi. Nel seguito chiameremo *mittente* (o *sender*) e *destinatario* (o *receiver*) i due partner della comunicazione e assumeremo che questi siano dei processi (al solito, potrebbero anche essere dei thread o altro ancora).

**Nomi** Il primo meccanismo che esaminiamo è forse quello intuitivamente più semplice: i processi hanno un nome, costituito da un identificatore univoco, e per comunicare fra di loro usano esplicitamente tale nome come argomento delle operazioni di *send* e *receive*. Questo modo di comunicare è stato introdotto nel linguaggio CSP di Hoare e quindi adottato anche in altri casi, quali le librerie PVM e MPI. Così, ad esempio, possiamo avere due processi A e B specificati come segue

```
process A {
    ...
    B!v;
    ...
}
process B {
    ...
    A?y;
    ...
}
```

dove il processo A invia il valore *v* al processo B, che lo riceve nella variabile *y* (! e ? nella notazione CSP indicano le operazioni di *send* e *receive*, rispettivamente).

**Porte** Il secondo metodo per identificare il partner della comunicazione usa le *porte*. Una porta di input concettualmente è un dispositivo che fornisce un punto di entrata nel processo ricevente. Analogamente, una porta di uscita fornisce un punto di uscita per un processo mittente. Il motivo principale per cui sono usate le porte è che permettono, in modo semplice, di condividere uno stesso dispositivo fisico di comunicazione fra più processi. Si pensi, ad esempio, ad un protocollo di comunicazione di livello trasporto quale TCP. Se un processo A vuole stabilire una connessione TCP con un processo che si trova su una macchina il cui indirizzo IP è *pioppo* (ovviamente usiamo nomi simbolici), A deve specificare sia tale indirizzo che il numero *x* (compreso fra 0 e 65535) della porta con la quale vuole connettersi. Questo permette di avere sulla macchina all'indirizzo *pioppo* molti processi diversi, attivi contemporaneamente (supponiamo che la macchina permetta la multiprogrammazione), che usano anche lo stesso dispositivo fisico di comunicazione (ad esempio la stessa porta ethernet). Il processo "giusto", con il quale A vuole comunicare, sarà quello che è in ascolto sulla porta *x*.

Ada è stato uno dei primi linguaggi ad usare le porte mediante le, cosiddette, *entry call*. Un task (l'equivalente di Ada per un processo) può infatti specificare una porta di input come segue:

```
task TipoTask is
    entry portaIn(dato: in integer);
    entry portaOut(dato: out integer);
end TipoTask;
```

In questo modo si specifica che un task di tipo *TipoTask* ha due porte chiamate *portaIn* e *portaOut*. Se poi il corpo del task è definito come segue

```
task body TipoTask is
    ...
    accept portaIn(dato: in integer) do .... end portaIn
    ...
    accept portaOut(dato: out integer) do ... end portaOut
    ...
end TipoTask;
```

Allora il comando *accept* indica che un processo *Pr*, di tipo *TipoTask* può ricevere nella sua porta *portaIn* un messaggio contenente una variabile *dato* di tipo intero. Dopo la ricezione di tale valore verrà eseguito la parte di programma che segue il comando *do*. Analogamente per la porta *portaOut*, con la differenza che in questo caso il parametro *out* specifica una modalità di passaggio dal chiamato (cioè ricevente del messaggio) al chiamante (mittente).

Per spedire un messaggio al processo *Pr*, il processo mittente eseguirà semplicemente un comando della forma

```
Pr.portaIn(D);
```

dove *D* è una variabile contenente il valore che vogliamo trasmettere. Si noti che ogni specifico processo ha le proprie porte, individuate dalla dichiarazione del tipo del processo e dal nome del processo. Quindi i messaggi inviati come visto sopra giungono ad un solo processo.

**Canali** L'ultimo metodo che vediamo per stabilire una comunicazione usa i *canali*. Questi apparvero per la prima volta, a livello di linguaggio di programmazione, in Occam e successivamente sono stati ampiamente usati in vari modelli teorici (ad esempio il  $\pi$ -calcolo) e in vari linguaggi. Un canale, concettualmente, è un'astrazione di una qualsiasi struttura di comunicazione che permetta di collegare i due partner. In questo senso anche una variabile condivisa o un monitor possono essere considerati come un'implementazione di un canale. Normalmente però il termine canale è riservato a quei modelli che sono astrazioni di una vera rete di comunicazione, nella quale si abbiano effettivamente canali fisici che collegino dispositivi separati. Nel seguito noi non faremo alcuna ipotesi su come sono implementati i canali ma ci limiteremo ad un uso astratto di questo termine.

Volendo essere un pò più precisi ma, al solito, senza riferirsi ad alcun linguaggio particolare, possiamo vedere il codice

```
channel nomeCh (Type)
```

come la dichiarazione di un canale nel quale l'identificatore *nomeCh* indica il nome del canale, mentre *Type* indica il tipo dei messaggi spediti sul canale. Ovviamente, la limitazione ad un campo è fatta per motivi di semplicità: possiamo avere più campi diversi, di tipo diverso, identificati da opportuni nomi.

Il mittente ed il ricevente per poter comunicare dovranno conoscere entrambi il nome di un canale. Facendo riferimento alla precedente dichiarazione la comunicazione avviene inviando dati al canale *nomeCh*, con l'operazione

```
send nomeCh(dati)
```

effettuata dal mittente, e ricevendo dati dal canale, con l'operazione

```
receive nomeCh(var)
```

eseguita dal destinatario. *dati* e *var* qui sono due variabili, di tipo *Type*: la prima contiene i valori da trasmettere mentre la seconda conterrà i valori ricevuti, dopo che la comunicazione è avvenuta.

Nella pratica vi sono molti modelli di canali diversi, che corrispondono ai diversi meccanismi di comunicazione esistenti. In particolare, possiamo distinguere i canali *monodirezionali*, che permettono il flusso dei dati in una sola direzione, da quelli *bidirezionali*, che invece lo permettono in entrambe.

Inoltre va notato anche che, sempre a livello di terminologia, un canale monodirezionale che connette un mittente ad un destinatario è detto *link*, mentre a volte viene chiamato *input port* un canale (monodirezionale) che connette più mittenti ad un unico destinatario. Infine si parla di *mailbox* quando si ha una comunicazione molti a molti, dove più mittenti possono spedire a più destinatari. Nel seguito noi assumeremo che i canali siano di questo tipo, a meno che non sia diversamente specificato.

Dal punto di vista della semantica un canale può essere visto come una coda di messaggi che sono stati spediti e non sono ancora stati ricevuti. La lunghezza di tale coda dipende dal fatto che si abbia comunicazione asincrona o sincrona, come vedremo nelle prossime due sezioni.

### 15.5.2 Comunicazione asincrona

In questo e nel prossimo paragrafo analizziamo le due modalità principali di comunicazione mediante scambio di messaggi. Lo facciamo riferendoci ad una struttura di comunicazione che usa i canali, comunque le considerazioni che seguono sostanzialmente valgono anche quando si usano i nomi esplicativi di processi o le porte.

Nella comunicazione asincrona il mittente, dopo aver inviato un messaggio, prosegue nella computazione senza aspettare che il destinatario abbia effettuato l'operazione di ricezione. In altri termini, con la comunicazione asincrona l'operazione

```
send nomeCh(dati)
```

non è bloccante: l'effetto di tale operazione infatti è quello di aggiungere un messaggio, contenente il valore della variabile dati alla lista dei messaggi associata al canale nomeCh, dopo di che il processo che ha effettuato la send può continuare la sua computazione. Assumiamo qui che l'operazione di accesso al canale sia atomica e anche che l'operazione di invio del messaggio sia affidabile (cioè i messaggi inviati sono effettivamente ricevuti e senza errori)<sup>14</sup>. Assumiamo anche che la coda del canale sia gestita in modo FIFO, il primo messaggio inviato è dunque il primo ad essere ricevuto. Ovviamente la correttezza di queste assunzioni dipende dalla struttura sottostante che implementa i canali. Si noti inoltre che, almeno in linea di principio, con la comunicazione asincrona la coda associata al canale ha una lunghezza non limitata, dato che possono essere inviati messaggi in numero arbitrariamente grande prima che sia effettuata una ricezione.

La ricezione, come accennato poc'anzi, è effettuata mediante un'operazione

```
receive nomeCh(var)
```

L'effetto di questa operazione è il seguente. Se la coda associata al canale nomeCh non è vuota viene prelevato e rimosso il primo messaggio della coda. I dati relativi a tale messaggio vengono assegnati<sup>15</sup> alla variabile var (anche in questo caso, assumiamo che l'accesso alla coda sia realizzato mediante un'operazione atomica). Se invece la coda dei messaggi è vuota, allora l'operazione receive si blocca ed il processo che l'ha effettuata viene bloccato fino a quando non ci sia almeno un messaggio nella coda. Quando questa condizione si verifica il processo bloccato viene risvegliato, secondo le modalità già descritte in precedenza quando abbiamo visto la sincronizzazione basata sul blocco. Dato che l'operazione receive è sempre bloccante, in alcuni casi si vuole dare al processo la possibilità di controllare se il canale è vuoto o meno, per evitare di effettuare una receive sul canale vuoto. Oppure, nel caso in cui si sia effettuata una receive su di un canale

<sup>14</sup>L'assunzione sull'atomicità dell'operazione sul canale implica che, per un breve periodo di tempo (dipendente da come è implementato il canale), il processo mittente debba attendere prima di proseguire. Qui ignoriamo quest'attesa.

<sup>15</sup>Chiaramente qui usiamo questo termine in modo informale, dato che potremmo avere linguaggi imperativi, ma anche funzionali o logici.

vuoto, si vuole limitare il tempo di attesa mediante un time-out che la interrompa dopo un prefissata durata. Specifici costrutti per trattare questi casi sono forniti in alcuni linguaggi. Inoltre in alcuni casi l'operazione di ricezione di un messaggio non avviene esplicitamente usando una primitiva di comunicazione ma è implicita. Questo caso sarà considerato nel Paragrafo 15.5.4 quando parleremo della Remote Procedure Call.

**Esempio 15.1** Come esempio di uso della comunicazione asincrona vediamo una semplice interazione fra due client e un server. Ogni client invia una richiesta al server specificando il nome del client (1 o 2) e comunicando dei dati da elaborare (di tipo char). Il server risponde al client inviando i dati elaborati.

```
channel richiesta (int client, char dati);
channel risposta1 (char ris);
channel risposta2 (char ris);

process Client1{
    char valori;
    char risultati;
    ...
    send richiesta(1, valori);
    receive risposta1(risultati);
    ...
} // Definizione dei valori

process Client2{
    char valori;
    char risultati;
    ...
    send richiesta(2, valori);
    receive risposta2(risultati);
    ...
} // Uso dei risultati

process Server{
    int cliente;
    char valori;
    char risultati;
    ...
    while true {
        receive richiesta (cliente, valori);
        if cliente = 1 then
            ...
            // Elaborazione valori
            send risposta1(risultati);
        }
        if cliente = 2 then
            ...
            // Elaborazione valori
            send risposta2(risultati);
    }
}
```

Nell'esempio precedente vi sono alcune cose da notare. Innanzitutto il canale richiesta è usato da entrambi i client per mandare delle richieste al server.

Quest'ultimo invece manda le riposte usando un canale per ogni client. Inoltre i nomi dei canali sono statici e globali a tutto il programma, quindi la correttezza della comunicazione dipende dal corretto uso degli stessi. Il processo Client1, ad esempio, potrebbe usare il canale `risposta2`, che dovrebbe essere riservato alle risposte per il secondo client, senza che vi sia alcun controllo su questo. Per evitare possibili errori, ed anche per motivi di riservatezza, ogni client potrebbe spedire al server anche un nome di un proprio canale locale, non visibile agli altri client, sul quale il server potrebbe inviare la risposta. Questo però richiederebbe una gestione dinamica dei nomi che in generale non è semplice. Infine, ovviamente, l'esempio è fatto per due client per motivi di semplicità. È ovvio come esso possa essere esteso a  $n$  client, noti staticamente, usando un array di  $n$  canali. È invece più complesso estendere l'esempio per gestire un numero di client non noto a priori (anche in questo caso servirebbe una gestione dinamica dei nomi per permettere ad ogni client di comunicare al server il nome del proprio canale).

### 15.5.3 Comunicazione sincrona

Nella comunicazione (con scambio di messaggi) sincrona la `receive` è come nel caso asincrono. Cambia invece l'operazione di `send` che diventa anch'essa bloccante. Il mittente, cioè, dopo aver effettuato l'invio di un messaggio viene bloccato fino a quando il destinatario non effettui la ricezione dello stesso. Quindi, almeno dal punto di vista concettuale, nel caso sincrono possiamo vedere la `send` e la `receive` come due operazioni che permettono ai processi di sincronizzarsi nel momento in cui viene effettuata la comunicazione. Dal punto di vista reale, ovviamente, le operazioni avvengono in momenti diversi e il mittente, dopo aver effettuato la `send`, aspetta un segnale che confermi la ricezione del messaggio. La comunicazione sincrona è apparsa per la prima volta in CSP.

Il principale vantaggio della comunicazione sincrona, rispetto a quella asincrona, è nel fatto che possiamo limitare la lunghezza massima della coda dei messaggi associata ad un canale. Difatti, ogni processo che abbia spedito un messaggio non può spedirne un altro fino a quando quello precedente non è stato ricevuto e rimosso dalla coda. Quindi la coda conterrà al più un messaggio per ogni processo mittente. Un altro vantaggio della comunicazione sincrona è nella gestione degli errori. Nel caso in cui la ricezione di un messaggio causi un errore, se il mittente è bloccato in attesa del completamento della comunicazione è più facile gestire la situazione anomala rispetto al caso in cui il mittente abbia proseguito nella propria computazione.

A fronte di questi aspetti positivi la comunicazione sincrona presenta anche degli svantaggi. Innanzitutto la concorrenza fra i vari processi viene ad essere ridotta a causa della `send` bloccante: dato che operazioni di `send` e `receive` sono effettuate in momenti diversi, quando avviene una comunicazione c'è sempre almeno un processo che viene bloccato. Questo non avviene nel caso della comunicazione asincrona, se la `send` è fatta prima della `receive`. Questa limitazione può essere seria nelle applicazioni reali. Supponiamo ad esempio, secondo un tipico pattern di interazione concorrente, di avere un processo *produttore* che

spedisce i dati ed un *consumatore* che li riceve. Se i due processi hanno tempi di elaborazione diversi, non c'è alcuna garanzia sull'ordine nel quale saranno eseguite la `send` e la `receive`. Comunque sia, ad ogni comunicazione sincrona abbiamo un ritardo dovuto alla sospensione di uno dei processi. Se invece si usasse la comunicazione asincrona il ritardo totale sarebbe minore, in quanto la coda associata al canale funzionerebbe da buffer e permetterebbe di gestire le velocità diverse: quando il produttore è più veloce del consumatore i messaggi sono messi in coda nel canale, nel caso opposto invece vengono prelevati dal consumatore. Un simile comportamento lo possiamo ottenere anche con la comunicazione sincrona, interponendo un processo buffer fra produttore e consumatore, tuttavia anche in questo caso l'efficienza è inferiore a quella del caso asincrono.

In generale, efficienza a parte, possiamo simulare la comunicazione asincrona mediante quella sincrona usando un processo che realizzi il canale asincrono e che risponda immediatamente sia ai mittenti che ai destinatari del canale stesso. È possibile anche fare il contrario, dato che possiamo sempre fare seguire ad una `send` non bloccante una `receive` con la quale ricevere dal destinatario la notifica della ricezione del messaggio, realizzando così una comunicazione sincrona.

Un secondo svantaggio della comunicazione sincrona è nella maggiore difficoltà di utilizzo per il programmatore. O meglio, il programmatore deve prestare un maggiore attenzione perché, con la comunicazione sincrona, è più facile generare deadlock ed altri errori. A titolo di esempio consideriamo due processi che comunichino eseguendo entrambi sia `send` che `receive` (questo avviene in quasi tutte le applicazioni peer to peer). Se entrambi i processi effettuano una `send` si genera una situazione di deadlock nella quale la computazione non può ulteriormente procedere, dato che ogni processo è bloccato in attesa di una comunicazione (`receive`) da parte dell'altro. Un simile errore non si avrebbe avuto con la comunicazione asincrona, dato che in questo caso la `send` non è bloccante (ovviamente si possono avere altre situazioni di deadlock che si verificano anche con la comunicazione asincrona).

### 15.5.4 Chiamata di procedura remota e rendez-vous

In questo paragrafo vediamo due ulteriori meccanismi di comunicazioni abbastanza usati nella pratica soprattutto nelle interazioni di tipo client-server: la chiamata di procedura remota (o RPC, dall'inglese Remote Procedure Call), usata in molti linguaggi fra cui Java, e il rendez-vous, introdotto ed estensivamente usato in ADA. In prima approssimazione entrambi questi meccanismi possono essere visti come metodi di comunicazione sincrona mediante scambio di messaggi. Tuttavia, a differenza della `send` sincrona vista nel paragrafo precedente, qui il mittente, dopo aver inviato un messaggio, prima di procedere nella computazione aspetta un messaggio di risposta del destinatario (e non la sola conferma del fatto che il suo messaggio è stato ricevuto). Con più precisione, e come suggeriscono i nomi di queste operazioni, sia la RPC che il rendez-vous sono usati per invocare una procedura che è eseguita in remoto, tipicamente da un processo che "gira"

su un'altra macchina, anche geograficamente distante. La differenza fra RPC e rendez-vous sta nel modo con cui viene gestita la chiamata in remoto.

Nel caso della RPC nel destinatario non esiste un processo attivo che aspetta la chiamata e che, al suo arrivo, effettua la ricezione della stessa. Al momento della chiamata quindi viene creato, nel destinatario, un nuovo apposito processo per gestirla. Questo processo si occuperà di passare i parametri, eseguire il corpo della procedura chiamata e trasmettere i risultati al chiamante. Fatto questo il processo termina la propria esecuzione ed eventuali nuove chiamate dovranno essere gestite da nuovi processi. Ovviamente la sottostante struttura di comunicazione deve permettere la rilevazione della presenza della chiamata usando meccanismi basati su porte, canali o altro. In termini di scambio di messaggi dunque possiamo dire che, nel caso della RPC, viene fatta una operazione di invio esplicita, chiamata `call`, mentre non viene fatta alcuna operazione di ricezione esplicita.

Con il rendez-vous invece il processo chiamante effettua, appunto, un rendez-vous con un processo già esistente ed attivo nel destinatario. Quando la chiamata arriva tale processo destinatario esegue la procedura richiesta (previo passaggio dei parametri, al solito), trasmette i risultati al chiamante e poi continua nella propria attività, ad esempio per gestire nuove richieste. Quindi, in questo caso oltre alla `call` nel mittente esiste anche un'operazione di ricezione esplicita nel destinatario, quale ad esempio la `accept` in Ada, già vista nel Paragrafo 15.5.1. Analogamente al caso della `receive`, la `accept` si sospende se non ci sono richieste (`call`) da gestire.

Nel seguito vediamo qualche ulteriore dettaglio sulla RPC, mentre per il rendez-vous rimandiamo alla letteratura specializzata.

Perché sia possibile eseguire una chiamata di procedura remota il processo che detiene il codice della procedura deve esportare all'ambiente esterno almeno il nome della procedura stessa, in modo che questo sia visibile ed usabile per la chiamata. Normalmente questo avviene in un contesto di programmazione che usa i moduli e nel seguito faremo dunque anche noi l'assunzione di avere questi costrutti. Un modulo è dunque un componente (di un programma) che contiene sia processi che procedure esportabili. Queste sono definite esplicitamente nel modulo stesso e sono distinte dalle procedure locali che possono essere utilizzate solo all'interno del modulo. Inoltre, anche i processi di un modulo saranno distinti in processi puramente locali, detti di background, e processi che vengono creati per servire le invocazioni di procedure dall'esterno. Moduli diversi possono risiedere su macchine diverse e possono comunicare fra di loro invocando procedure remote. Un modulo quindi, per noi, ha la seguente struttura:

```
module NomeM{
    op nome1(tipo formali) returns(tipo risultati)
        // intestazioni
    ...
    body {
        variabili locali;
        codice di inizializzazione;
        proc nome1(formali) returns(risultati){
            corpo procedura;
            // implementazione
        }
    }
}
```

```
...
procedure locali;
processi locali;
}
```

Come si può vedere, nella parte iniziale vengono elencate i nomi delle procedure esportabili con i nomi ed i tipi dei parametri formali e dei risultati restituiti. Nel corpo poi, con la parola chiave `proc`, per ogni procedura così definita viene fornito il corpo (si noti che qui non viene ripetuto il tipo dei parametri formali e dei risultati, ma viene fornito solo il loro nome).

Facendo riferimento al codice precedente, un altro modulo può chiamare la procedura `Nome1` del modulo `NomeM` usando il comando `call` nel quale siano specificati questi due nomi ed i parametri attuali:

```
call NomeM.nome1(Attuali)
```

Come accennato in precedenza, una simile chiamata causa la creazione di un nuovo processo il quale, dopo aver ricevuto i parametri attuali, esegue il corpo della procedura `nome1`. Quando questa termina il processo manda i risultati al chiamante e poi termina anch'esso. Il chiamante, che si era bloccato dopo l'esecuzione della `call`, una volta ricevuti i risultati può continuare la propria esecuzione.

Si noti che i vari processi attivi in uno stesso modulo, sia quelli di background che quelli che stanno eseguendo chiamate remote, se eseguiti concorrentemente devono usare opportuni meccanismi di sincronizzazione e di mutua esclusione, dato che questi non sono in alcun modo forniti dalla RPC. Alternativamente si può assumere che, in ogni modulo, vi sia al massimo un processo alla volta attivo, in modo analogo a quello che avviene con un monitor.

Come ultima nota, osserviamo che in molti casi la possibilità di usare RPC, se non supportata direttamente dal linguaggio, è ottenuta aggiungendo ad un linguaggio esistente dei, cosiddetti, *stub* che permettono di rendere la chiamata di procedura remota il più possibile simile ad una chiamata di procedura "normale", nascondendo al programmatore i dettagli relativi alle comunicazioni necessarie per gestire la RPC. Gli stub di solito vengono generati automaticamente da uno *stub compiler*. Quando viene effettuata una chiamata remota, in realtà viene chiamato uno stub che prende tutti i parametri necessari all'individuazione esatta dell'operazione (nome della procedura chiamata, parametri attuali ecc.) costruisce un opportuno messaggio che li contiene e invia il messaggio ad un altro stub (detto a volte *skeleton*) che si trova presso destinatario. Il secondo stub riceve il messaggio, estrae i parametri e quindi chiama la procedura locale con la conseguente creazione di un nuovo processo. Quando l'esecuzione è terminata, il secondo stub, in modo duale rispetto al primo, crea un messaggio con i risultati, lo trasmette al primo che infine estrae i risultati e li trasmette al processo che aveva invocato la RPC.

## 15.6 Non determinismo e composizione parallela

In questa sezione vediamo due aspetti fondamentali nella specifica di computazioni concorrenti. Iniziamo dal non determinismo.

I costrutti per il controllo della computazione che abbiamo visto nei precedenti capitoli, ed in particolare nel Capitolo 8, hanno tutti una caratteristica comune: sono deterministici. Questo significa che in ogni momento della computazione, per un dato stato corrente, la prossima istruzione da eseguire è univocamente determinata e non vi è mai a possibilità di scegliere una fra due o più alternative egualmente possibili. Si pensi, ad esempio, al comando di scelta condizionale `if then else`: la condizione che viene valutata è o vera o falsa, e a seconda del valore si sceglie una delle due alternative. Non si dà mai il caso in cui entrambi i rami `then` ed `else` siano egualmente possibili come continuazione della computazione. In altri termini in una computazione deterministica lo stato iniziale, in assenza di ulteriori input esterni, determina in modo univoco quello finale.

Al di là di considerazioni di tipo filosofico sulla natura ultima dei nostri dispositivi di calcolo, la visione deterministica permette di semplificare il modello di computazione. Tuttavia vi sono situazioni, soprattutto in ambito concorrente, nelle quali tale visione è limitativa. Si pensi ad esempio ad un server che, usando una comunicazione a scambio di messaggi, riceva da un client delle richieste di lettura e da un altro client delle richieste di scrittura (ad esempio per l'accesso ad un file condiviso). Le richieste possono arrivare in un ordine qualsiasi e a momenti impredicibili. Inoltre, per motivi di riservatezza, i canali usati dai due client per inviare le richieste devono essere diversi. Seguendo lo schema del programma visto nel Paragrafo 15.5.2 potremmo realizzare il server come segue:

```

channel lettura (int dati);
channel scrittura (int dati);
process ClientLettura{
    int risultati;
    ...
    send lettura(risultati);
    ...
    // Uso di risultati
}
process ClientScrittura{
    int valori;
    ...
    // Definizione di valori
    send scrittura(valori);
    ...
}
process Server{
    int risultati;
    int valori
    while true {
        receive lettura (risultati);
        servi richiesta lettura;
        receive scrittura (valori);
        servi richiesta scrittura;
    }
}

```

Tuttavia questa soluzione non funziona bene. Difatti, avendo imposto un ordine per la ricezione delle operazioni di lettura e scrittura, è evidente che se il server effettua la `receive` sul canale `lettura` ma invece è stata fatta una richiesta di scrittura si crea una situazione di attesa inutile. L'attesa potrebbe anche diventare un deadlock nel caso in cui la `send` sul canale `lettura` non venga più fatta (ad esempio perché il processo corrispondente ha interrotto la propria esecuzione).

Per risolvere questo tipo di problemi la soluzione più naturale ed elegante è quella di usare un costrutto che permetta di introdurre il non determinismo nella scelta fra più alternative. Una delle prime proposte in questo senso, degli anni '70 e ad opera del solito Dijkstra, è il *comando con guardia* (guarded command) che ha la seguente struttura:

```

condizione -> comando
[ ]
condizione -> comando
[ ]
...
condizione -> comando

```

Il significato di questo comando è il seguente. La condizione (detta anche guardia) a sinistra della freccia è un'espressione booleana che viene valutata. Se la condizione è vera allora il ramo corrispondente può essere scelto e può essere eseguito il comando a destra della freccia. Se vi sono più condizioni vere, e quindi più rami che possono essere eseguiti, allora uno di essi è scelto in modo non ulteriormente specificato, cioè in modo non deterministico. Evidentemente, in presenza di tale non determinismo, dato uno stato iniziale la computazione può raggiungere più stati finali diversi.

In una versione più recente di questo comando la guardia oltre ad una condizione booleana contiene anche un comando di comunicazione che si può anche bloccare: una guardia così può fallire, se la condizione booleana ha valore falso, può avere successo, se la condizione booleana ha valore vero e il comando di comunicazione può essere eseguito senza sospensione, può infine bloccarsi se il comando di comunicazione si blocca. Se tutte le guardie sono bloccate allora tutto il comando viene bloccato, mentre nel caso in cui vi sia almeno una guardia che abbia successo, in modo analogo a quanto visto prima, viene scelto, in modo non deterministico, un ramo fra tutti quelli le cui guardie hanno successo. Se tutte le guardie falliscono il comando non ha alcun effetto.

Usando un tale comando possiamo programmare il nostro server come segue

```

... % come sopra

process Server{
    int v;
    int r;
    while true{
        receive lettura(v) -> servi richiesta lettura
        []
        receive scrittura (r) -> servi richiesta scrittura
    }
}

```

È evidente che in questo modo si eliminano i problemi visti poc' anzi, perché appena una delle due richieste viene fatta questa può essere servita senza dover aspettare l'altra.

Il discorso sul non determinismo e sul suo potere espressivo ci porterebbe ben oltre gli scopi di questo testo. Qui osserviamo soltanto che, a livello di implementazione, servono opportune tecniche, non banali (ad esempio generatori di numeri pseudo causali) per gestirlo in modo corretto. Osserviamo anche che quando un comando con guardia è usato nel contesto di un loop infinito, come nell'esempio del server, si pone anche un problema di fairness: se una guardia è sempre verificata vorremmo che il ramo corrispondente prima o poi venisse scelto. Anche in questo caso, varie nozioni di fairness e varie soluzioni sono state proposte.

Concludiamo questo paragrafo con un altro esempio che mostra, ancora di più del precedente, l'utilità del non determinismo. Supponiamo di avere due processi che producano due stream di dati che vogliamo fondere in un unico stream. Il processo che realizza questa "fusione" è un particolare tipo di filtro, detto merge (in generale, un filtro è un processo che riceve i dati da uno o più canali di input e li invia a uno o più canali di output). Gli stream che consideriamo sono sequenze, anche infinite, di dati, quindi non è possibile ricevere prima tutti i dati di un canale e poi tutti quelli dell'altro. Inoltre, come avviene spesso nella pratica, i singoli dati su ogni canale sono inviati a intervalli irregolari, senza alcuna sincronizzazione e comunque senza la possibilità di fare ipotesi sul momento del loro arrivo. Usando il comando con guardia il nostro processo merge può essere programmato come segue:

```
channel input1 (int dati);
channel input2 (int dati);
channel output (int dati);

process Server{
    int dati;
    while true {
        receive input1(dati) -> send output(dati)
        []
        receive Input2(dati) -> send output(dati)
    }
}
```

Si osservi che, in questo modo, appena uno dei due canali in input produce un dato questo è inviato in output senza dover aspettare l'altro canale. Nel caso limite in cui su di un canale non arrivino più dati il processo continua comunque a trasmettere i dati dell'altro canale. Questo non sarebbe stato possibile imponendo un ordinamento statico fra le ricezioni sui due canali di input. Infine, ovviamente, il processo merge può essere arbitrariamente complicato inserendo delle elaborazioni su dati ricevuti o aumentando il numero dei canali.

### 15.6.1 La composizione parallela

Il secondo costrutto importante per la definizione di computazioni concorrenti è l'operatore di composizione parallela, spesso indicato con  $\parallel$ <sup>16</sup>, che abbiamo visto nel programma per i filosofi a cena.

Intuitivamente quest'operatore specifica che i processi che costituiscono i suoi operandi debbano essere eseguiti in parallelo, ossia in modo concorrente. Tuttavia dal punto di vista semantico vi sono varie interpretazioni possibili per questo. L'interpretazione adottata nel caso della programmazione parallela è quella che in ambito teorico è usualmente detta del *massimo parallelismo*: tutti i processi che compaiono nella composizione parallela e che possono procedere nella computazione, lo fanno. Ovviamente questo presuppone un'architettura sottostante multiprocessore ed anzi, nel caso in cui vi sia creazione dinamica dei processi, in linea di principio richiede un modello architettonicale con un numero non limitato di processori.

Per questo, non volendo fare alcuna assunzione sull'architettura sottostante, si usa più spesso un modello semantico diverso dell'operatore  $\parallel$ , detto a *interleaving*. Secondo questo modello ad ogni istante uno solo dei processi può procedere nella computazione, eventualmente alternando le proprie azioni con quelli di altri processi che appaiono nella composizione parallela (to interleave significa appunto "alternare in livelli diversi"). Così, se il processo  $P$  può fare (una computazione descritta da) una sequenza di azioni  $s_1$  ed il processo  $Q$  può fare la sequenza  $s_2$ , il processo  $P \parallel Q$  può fare una qualsiasi sequenza ottenibile alternando in modo arbitrario le azioni di  $s_1$  e  $s_2$ , purché sia rispettato l'ordine dato da  $s_1$  e  $s_2$  stesse. Come esempio concreto, se  $P$  può fare le sequenze di azioni  $ab$ <sup>17</sup> e  $ac$ , mentre  $Q$  può fare solo  $d$ , allora  $P \parallel Q$  può fare le sequenze  $dab$ ,  $adb$ ,  $dac$ ,  $adc$  e  $acd$ . Si noti che l'interleaving introduce un ulteriore livello di non determinismo nella computazione, dato che se anche  $P$  e  $Q$  sono processi deterministici (che hanno ognuno una sola computazione possibile per un dato input), la composizione parallela  $P \parallel Q$  può avere comunque più computazioni possibili a seconda di come si scelga di alternare le azioni dei due processi (si consideri  $R \parallel Q$ , dove  $Q$  è quello di prima e il processo  $R$  può fare solo la sequenza  $ab$ ).

## 15.7 La concorrenza in Java

In questo paragrafo cercheremo di concretizzare alcuni dei concetti visti in questo capitolo esaminando il caso di un linguaggio reale. Ovviamente presenteremo solo alcuni aspetti principali. Per una comprensione approfondita della concorrenza in Java esistono testi specifici riportati in bibliografia.

<sup>16</sup>Quest'operatore è binario ma essendo associativo e commutativo possiamo omettere le parentesi in caso di più occorrenze dello stesso.

<sup>17</sup>Qui  $ab$  indica la sequenza nella quale viene fatta prima l'azione  $a$  e poi l'azione  $b$ .

### 15.7.1 Creazione dei thread

In Java un programma sequenziale contiene un unico thread che serve per eseguire il metodo *main* del programma. Se invece vengono creati ed eseguiti più thread nell'ambito di uno stesso programma allora abbiamo una forma di programmazione concorrente (multithreaded, appunto).

Un thread è un oggetto della classe *Thread* fornita dal package *java.lang*. Per creare un thread possiamo semplicemente creare un oggetto di questa classe, ad esempio dichiarando:

```
Thread pippo = new Thread();
```

Una volta creato un oggetto *Thread* ed eventualmente definiti alcuni parametri iniziali (ad esempio la priorità), il nuovo thread viene eseguito invocando il metodo *start()*, definito nella classe *Thread*, come nel codice seguente:

```
pippo.start();
```

Più in dettaglio, il metodo *start()* genera un nuovo thread del controllo utilizzando i dati forniti nell'oggetto *pippo*, invoca il metodo *run* (anch'esso definito nella classe *Thread*) per l'oggetto *pippo* e quindi termina. A questo punto il nuovo thread viene reso attivo dalla Java Virtual Machine che esegue *run*. Si noti che il metodo *run* non è accessibile direttamente al programmatore, che lo può attivare solo attraverso *start*. Inoltre, per ogni oggetto di tipo *Thread* si può invocare una sola volta il metodo *start*.

Lasciando così le cose tuttavia non accade nulla, perché il metodo *run* definito nella classe *Thread* non fa niente. Per avere un comportamento utile dovremo definire un opportuno metodo *run* per la nostra applicazione. Questo può essere fatto in due modi.

Il primo è quello di estendere la classe *Thread* mediante una sottoclasse nella quale ridefiniamo il metodo *run*. Ad esempio, se facciamo

```
class MioThread extends Thread{
    public void run() {
        System.out.println("Sono il thread pippo");
    }
}
Thread pippo = new MioThread();
pippo.start();
```

in questo caso il thread *pippo* viene eseguito ed il suo metodo *run* stampa la stringa di caratteri.

L'altro modo per definire un comportamento utile del metodo *run* è quello di usare l'interfaccia *Runnable*. Questa interfaccia, fornita anch'essa dal package *java.lang*, rappresenta in astratto un compito, un lavoro che deve essere eseguito mediante il metodo *run*, dichiarato nell'interfaccia stessa come segue:

```
public void run();
```

Tale lavoro dovrà essere eseguito da uno specifico thread che, continuando nella metafora, può essere visto come chi esegue il lavoro. perché questo sia possibile dovremo innanzitutto definire una classe C che implementi *Runnable* e

quindi fornisca uno specifico codice per *run*. Poi, dovremo creare un oggetto di classe C in modo da definire una specifica istanza del nostro lavoro. L'oggetto così creato dovrà quindi essere passato ad uno specifico oggetto della classe *Thread* mediante il costruttore di questa classe. Infine il lavoro può iniziare usando il metodo *start*, come già visto.

L'esempio di prima, realizzato usando l'interfaccia *Runnable*, diventa quindi il seguente:

```
class MioThread implements Runnable{
    public void run() {
        System.out.println("Sono il thread pippo");
    }
}
Runnable esec = new MioThread();
Thread pippo = new Thread(Esec);
pippo.start();
```

Si noti che *esec* ha tipo statico *Runnable* e tipo dinamico *MioThread* e che tale oggetto è passato al costruttore della classe *MioThread* per poter permettere l'esecuzione del metodo *run*. Tale metodo, come prima, scrive la stringa quando il thread è attivato dal metodo *start*.

Il vantaggio di questa seconda modalità di definizione dei thread è nella possibilità di avere più interfacce implementate da una singola classe. Questo permette di definire una classe che ha oggetti eseguibili (in quanto implementazione di *Runnable*) e allo stesso tempo implementa altre interfacce che definiscono altre caratteristiche degli oggetti stessi. Questo con il primo approccio non è possibile perché in Java, come sappiamo, una classe può estendere al più un'altra classe. Inoltre la classe *Thread*, che implementa anch'essa *Runnable*, spesso contiene molto più di quello di cui si ha effettivamente bisogno per eseguire un thread.

### 15.7.2 Scheduling e terminazione dei thread

Normalmente nei sistemi multithreaded vi sono più thread che processori per cui è necessario definire delle politiche di *scheduling* che permettano di suddividere opportunamente il tempo di CPU fra i vari thread in esecuzione. Tali politiche nel dettaglio dipendono dalla particolare implementazione della Java Virtual Machine e dalla piattaforma sulla quale stiamo eseguendo le nostre applicazioni Java. In generale possiamo dire che ad ogni thread è associata una *priorità* che è usata dal sistema a run-time per determinare qual è il prossimo thread da eseguire. I thread con priorità più alta normalmente sono eseguiti prima di quelli con priorità più bassa, tuttavia vi sono anche meccanismi che limitano il tempo di CPU usabile da un singolo thread per evitare la cosiddetta *starvation* dei thread con priorità più bassa, ossia per evitare che quest'ultimi, in presenza di thread eseguibili di priorità maggiore, non siano mai eseguiti. Un thread in esecuzione, quindi, continua ad usare la CPU fino a quando o esegue un'operazione bloccante che, come abbiamo già visto, causa il rilascio della CPU, oppure fino a quando riceve dal sistema un comando di interruzione dell'esecuzione, operazione normalmente chiamata *preemption*. Tale interruzione avviene o perché un thread con priorità maggiore

è divenuto pronto per essere eseguito oppure perché lo scheduler del sistema ha deciso che il thread in esecuzione ha superato la sua quota di utilizzo della CPU. I thread possono anche rilasciare volontariamente la CPU usando specifici metodi quali `sleep`.

La priorità di un thread inizialmente è la stessa del thread che lo ha creato e può essere cambiata usando il comando `setPriority` per impostare un valore compreso fra le costanti `MIN_PRIORITY` e `MAX_PRIORITY`. L'uso di questo comando deve essere fatto tenendo ben presente che l'esecuzione effettiva dei vari thread dipende, come detto prima, oltre che dalla priorità anche dalla politica usata dallo scheduler. Quindi se ha certamente senso assegnare una priorità più alta ad un thread che vogliamo eseguire prima di un altro (ad esempio, quello corrispondente a una funzionalità che interrompa la normale esecuzione della nostra applicazione), non dobbiamo tuttavia pensare che i valori espressi dalle priorità siano assoluti.

Un thread termina e dunque rilascia definitivamente le risorse di calcolo che sta usando quando termina l'esecuzione del metodo `run`. Vi sono tuttavia anche altre modalità di terminazione.

Innanzitutto, usando il metodo `interrupt`, si può inviare ad un thread un segnale di interruzione che, anche se non causa direttamente la terminazione di un thread attivo, può essere opportunamente gestito nel programma del thread che lo riceve per causare la terminazione. Ad esempio, se abbiamo due thread *t1* e *t2* possiamo avere in *t1* il codice

```
t2.interrupt()
```

mentre nel thread *t2* abbiamo

```
...  
while (!interrupted())  
...  
}
```

In questo modo, quando *t1* lancia l'interrupt il ciclo while viene interrotto e il thread *t2* può terminare (se non c'è altro da fare dopo il while). Si noti che un interrupt causa l'interruzione del periodo di attesa di un thread bloccato. Inoltre se il thread sta eseguendo un metodo `sleep` o `wait` quando riceve una richiesta di interruzione, allora viene lanciata una `InterruptedException` che deve essere opportunamente gestita.

Un secondo aspetto della terminazione dei thread riguarda il fatto che, normalmente, in un'applicazione multithreaded abbiamo un thread corrispondente al `main` e altri thread che sono lanciati da questo ed eseguiti concorrentemente. Se il `main` termina quello che accade agli altri thread dipende dalla loro natura. Distinguiamo infatti i *thread di utente* (user thread) dai *thread demoni* (daemon thread): i primi sono quelli creati dal `main`, se non diversamente specificato; i secondi invece sono quelli definiti esplicitamente tali mediante il metodo `setDaemon(true)` e quelli creati da altri `thread demoni`. I *thread di utente* continuano la loro esecuzione anche se il `main` ha terminato mentre *thread demoni* vengono fatti terminare immediatamente quando non ci sono più *thread di utente* attivi.

### 15.7.3 Sincronizzazione e comunicazione fra thread

Come abbiamo visto nelle sezioni precedenti, in presenza di thread concorrenti sono necessari opportuni meccanismi di sincronizzazione e di comunicazione fra thread per evitare accessi non corretti a dati condivisi. La soluzione adottata in Java è simile a quella che abbiamo visto nel caso dei monitor anche se vi sono alcune specificità dovute alla presenza di un paradigma orientato agli oggetti.

I dati condivisi per i quali vogliamo garantire la mutua esclusione in Java sono normalmente gli oggetti (ma vi sono anche i campi statici). Associato ad un singolo oggetto abbiamo un *lock* che, per garantire la sincronizzazione, deve essere acquisito prima dell'uso dell'oggetto e rilasciato dopo. Tuttavia, come nel caso dei monitor, il programmatore Java non deve esplicitare queste operazioni ma deve semplicemente usare metodi (o comandi) `synchronized`. Tali metodi sono dichiarati nella classe che contiene gli oggetti per i quali vogliamo garantire la mutua esclusione. Quando si invoca un metodo `synchronized` su di un oggetto in uno specifico thread, prima dell'accesso all'oggetto il metodo esegue un'operazione (implicita) di acquisizione del lock sull'oggetto stesso. Quando il metodo termina (o normalmente, oppure lanciando un'eccezione) il lock viene rilasciato. Se il lock su un oggetto è già stato acquisito, altri metodi, di thread diversi e che siano stati anch'essi dichiarati `synchronized`, se vengono invocati sullo stesso oggetto vengono bloccati, in attesa che il lock sia rilasciato. Metodi che invece non siano stati dichiarati `synchronized` possono procedere normalmente. Si noti inoltre che il lock su di un oggetto è posseduto da un thread e non da un singolo metodo: quindi se un metodo `synchronized` che possiede un lock su di un oggetto invoca un altro metodo `synchronized` sullo stesso oggetto (nello stesso thread) la computazione procede senza alcun blocco.

Come esempio di uso dei metodi `synchronized` vediamo un semplice esempio nel quale vogliamo realizzare un contatore al quale più thread possano accedere, in modo concorrente, per leggere il valore, incrementarlo o decrementarlo.

```
class Contatore {  
    private int c;  
    public Contatore (int valoreIniziale) {  
        c = valoreIniziale;  
    }  
    public synchronized int leggi() {  
        return c;  
    }  
    public synchronized incrementa() {  
        c++;  
    }  
    public synchronized decrementa() {  
        c--;  
    }  
}
```

Si noti, come si vede sopra, che il costruttore di una classe non deve essere dichiarato `synchronized` in quanto esso è invocato da un solo thread per ogni specifico oggetto. Gli altri metodi, dichiarati `synchronized`, garantiscono la mutua

esclusione nell'accesso al contatore. A questo punto, proseguendo nell'esempio, possiamo creare i thread che usano il contatore:

```
class Incr extends Thread {
    int valore;
    Contatore cont;
    public Incr(int val, Contatore contat) {
        valore = val;
        cont = contat;
    }
    public void run() {
        for (int i = 0; i < valore; i++) {
            cont.incrementa();
        }
    }
}
class Decr extends Thread {
    int valore;
    Contatore cont;
    public Decr(int val, Contatore contat) {
        valore = val;
        cont = contat;
    }
    public void run() {
        for (int i = 0; i < valore; i++) {
            cont.decrementa();
        }
    }
}
class main{
    static Contatore incrDecr new Contatore();
    public static void main (int numeroIncr, int numeroDecr){
        Thread incrementa new Incr(numeroIncr, incrDecr);
        Thread decrementa new Decr(numeroDecr, incrDecr);
        incrementa.start();
        decrementa.start();
    }
}
```

Come si vede dal codice sopra, i metodi `run` delle due classi che estendono `Thread` definiscono il comportamento per i due tipi di thread, ossia l'incremento ed il decremento del contatore. Il valore dell'incremento è passato come parametro al `main` e quindi al costruttore della classe `Incr` quando, nel `main`, viene creato un nuovo oggetto di tale classe che realizza un nuovo thread (il caso del decremento è analogo). Inoltre a tale oggetto, sempre tramite il costruttore, è passato anche un riferimento ad un oggetto di tipo `Contatore` che è quello che contiene il contatore sul quale i due thread dovranno effettuare le operazioni di incremento e decremento. La mutua esclusione è garantita dal fatto che i metodi dichiarati in `Contatore` sono `synchronized`. Il comando `start`, come abbiamo già visto, fa partire l'esecuzione dei thread.

Si tenga presente che, nel precedente esempio come in generale, se la mutua esclusione è garantita, non possiamo fare alcuna assunzione sull'ordine con il quale saranno eseguiti i metodi invocati, in thread diversi, allo stesso tempo. Come detto in precedenza questo dipende dallo scheduler dei thread per cui se

vogliamo che sia rispettato un ordine preciso questo deve essere esplicitamente programmato.

Si possono dichiarare come `synchronized` anche metodi statici per garantire la sincronizzazione nell'accesso a dati statici condivisi da più thread. Inoltre, come accennato poc'anzi, si possono dichiarare anche porzioni di codice, più limitate di un intero metodo, come `synchronized`. In questo caso la sintassi è la seguente

```
synchronized(espressione) {
    comandi
}
```

Il significato un tale comando `synchronized` è analogo a quello dei metodi `synchronized` con due importanti differenze. Innanzitutto qui l'oggetto sul quale si richiede l'acquisizione del lock non è quello corrente (cioè `this`) ma è l'oggetto il cui riferimento è il risultato della valutazione di `espressione` (che, ovviamente, deve essere del tipo giusto). In questo modo possiamo quindi esprimere sincronizzazione su più oggetti diversi tramite un unico metodo, permettendo così la progettazione di interazioni più sofisticate di quelle ottenibili con i soli metodi `synchronized`. La seconda differenza è nel fatto che qui la porzione di codice che è eseguita nella sezione critica, e cioè tra l'acquisizione ed il rilascio del lock, è più limitata rispetto a quella relativa ad un intero metodo. Questo ha conseguenze positive sulla performance, in quanto, ovviamente, avere regioni critiche più brevi permette di limitare i tempi di attesa dei thread che vogliono accedere agli stessi dati.

I metodi `synchronized` permettono la mutua esclusione ma non la sincronizzazione condizionale. Per realizzare questa e permettere quindi ai thread di comunicare direttamente fra di loro, Java mette a disposizione i metodi specifici `wait`, `notify` e `notifyAll`, tutti definiti nella classe `Object`. Il metodo `wait`, simile all'omonimo comando dei monitor, è usato per bloccare l'esecuzione di un thread fino al verificarsi di una certa condizione. I metodi `notify` e `notifyAll` invece sono simili al comando `signal` e permettono di comunicare ad uno o a tutti i thread bloccati che si è verificata una condizione che li risveglia.

Più in dettaglio, questi tre metodi sono invocati tramite un oggetto specifico e devono essere eseguiti in porzioni di codice `synchronized`, quando quindi si è già acquisito il lock su quell'oggetto.

Il metodo `wait` rilascia il lock sull'oggetto e sospende l'esecuzione del thread, che quindi è inserito nella coda d'attesa associata all'oggetto stesso. La coda di solito è gestita con politica FIFO ma vi possono essere eccezioni in quanto, come visto, possono essere usate priorità diverse. In Java non vi sono variabili condizionali esplicite come nei monitor, tuttavia possiamo pensare che vi sia una variabile condizionale implicita per ogni oggetto.

Il metodo `notify` risveglia un thread fra quelli presenti nella coda d'attesa dell'oggetto sul quale il metodo stesso è invocato. Il thread risvegliato non può essere scelto ma è il primo della coda. Data la gestione della coda, che non sempre garantisce la politica FIFO, nella pratica bisogna stare molto attenti nell'usare questo metodo per risvegliare il thread voluto. `notifyAll` invece risveglia tutti i thread in attesa sull'oggetto, quindi è di utilizzo più semplice. Sia nel caso

di `notify` che di `notifyAll` il thread che invoca il metodo mantiene il lock sull'oggetto, quindi il thread o i thread risvegliati potranno essere eseguiti sono successivamente, quando riusciranno ad acquisire il lock sull'oggetto. Usando la terminologia vista per i monitor, questi metodi hanno dunque una semantica di tipo segnala e continua.

**Nota sulla programmazione distribuita in Java** I costrutti che abbiamo sin qui visto permettono la programmazione concorrente multithreaded. Java non contiene primitive specifiche per la programmazione concorrente distribuita, tuttavia vi sono dei package che la supportano. In particolare, il package `java.net` permette la comunicazione con scambio di messaggi sia usando i datagram ed il protocollo UDP, sia usando stream e TCP. Questo package mette a disposizione le classi `Socket` e `ServerSocket` che permettono di programmare molte applicazioni di tipo client/server usando i socket (sostanzialmente dei terminali di una connessione). Con la prima delle due classi possiamo creare un socket fornendo un nome di host (o indirizzo IP), un numero di porta per tale host ed un numero di porta locale, al quale associare il socket stesso. Questo è quindi usato, tipicamente dal lato client, per comunicare dei dati. La classe `ServerSocket` invece permette di creare un particolare socket per restare in attesa su un porta, dal lato server, in attesa di richiesta di connessioni.

Java contiene anche due package, `java.rmi` e `java.rmi.server` che supportano la chiamata di procedura remota, qui detta RMI (Remote Method Invocation, cioè invocazione di metodo remoto). Il funzionamento è simile a quello che abbiamo descritto nel Paragrafo 15.5.4, con il comando `rmiic` che è l'analogo dello stub compiler lì descritto. Per ulteriori dettagli su questi package si rimanda ai manuali Java.

## 15.8 Sommario del capitolo

In questo capitolo abbiamo visto le principali problematiche della programmazione concorrente, un tema importante che per essere trattato in modo esauriente richiederebbe ben più spazio di quello che gli abbiamo potuto dedicare. Abbiamo esaminato i seguenti aspetti.

- La nozione di thread, di processo e le tre modalità principali di programmazione concorrente: parallela, multithreaded e distribuita.
- I meccanismi di sincronizzazione usati nei modelli a memoria condivisa ed in particolare l'attesa attiva, i semafori ed i monitor.
- I meccanismi di comunicazione usati nei modelli a scambio di messaggi ed in particolare la comunicazione asincrona, quella asincrona, la chiamata di procedura remota e i rendez-vous.
- I modelli di computazione non-deterministici ed i comandi con guardia.
- I principali costrutti per la programmazione multithreaded in Java e cioè le primitive di creazione, terminazione e scheduling dei thread e i metodi `synchronized`.

## 15.9 Nota bibliografica

La bibliografia sulla programmazione concorrente è molto vasta e spazia da aspetti puramente teorici a manuali pratici. Nell'ambito della teoria sono particolarmente importanti le, cosiddette, algebre di processi, formalismi che permettono di descrivere processi concorrenti e che sono dotati di leggi algebriche che permettono di verificare formalmente proprietà delle computazioni. Fra gli approcci più importanti in questo ambito ricordiamo il CSP di Hoare [44], il CCS di Milner [67] e, più recentemente, il  $\pi$ -calcolo [68]. Molto importanti sono stati anche i numerosi contributi di Edsger W. Dijkstra alla teoria della concorrenza: in [32] vennero introdotti i semafori, in [34] l'idea di monitor (con il nome di "secretary") e in [35] i comandi con guardia.

Vi sono vari testi didattici che trattano in modo completo le problematiche della programmazione concorrente. Fra questi ricordiamo [7] che offre un'ampia panoramica delle principali tecniche usate, con precisi riferimenti a vari linguaggi attuali. Un recente testo in italiano che segue un approccio orientato anche ai sistemi operativi è [6]. Per la programmazione concorrente in Java, oltre al manuale classico [10] si può consultare [54].

## 15.10 Esercizi

1. Si consideri il seguente frammento di codice:

```
int x = 4;
int y = 6;
<x = x+y> || <y = y-x>
```

dove  $\parallel$  indica la composizione parallela e si assume che tutti i comandi racchiusi nella parentesi  $< >$  siano eseguiti in modo atomico. Quali sono i risultati possibili? Se si tolgono le parentesi angolate cambiano i risultati?

2. Facendo riferimento alla notazione dell'esercizio precedente si consideri il seguente frammento di codice:

```
int x = 2;
int y = 8;
while (x != y) {x = x+1;} || while (x != y) {y = y-1;}
```

La computazione termina? Motivare la risposta.

3. Supponiamo di disporre di un'istruzione `swap(x, y)` che in modo atomico scambia il contenuto delle due variabili `x` e `y` (usando un registro interno per effettuare lo scambio). Usando una tale istruzione si realizzino le procedure di acquisizione e rilascio del lock per l'accesso a una sezione critica (si veda il Paragrafo 15.4.1).

4. Simulare i semafori generali usando i semafori binari.

5. Si consideri il seguente frammento di codice

```
int x = 10;
sem s1 = 1;
sem s2 = 0;
```

```
{P(s1); P(s2); x = x-5; V(s2); Vs(1);} ||
{P(s1); x = x*2; V(s1); V(s2);}
```

dove `sem` indica il tipo semaforo, `||` indica la composizione parallela mentre `P` e `V` sono le operazioni sui semafori che abbiamo visto. Si dica quali sono i valori finali possibili per la variabile `x` dopo l'esecuzione del frammento.

6. Si fornisca una soluzione del problema dei filosofi a cena usando un controllore centralizzato.
7. Si estenda l'applicazione client server dell'esempio 15.1 al caso in cui il numero dei client non sia noto staticamente (si usino opportune primitive che permettano la creazione di nomi nuovi e la spedizione di nomi sui canali).
8. Usando un qualsiasi pseudo linguaggio che permetta solo la comunicazione sincrona si simuli la comunicazione asincrona programmando un opportuno processo che realizzi il canale asincrono.
9. Si scriva un programma che risolva il *problema del bagno unisex*: un bagno può essere usato sia da uomini che da donne e un numero arbitrario di persone può essere nel bagno contemporaneamente, tuttavia non ci possono essere uomini e donne allo stesso tempo.
10. Si fornisca una soluzione del problema dei filosofi a cena che usi un monitor. Il monitor dovrebbe avere due operazioni `acquisisciForchetta(id)` e `rilasciaForchetta(id)`, di significato ovvio, dove `id` è l'identificatore del processo filosofo che chiama l'operazione.
11. Si definisca un monitor che gestisca un conto comune. Le operazioni del monitor devono permettere il deposito ed il ritiro di denaro da parte di più persone (processi) che condividono l'uso del conto. Il saldo del conto non può diventare negativo, e se viene richiesto il ritiro di una somma superiore al saldo questa operazione deve essere bloccata fino a quando il saldo non diventa maggiore o eguale alla somma richiesta.
12. Usando la comunicazione asincrona si sviluppino dei processi filtro che permettono di ordinare un array di `n` elementi come segue. I vari processi filtro sono in cascata uno dopo l'altro. Il primo riceve i dati, uno alla volta, trattiene il valore minimo e passa i rimanenti al secondo processo. Gli altri processi fanno lo stesso: ricevono uno stream di valori dal precedente, trattengono il minimo e passano i rimanenti valori ai successivi. Si assuma che ogni processo può memorizzare due soli valori (il minimo e il prossimo dato di input).
13. Si produca un codice che, usando la comunicazione asincrona, risolva il seguente problema del *matrimonio stabile*. Si hanno due vettori di `n` elementi, uomini e donne, rispettivamente. Ogni uomo ordina le donne da 1 a `n` (in base alle proprie preferenze) e ogni donna fa lo stesso con gli uomini. Un accoppiamento è una corrispondenza biunivoca fra uomini e donne. Un accoppiamento è stabile se non esistono due coppie  $(m_1, d_1)$  e  $(m_2, d_2)$  in esso tali che  $m_1$  preferisce  $d_2$  a  $d_1$  e  $d_2$  preferisce  $m_1$  a  $m_2$  (ossia non esiste una coppia uomo donna nella quale entrambe le persone preferiscono l'altra al proprio attuale partner).

## Una breve panoramica storica

Anche se i primi calcolatori in senso moderno, e quindi i primi linguaggi di programmazione, compaiono solo alla fine degli anni '40 del secolo scorso, sino ad oggi sono già state definite varie centinaia (se non migliaia) di linguaggi. Nei precedenti capitoli di questo libro abbiamo cercato di identificare le caratteristiche più importanti, sia progettuali che implementative, comuni ad ampie classi di linguaggi contemporanei.

In quest'ultimo capitolo cercheremo di capire quali sono stati i motivi che hanno portato, nell'arco di circa 60 anni, all'affermazione di tali caratteristiche e, quindi, al successo di alcuni linguaggi e alla scomparsa di molti altri.

In questa panoramica vedremo brevemente alcuni dei linguaggi più importanti, senza per questo avere alcuna pretesa di esaustività.

### 16.1 Gli inizi

I primi calcolatori elettronici (computer) comparvero intorno alla seconda metà degli anni '40, in epoca quindi recente, se considerata nella prospettiva della storia della scienza, ma in tempi remoti se consideriamo la velocità di sviluppo della tecnologia dell'informazione. Per rendersi conto di quale distanza ci separa dai primordi dell'informatica si pensi che i primi calcolatori erano macchine di dimensioni mastodontiche (lunghezze maggiori di 10 metri, pesi superiori alle 4 tonnellate), costi enormi, affrontabili solo da istituti di tipo governativo o da grosse aziende e potenza di calcolo inferiore a quella di una vecchia calcolatrice tascabile programmabile.

Vi è tuttora un certo dibattito su quale debba essere considerato il primo calcolatore, anche perché questo primato dipende da che cosa si intenda esattamente con tale termine. In ambito informatico vi è un certo consenso per considerare come calcolatore in senso moderno una macchina che abbia le seguenti caratteristiche: (i) sia elettronica e digitale; (ii) sia in grado di eseguire le quattro operazioni aritmetiche elementari; (iii) sia programmabile; (iv) permetta di memorizzare programmi e dati. Se consideriamo questa definizione, probabilmente il primo calcolatore che sia stato operativo, con una memoria adeguata, fu l'EDSAC, progettato e sviluppato all'università di Cambridge dal gruppo di M. Wilkes e

funzionante dal 1949. Va comunque ricordato che l'EDSAC fu influenzato da un noto lavoro di J. Mauchly e J.P. Eckert della Moore School dell'università della Pennsylvania. In questo lavoro, che secondo alcuni riprese direttamente le idee di J. von Neumann, si descriveva un calcolatore, chiamato EDVAC, con caratteristiche simili a quelle enunciate sopra. L'EDVAC tuttavia fu costruito solo nel 1951.

Se invece ammettiamo una definizione di calcolatore più generale, allora possiamo considerare come pretendenti al titolo di primo calcolatore anche ASCC/MARK I ed ENIAC (1946), che, comunque sia, sono precursori fondamentali. Queste macchine erano in grado di eseguire sequenze di operazioni aritmetiche in modo controllato da un vero e proprio programma, anche se questo non poteva essere memorizzato ed era espresso usando formalismi molto rozzi, addirittura di tipo fisico in alcuni casi.

ASCC/MARK I (IBM Automatic Sequence Controlled Calculator, o Harvard Mark I) fu costruito nel 1944 da IBM con la cooperazione dell'università di Harvard e la guida principale di H. Aiken. Questa macchina, usata dalla U.S. Navy per scopi militari, era molto rudimentale ai nostri occhi moderni: infatti riusciva solo a svolgere il lavoro di poche decine di esseri umani e richiedeva l'uso di nastri perforati e di altri dispositivi fisici esterni per impartire le istruzioni e trasmettere i dati ai dispositivi di calcolo. Si pensi, ad esempio, che una costante (decimale) di 23 cifre era specificata manualmente usando 23 interruttori che avevano ognuno 10 posizioni corrispondenti alle 10 cifre decimali !

ENIAC invece fu costruito nel 1946 da J. Mauchly e J.P. Eckert alla Moore School dell'università della Pennsylvania e inizialmente al progetto lavorò anche J. von Neumann. Anche ENIAC non permetteva la memorizzazione dei programmi ed era programmato usando dispositivi fisici, ad esempio usando internamente alla macchina cavi elettrici per collegare in vario modo, a seconda dei parametri di ingresso, le parti fisiche del calcolatore. Tuttavia molti considerano questa macchina il primo vero calcolatore perché, a differenza di ASCC/MARK I, ENIAC era molti ordini di grandezza più veloce degli esseri umani nei calcoli e fu riconosciuto immediatamente come uno strumento di progresso fondamentale: già nel 1947, L.P. Tabor della Moore School profetizzava che la velocità di ENIAC e dei calcolatori avrebbe permesso "la soluzioni di problemi matematici sin qui mai considerati a causa dell'enorme quantità di calcoli richiesti". ENIAC fu usato effettivamente per complicati calcoli di traiettorie balistiche.

Le limitazioni di queste prime macchine, ovviamente, erano dovute alla loro novità: la tecnologia hardware era tutta da inventare, i dispositivi elettronici usati nei computer moderni anche, l'informatica stessa in senso teorico stava muovendo i primi passi. Dal punto di vista della programmazione, inoltre, ancora non si era riconosciuta appieno l'importanza di quest'attività e dello sviluppo di adeguati strumenti linguistici per essa. Nelle prime applicazioni di tipo dedicato (spesso militare) l'attività di programmazione veniva vista come una fase accessoria, per così dire, del processo di calcolo. Anche in applicazioni più generali, quali quelle di EDSAC e delle altre macchine di questa generazione, le cose non migliorarono molto: per programmare si usavano linguaggi macchina di basso livello, che sostanzialmente fornivano una qualche descrizione, in codice binario, delle ope-

razioni e dei meccanismi di calcolo della macchina stessa. Si trattava quindi di *linguaggi macchina* composti da istruzioni elementari (ad esempio istruzioni di somma, di caricamento di un valore in un registro ecc.) di immediata esecuzione da parte del processore. Il processo di codifica era del tutto manuale, non esistendo ancora neppure il concetto di notazione simbolica assembly. I linguaggi macchina sono detti anche linguaggi di prima generazione (o 1GL)

Senza arrivare ai casi limite della codifica delle costanti in MARK I, è evidente che l'uso di tali linguaggi rendeva molto difficile la stesura di un programma e virtualmente impossibile, oltre una certa dimensione, la sua correzione. Ci si rese conto ben presto, quindi, che per poter utilizzare appieno il potere dei calcolatori occorreva sviluppare formalismi adeguati, più lontani dal "linguaggio" della macchina e più vicini al linguaggio naturale dell'utente.

Un primo passo in questo senso fu l'introduzione dei *linguaggi assembly*. Questi linguaggi possono essere visti come rappresentazioni simboliche dei linguaggi macchina e, difatti, vi è una corrispondenza uno a uno fra la maggior parte delle istruzioni in un linguaggio macchina ed il linguaggio assembly. Il programma scritto in assembly è tradotto in un programma scritto in linguaggio macchina da un opportuno programma detto assemblatore (assembler). Ogni modello di calcolatore ha il proprio linguaggio assembly quindi la portabilità di questi programmi è molto scarsa. I linguaggi assembly sono detti anche linguaggi di seconda generazione (o 2GL).

Il vero salto di qualità si ebbe con l'introduzione, negli anni '50, dei *linguaggi di alto livello*, detti anche linguaggi di terza generazione (o 3GL). Questi furono concepiti come linguaggi astratti, che prescindessero dalle caratteristiche fisiche del calcolatore e fossero invece adatti ad esprimere algoritmi in modo relativamente facile per l'utente umano. Fra i primi tentativi in questo senso vanno ricordati alcuni formalismi che permettevano l'uso di notazione simbolica per indicare espressioni aritmetiche. Le espressioni così codificate potevano essere poi tradotte automaticamente in istruzioni eseguibili dalla macchina. Da questi tentativi, come il nome stesso indica, nacque nel 1957 il linguaggio FORTRAN (FORmuLA TRANslator) che, come vedremo più avanti, può essere considerato a tutti gli effetti come il primo vero linguaggio di alto livello.

Dal 1957 ad oggi sono stati realizzati varie centinaia di linguaggi di programmazione e si stima che quelli di ampia diffusione siano stati più di cento. Oltre che alle mode, a motivi commerciali (certamente molto importanti), a circostanze occasionali, nello sviluppo dei linguaggi di alto livello si possono riconoscere alcune linee guida e alcuni principi evolutivi. Nella parte rimanente di questo capitolo cercheremo dunque di delineare questi principi in modo tale da fornire uno strumento di orientamento per addentrarsi nella moderna "babele" dei linguaggi di programmazione. Non abbiamo comunque alcuna pretesa di esaustività, perché una guida completa richiederebbe un altro libro.

## 16.2 Fattori evolutivi dei linguaggi

I linguaggi di alto livello sono sempre stati progettati con lo scopo di facilitare l'attività di programmazione del calcolatore. Tuttavia dagli anni '50 ad oggi è sostanzialmente cambiata l'importanza ed il costo dei vari componenti coinvolti nella realizzazione di un programma, e dunque sono completamente cambiate le priorità nella progettazione di un linguaggio.

Negli anni '50 l'hardware era sicuramente la risorsa più costosa e importante (si pensi che Thomas Watson, presidente di IBM, nel 1943 sosteneva che nel mondo non ci sarebbe stato mercato per più di cinque computer!). I primi linguaggi di alto livello furono dunque progettati con lo scopo di ottenere dei programmi efficienti, che potessero sfruttare al massimo le potenzialità dell'hardware. Questa attitudine dei primi linguaggi si riflette nella presenza di molti costrutti ispirati direttamente dalla struttura della macchina fisica. Ad esempio l'istruzione di "salto a tre vie" presente in FORTRAN derivava direttamente dalla corrispondente istruzione dell'IBM 704. Il fatto poi che la programmazione risultasse molto difficoltosa e richiedesse tempi molto lunghi era considerato un problema di secondaria importanza, risolvibile con l'impiego di maggiori risorse umane, sicuramente meno costose di quelle hardware.

Oggi la situazione è diametralmente opposta: l'hardware è relativamente economico ed efficiente e i costi preponderanti nella realizzazione di un sistema informatico sono legati all'impiego di specialisti informatici. Inoltre, date le applicazioni sempre più critiche dei sistemi di calcolo (si pensi alle applicazioni avioniche o ai sistemi di controllo per una centrale nucleare), sono presenti preoccupazioni di correttezza e di sicurezza che cinquant'anni fa erano minimi, se non assenti. I linguaggi odierni sono dunque concepiti tenendo presente innanzitutto il miglioramento delle varie attività di progettazione del software, incluse la verifica della correttezza dei programmi e la loro manutenzione, mentre le preoccupazioni relative all'uso efficiente della macchina fisica sono passate in secondo piano, salvo pochi casi particolari.

Ovviamente, non si è passati dalla visione degli anni '50 a quella attuale con una soluzione di continuità: lo sviluppo dei linguaggi di programmazione ha seguito un lungo, continuo, processo governato da vari fattori. Qui ne vediamo alcuni dei più importanti:

**Hardware** Il tipo e le prestazioni dei dispositivi hardware disponibili influenzano, evidentemente, i linguaggi che utilizzano tale hardware. Vedremo meglio questo punto nelle sezioni successive, dove identifieremo, in varie epoche storiche, le influenze delle macchine fisiche sui linguaggi del tempo.

**Applicazioni** Le applicazioni dei calcolatori, inizialmente di tipo solo numerico, si sono rapidamente estese a molti campi diversi, inclusi alcuni nei quali è necessaria l'elaborazione di informazione non numerica. Nuovi campi di applicazione possono richiedere linguaggi con specifiche caratteristiche. Ad esempio, nell'ambito dell'intelligenza artificiale e della gestione della conoscenza sono necessari linguaggi che permettano la manipolazione di formalismi simbolici

più che la soluzione di problemi matematici. Oppure, i giochi per calcolatore di solito sono realizzati usando particolari linguaggi di programmazione.

**Nuove metodologie** Lo sviluppo di nuove metodologie di programmazione, soprattutto per la programmazione in grande, ha influenzato lo sviluppo di nuovi linguaggi. Un esempio significativo in questo senso è costituito dalla programmazione orientata agli oggetti.

**Implementazione** L'implementazione dei costrutti di un linguaggio è significativa per lo sviluppo dei linguaggi successivi in quanto permette di rendersi conto della validità di un costrutto e della sua realizzabilità pratica.

**Teoria** Infine non è da dimenticare il ruolo svolto dagli studi teorici che hanno una parte importante nel selezionare alcune tipologie di costrutti e, soprattutto, nell'identificare nuovi strumenti tecnici per migliorare l'attività di programmazione: si pensi, ad esempio, a quanto visto riguardo all'eliminazione del goto e all'introduzione di raffinati sistemi di tipi.

Questi fattori, ed altri ancora, sono determinanti nel ciclo di vita di un linguaggio di programmazione che, nella maggior parte dei casi, è abbastanza breve. Fanno eccezione alcuni linguaggi che per la bontà del progetto iniziale oppure, più spesso, per motivi commerciali, sopravvivono per più di un decennio. Ne vedremo alcuni nel seguito.

## 16.3 Anni '50 e '60

Come abbiamo visto all'inizio di questo capitolo, i primi calcolatori, dalla fine degli anni '40 agli inizi degli anni '50, possono essere considerati dei precursori: interessanti da un punto di vista storico, ma molto lontani dai calcolatori moderni, sia in termini di linguaggi usati che di applicazioni realizzate.

Verso la fine degli anni '50 e negli anni '60 si affermarono i mainframe, primi veri calcolatori di uso generale, utilizzabili per molte applicazioni diverse. Si trattava, comunque, sempre di macchine disponibili solo per pochi importanti centri d'elaborazione, in quanto avevano dimensioni e costi enormi (riempivano una stanza e costavano milioni di euro, in termini di valori odierni) ed erano usabili solo da personale specializzato. L'IBM 360 è un famoso esempio di mainframe, che ha resistito per molti anni nei centri di calcolo più importanti.

I sistemi di elaborazione utilizzati dai mainframe erano detti a lotti (o batch). In questo tipo di sistema, i vari programmi erano eseguiti in modo rigidamente sequenziale: l'intera risorsa di calcolo era assegnata ad un programma che prendeva in input un "lotto" di dati e produceva in output un altro lotto di dati. Quando il programma terminava la propria esecuzione, si poteva passare all'elaborazione del programma successivo. I dati e i programmi erano inizialmente contenuti in schede perforate che venivano lette da opportuni dispositivi. Le strutture dati principali usate per l'ingresso e l'uscita dei dati erano costituite da file e, complessivamente, un tale sistema prevedeva una scarsa interazione della macchina con l'esterno. Ad esempio, in caso di errori durante l'esecuzione il programma

doveva essere in grado di poter ripristinare da solo lo stato corretto, dato che non era possibile alcun intervento interattivo esterno.

I primi linguaggi di alto livello furono sviluppati per questo tipo di sistemi di elaborazione. Si tratta di linguaggi che offrono poche possibilità di interazione con la macchina e che sono adatti a scrivere programmi monolitici, da eseguirsi dall'inizio alla fine, senza interazione con l'esterno.

**FORTRAN** Come abbiamo già detto, il primo vero linguaggio di alto livello imperativo può essere considerato FORTRAN, sviluppato dal gruppo di J. Backus nel 1957 e concepito per applicazioni di tipo numerico-scientifico. In un'epoca in cui si programmava unicamente in assembly e in cui la maggiore preoccupazione era relativa all'efficienza dei programmi, la progettazione di linguaggio di alto livello non poteva prescindere dalla prestazioni del codice compilato. Anche la progettazione di FORTRAN fu dunque fatta tenendo presente le prestazioni e, quindi, le caratteristiche di una specifica macchina fisica di riferimento (l'IBM 704). Tuttavia, a differenza dei linguaggi precedenti, già nella sua prima versione FORTRAN fu un vero e proprio linguaggio di programmazione di alto livello in senso moderno, con molti costrutti sostanzialmente indipendenti da una macchina specifica. In particolare, va sottolineato che FORTRAN fu il primo linguaggio di programmazione a permettere l'uso diretto di un'espressione aritmetica simbolica: una semplice espressione quale  $a * 2 + b$ , che oggi possiamo usare in (quasi) tutti i linguaggi di alto livello, non poteva essere usata in un programma fino all'avvento di FORTRAN. Dopo varie modifiche e nuove versioni (in particolare quelle del 1966, del 1977 e del 1990) FORTRAN è sopravvissuto fino ai giorni nostri ed è ancora usato per alcune applicazioni numeriche. Si tratta comunque di un linguaggio datato, la cui sopravvivenza è legata soprattutto a questioni pratiche, quali l'esistenza di una vasta libreria di funzioni per calcoli scientifici ed ingegneristici. Un programma FORTRAN consiste di una routine principale (main) e di una serie di sottoprogrammi, compilabili separatamente. Non vi è la possibilità di definire ambienti annidati (solo in FORTRAN 90 è possibile definire dei sottoprogrammi annidati) e si hanno solo due tipi di ambiente: quello locale e quello globale. Questo, come sappiamo, semplifica di molto la gestione dei nomi e l'implementazione dell'ambiente. La memoria, sia per sottoprogrammi che per il main, è allocata staticamente e non vi è alcuna gestione dinamica (in questo fa eccezione FORTRAN 90). I comandi per il controllo sequenza nelle prime versioni si rifanno direttamente ai linguaggi assembly e quindi si usa molto il goto. Nelle versioni successive sono stati introdotti comandi più strutturati (ad esempio if then else). Il passaggio dei parametri è per riferimento oppure per valore-risultato. I tipi, infine, sono presenti in modo molto limitato, includendo solo tipi numerici (interi, reali in singola e doppia precisione, complessi), booleani, array, stringhe e file.

**ALGOL** ALGOL indica, più che un linguaggio, una famiglia di linguaggi imperativi introdotti a partire della fine degli anni '50. Questi linguaggi, anche se non hanno mai raggiunto un vero successo commerciale, sono stati dominanti nel

mondo accademico dagli anni '60 fino agli anni '70 ed hanno avuto un impatto formidabile nel progetto di tutti i linguaggi di programmazione successivi. Molti dei concetti e costrutti che troviamo nei linguaggi moderni sono stati introdotti, o sperimentati, per la prima volta nei linguaggi della famiglia ALGOL.

Il capostipite fu ALGOL 58, progettato nel 1958 da un comitato, guidato da P. Naur, che risultò dalla fusione di due precedenti gruppi, uno europeo ed uno americano, impegnati nella definizione di un nuovo linguaggio. Il nome ALGOL è un acronimo di ALGOrithmic Language e indica chiaramente la finalità del linguaggio: a differenza di FORTRAN, infatti, ALGOL fu progettato come linguaggio universale, adatto ad esprimere algoritmi in generale invece che a realizzare specifici tipi di applicazioni. Vi furono varie revisioni del linguaggio, tutte originate mediante lavori collegiali di un comitato internazionale (dalla vita talvolta turbolenta, soprattutto nel suo ultimo periodo di vita): nel 1960 venne definito ALGOL 60, la versione forse più importante, con una revisione minore nel 1962. A partire dal 1966, in dissenso con la maggioranza del comitato, T. Hoare e N. Wirth iniziarono a gettare le basi di ALGOL W, realizzato poi nel 1968 da Wirth e che costituisce il progenitore di Pascal. La maggioranza del comitato, invece, proseguì i propri lavori che avrebbero condotto alla definizione di ALGOL 68.

ALGOL 58 e, soprattutto, ALGOL 60 migliorarono notevolmente l'indipendenza del linguaggio dalla macchina, aumentando la vicinanza della notazione usata nei programmi alla notazione matematica ed evitando qualsiasi riferimento ad architetture specifiche, anche a costo di qualche complicazione progettuale in più. Ad esempio, le operazioni di ingresso e uscita, invece che essere codificate da opportune istruzioni del linguaggio, dovevano essere realizzate da opportune procedure, da definirsi a seconda dei dispositivi usati.

ALGOL 60 ha dato almeno tre contributi fondamentali ai linguaggi moderni. Il primo consiste nell'introduzione del passaggio dei parametri per nome, un meccanismo, come abbiamo visto, complicato da implementare (e per questo successivamente abbandonato), ma che è stato di grande importanza per definire i meccanismi che si usano nei linguaggi attuali per il passaggio di funzioni.

Altra importante innovazione di ALGOL 60 fu l'introduzione dei blocchi, e quindi la possibilità di strutturare l'ambiente in modo gerarchico.

A livello sintattico invece, grazie al contributo di J. Backus, ALGOL 60 fu il primo linguaggio ad usare le grammatiche generative di Chomsky per esprimere la sintassi del linguaggio (in particolare furono usate le grammatiche libere nella forma che oggi conosciamo come Backus Naur Form o BNF). Questa novità aprì la strada ad un'intera nuova area di ricerca che ha avuto estrema importanza nella teoria (e pratica) dei compilatori.

Fra gli altri contributi dei linguaggi della famiglia ALGOL ai linguaggi moderni ricordiamo inoltre: ricorsione e gestione dinamica della memoria (ma per queste caratteristiche si veda anche quanto detto più avanti a proposito del LISP); sistemi di tipi di dato con la possibilità di definire nuovi tipi da parte dell'utente; infine molti dei comandi strutturati per il controllo di sequenza, nella forma che usiamo oggi, derivano da ALGOL 60 (if then else, for e while) o ALGOL 68 (case).

**LISP** LISP (LISt Processor) fu progettato nel 1960 da un gruppo guidato da J. McCarthy al M.I.T. (Massachusetts Institute of Technology) e fu uno dei primi linguaggi progettati espressamente per applicazioni di tipo non numerico. Come abbiamo già visto nel riquadro di pag. 199, si tratta di un linguaggio concepito per manipolare una certa classe di espressioni simboliche ( dette s-espressioni ) che sostanzialmente sono delle liste e che tipicamente si utilizzano nell'ambito dell'intelligenza artificiale. Fra le applicazioni di LISP ricordiamo i primi tentativi di realizzare programmi per la traduzione automatica di testi.

Anche se si tratta di un linguaggio non standardizzato, sviluppato e implementato nel corso di 30 anni in molte versioni diverse e non molto diffuso a livello commerciale, si tratta di un linguaggio molto importante, nel cui ambito sono state sviluppate varie tecniche di interesse generale per i linguaggi e che in ambito accademico gode ancora di un certo seguito ( il linguaggio Scheme, attualmente usato in molti corsi, è nato da una variante di LISP ). Le prime implementazioni di LISP risultarono molto inefficienti, tanto che si arrivò a costruire architetture dedicate per il LISP ( le cosiddette LISP machines ). Successivamente l'uso di vari accorgimenti, e in particolare i miglioramenti tecnologici nel garbage collector, permisero implementazioni efficaci anche su architetture tradizionali.

LISP è un linguaggio funzionale che, come detto, permette di manipolare particolari strutture dati. Ogni programma consiste di una sequenza di espressioni che devono essere valutate; alcune di tali espressioni possono essere definizioni di funzioni, da usarsi poi in altre espressioni. Tipicamente il linguaggio è implementato in modo interpretativo ed i programmi vengono valutati in un ambiente interattivo. Fra i contributi importanti di LISP ricordiamo l'introduzione dell'ordine superiore, ossia la possibilità di costruire funzioni che accettano come parametri e/o restituiscono come risultato della valutazione altre funzioni, e la gestione dinamica della memoria con heap e garbage collector. Altre caratteristiche specifiche di LISP sono invece rimaste confinate quasi esclusivamente a questo linguaggio: è questo, ad esempio, il caso dell'adozione della regola di scope dinamico, gestita mediante lista delle associazioni ( A-list ).

**COBOL** Anche questo, come LISP, è un linguaggio che risale al 1960 e anche in questo caso il nome è un acronimo: COBOL sta per COnmon Business Oriented Language. Le somiglianze fra i due linguaggi però finiscono qui. COBOL infatti fu progettato con lo scopo di ottenere un linguaggio specifico per applicazioni commerciali la cui sintassi fosse il più possibile vicina a quella della lingua inglese. Dopo varie revisioni del linguaggio iniziale, progettato da un gruppo guidato da G. Hopper al Dipartimento della Difesa statunitense, fu definito il linguaggio COBOL standard nel 1968 che, anche se in versioni riviste e modificate, è un linguaggio ancor oggi utilizzato.

I programmi COBOL sono composti da 4 "divisions" ( sezioni ): nella "procedure division" viene inserito il codice relativo agli aspetti algoritmici del programma; la "data division" contiene la descrizione dei dati; la "environment division" contiene delle specifiche relative all'ambiente esterno al programma, dipendenti dalla macchina fisica sulla quale il linguaggio è implementato; infine la "identifi-

cation division" contiene informazioni utile per identificare il programma ( nome, autore ecc.). Questa divisione ha sia lo scopo di separare, anche se in modo molto grossolano, i dati dai programmi che li utilizzano, sia di separare gli aspetti dipendenti dalla macchina fisica da quelli indipendenti. Si tratta comunque di meccanismi molto rudimentali, superati dagli strumenti linguistici che abbiamo visto esistere nei linguaggi moderni ( tipi, tipi di dato astratti, oggetti, moduli, codice intermedio ecc.). Inoltre, questa struttura del programma, unitamente alla sintassi vicina al linguaggio naturale, rende anche i programmi più semplici abbastanza lunghi. La gestione della memoria è interamente statica.

**Simula** Simula, altro discendente di ALGOL 60, è un esempio da manuale di una tecnologia troppo avanzata per la propria epoca. Sviluppato a partire dal 1962 presso il Norwegian Computing Center da K. Nygaard e O.-J. Dahl, Simula è un'estensione di ALGOL progettata per applicazioni di simulazione discreta, cioè per la realizzazione di programmi che simulano situazioni di carico e code al fine di misurarne i parametri fondamentali ( tempi medi di attesa, lunghezza delle code ecc.). Nella sua versione più importante ( Simula 67 ), il linguaggio introduce per la prima volta le nozioni di classe, oggetto, sottotipo, selezione dinamica dei metodi. Si tratta senza dubbio del primo linguaggio orientato agli oggetti, che ebbe un'influenza notevole sui successori ( Smalltalk e C++ ) anche se la nozione di "oggetto" come metafora biologica non sarebbe arrivata che con Smalltalk ed il suo vulcanico ideatore, A. Kay.

Da un punto di vista linguistico, Simula 67 apporta piccole modifiche ai costrutti esistenti in ALGOL 60 ( la più importante delle quali è la modalità default di passaggio dei parametri che passa da nome a valore-risultato ), ma aggiunge diversi meccanismi, tra i quali il passaggio per riferimento, puntatori, coroutine ( un meccanismo per definire procedure concorrenti in qualche modo vicino ai moderni thread ), classi e oggetti. Una classe è in Simula una procedura che, alla sua terminazione, lascia il proprio Rda sulla pila, restituendo un puntatore a tale Rda. Un Rda di questo tipo, che contiene le variabili locali dichiarate della procedura ( diremmo oggi: le variabili di istanza ) e ( puntatori alle ) funzioni locali ( i metodi ), è un oggetto. Sottotipi, selezione dinamica e ereditarietà sono già presenti in Simula 67 , mentre si dovranno attendere le versioni successive per l'introduzione di meccanismi di astrazione.

Simula ha sempre avuto un impatto significativo anche al di fuori del mondo accademico; ancora nel 2003 venivano citate applicazioni reali di simulazione discreta che usano questo linguaggio.

**Nascita della concorrenza** Come abbiamo detto nel Paragrafo 15.2, gli anni '60 hanno visto anche la nascita dei primi costrutti per la programmazione concorrente. Si trattava di costrutti di basso livello, impiegati nei sistemi operativi, necessari per gestire la concorrenza introdotta nella computazione dai controllori dei dispositivi e dai relativi meccanismi di comunicazione basati sulle interruzioni. Dal punto di vista teorico in questi anni si iniziarono a studiare i primi problemi di sincronizzazione, cercando di individuare condizioni e costrutti che garantisse-

ro proprietà di indipendenza dell'esecuzione di processi paralleli (Bernstein) e di mutua esclusione (Dijkstra). Come già ricordato, i semafori vennero introdotti in questi anni (erano presenti già in ALGOL 68). Anche il classico problema di sincronizzazione dei filosofi a cena risale a questo periodo.

## 16.4 Anni '70

Grazie all'avvento del microprocessore, gli anni '70 videro l'affermazione dei minicomputer, calcolatori di dimensioni ridotte e prestazioni paragonabili a quelle dei mainframe più vecchi.

Dal punto di vista del software si passò dalle elaborazioni batch ad elaborazioni di tipo interattivo nelle quali l'utente, tramite un terminale, poteva interagire direttamente con l'esecuzione del programma. Le caratteristiche dei linguaggi sviluppati nell'epoca dei mainframe non erano più adatte a sistemi interattivi. Ad esempio, divenne necessario poter esprimere tramite un programma (e dunque tramite i costrutti di un linguaggio) operazioni di ingresso/uscita più sofisticate della semplice lettura e scrittura da un file. Nei sistemi interattivi sono presenti vincoli sui tempi di risposta e quindi i nuovi linguaggi inclusero opportuni costrutti linguistici di natura temporale (ad esempio, meccanismi di time-out). In generale, i linguaggi di alto livello "tradizionali" degli anni '70 permisero un'interazione più diretta con la macchina di quanto non fosse possibile con i linguaggi della generazione precedente.

Sopra abbiamo detto "tradizionali", sottintendendo con questo aggettivo il riferimento ai linguaggi imperativi, ispirati al modello di calcolo classico basato sulla modifica di valori memorizzati in locazioni di memoria. Gli anni '70 hanno però visto anche la nascita di due nuovi paradigmi di programmazione: quello orientato agli oggetti e quello dichiarativo, dove il secondo può essere più propriamente separato nelle due componenti funzionale e logica. Vedremo sotto i linguaggi più importanti di questi anni nei diversi paradigmi.

**C** Fra i nuovi linguaggi imperativi degli anni '70 il più importante è probabilmente C, un linguaggio progettato da D. Ritchie e K. Thompson nei laboratori della AT&T Bell Telephone Corp. Inizialmente concepito come linguaggio per la programmazione di sistema nell'ambito dello sviluppo del sistema operativo UNIX, C divenne ben presto un linguaggio di uso generale, anche se la programmazione di sistema rimane un suo importante campo d'applicazione. A titolo aneddotico ricordiamo che il nome deriva dal fatto che il linguaggio del 1972 fu la versione successiva di un precedente linguaggio chiamato B, che a sua volta era la versione ridotta del linguaggio di sistema BCPL.

Rispetto ai linguaggi della famiglia ALGOL, dai quali comunque ha ereditato molte caratteristiche, C offre molte più possibilità per quanto riguarda l'accesso a funzionalità di basso livello della macchina e per programmare sistemi interattivi. Ad esempio, in C è possibile accedere in modo diretto ai caratteri immessi da un terminale oppure è possibile usare comandi specifici per l'elaborazione di dati in

tempo reale. C si è rapidamente affermato, sia per la programmazione di sistema sia come linguaggio di uso generale, grazie a queste caratteristiche, alla sua sintassi compatta ed alla possibilità di tradurre i suoi programmi in codice macchina efficiente.

Abbiamo visto, nei capitoli precedenti, numerosi riferimenti a C per specifici costrutti o specifiche caratteristiche implementative e sarebbe inutile ripeterli tutti qui. Ricordiamo solo alcune caratteristiche salienti: abbiamo visto come C includa molti operatori aritmetici e di assegnamento e come la struttura a blocchi di C, molto semplificata rispetto ai linguaggi della famiglia ALGOL (in sostanza C non permette funzioni annidate), permetta una gestione molto più semplice dell'ambiente e, quindi, del passaggio di funzioni come parametro. La presenza esplicita dei puntatori, gestibili direttamente da programma e l'equivalenza fra questi e gli array, se da un lato permettono operazioni molto potenti, dall'altro possono essere causa di errori anche molto insidiosi. Questo fatto, unitamente alla mancanza di un sistema di tipi forte, costituisce uno dei punti critici del linguaggio che, dove si cerchi più l'affidabilità che l'efficienza, può trovare validi antagonisti in altri linguaggi moderni.

**Pascal** Pascal fu sviluppato intorno al 1970 da N. Wirth come evoluzione e semplificazione di ALGOL W ed ebbe un'ottima diffusione come linguaggio didattico fino alla fine degli anni '80. Il nome è in onore del matematico (nonché fisico, filosofo e scrittore) B. Pascal il quale, pare per aiutare il padre incaricato di un difficile compito presso l'amministrazione fiscale della Normandia, progettò nel 1642 una "macchina aritmetica", capostipite delle macchine calcolatrici meccaniche.

Uno dei motivi principali del successo e della fama di Pascal è nell'essere stato il primo linguaggio che, precorrendo di circa vent'anni Java ed il suo bytecode, ha introdotto la nozione di codice intermedio quale strumento per favorire la portabilità dei programmi. Un programma Pascal veniva tradotto dal compilatore Pascal (scritto in Pascal) in P-code: questo era il linguaggio di una macchina intermedia, con architettura a pila che veniva poi implementata in modo interpretativo sulla macchina ospite. In questo modo, per portare Pascal su una diversa macchina piattaforma bastava riscrivere l'interprete di P-code. Pascal è stato implementato anche in modo compilativo, senza la presenza di una macchina intermedia e quindi con maggiore efficienza.

Anche per Pascal abbiamo inserito molti riferimenti nel corso di questo testo e non intendiamo qui farne un riassunto completo. Ricordiamo solo alcune caratteristiche importanti, cercando di confrontarle con quelle di C.

Pascal è un linguaggio strutturato a blocchi nel quale è possibile definire funzioni e blocchi annidati di arbitraria complessità. Questo fatto, se da un lato aumenta le possibilità di strutturazione del codice, complica la gestione dell'ambiente e del meccanismo di passaggio dei parametri funzionali, come già detto. Pascal (come C), usa la regola di scope statico e richiede la gestione dinamica della memoria, sia con pila (per i record di attivazione) che con heap (per la memoria allocata esplicitamente). Il sistema di tipi di Pascal è abbastanza esteso e supporta

meccanismi di astrazione permettendo all'utente la definizione di nuovi tipi con la primitiva `type`. La maggior parte delle informazioni di tipo sono controllate staticamente dal compilatore, anche se alcuni controlli sono eseguiti a tempo di esecuzione. Anche in Pascal è possibile gestire esplicitamente i puntatori, anche se non sono presenti tutte le possibilità (incluse quelle negative) di C: ad esempio, in Pascal array e puntatori sono due tipi diversi. Forse, nella ricerca di un sistema di tipi affidabile, riguardo agli array in Pascal si è ecceduto nel verso opposto: infatti, in Pascal, due dichiarazioni di array che abbiano dimensioni diverse e contengano oggetti dello stesso tipo definiscono due tipi diversi, cosa che rende difficile la progettazione di procedure generali per la manipolazione di array. Un altro limite del linguaggio, almeno nella sua versione originale, è la mancanza di moduli compilabili in modo separato, anche se questo è stato superato in molte implementazioni successive, permettendo la definizione di procedure esterne.

**Smalltalk** Un forte limite di Pascal, così come di tutti gli altri linguaggi di programmazione "convenzionali" degli anni '70, risiede nella mancanza di strumenti che supportino le nozioni di encapsulamento e di information-hiding in modo veramente efficace. Infatti, anche se in Pascal è possibile definire nuovi tipi di dato, non vi è alcun modo per poter limitare ad un insieme prefissato di operazioni l'accesso ai valori, in modo da poter garantire astrazione sui dati.

Smalltalk presenta un modo innovativo per integrare in un linguaggio di programmazione strumenti di encapsulamento e di information-hiding, attraverso la nozione di classe e oggetto (già presenti in Simula) e precise regole di visibilità tra classi (i metodi sono pubblici, le variabili di istanza sono private). Sviluppato nel corso degli anni anni '70 da A. Kay allo Xerox PARC (Palo Alto Research Center)<sup>1</sup>, Smalltalk è un linguaggio molto particolare, del quale abbiamo visto alcune caratteristiche nel Capitolo 12 e la cui descrizione richiederebbe molto più spazio di quello qui disponibile. Ci interessa solo ricordare che, a differenza di alcuni linguaggi orientati agli oggetti introdotti successivamente (ad esempio C++), Smalltalk fu progettato sin dall'inizio includendo come primitiva la nozione di oggetto e non aggiungendo tale nozione ad un linguaggio già esistente. Questo significa che tutti i meccanismi implementativi (chiamate di procedura, gestione della memoria ecc.) sono pensati per tale nozione. Inoltre Smalltalk fu concepito per essere non solo un linguaggio, ma anche una sorta di "sistema totale" che includesse linguaggio, ambiente di programmazione ed anche una specifica macchina dedicata che permettesse una maggiore efficienza nell'esecuzione dei programmi. Infatti, dato che il sistema di tipi del linguaggio è interamente dinamico, l'implementazione di un meccanismo efficiente di lookup dei metodi risulta abbastanza problematica. Ben presto comunque furono proposte implementazio-

<sup>1</sup>Xerox PARC fu in quegli anni un centro di ricerca mitico. Oltre a Smalltalk, vi furono sviluppati Ethernet; la tecnologia per la stampa laser (poi sviluppata da Adobe, una spin-off di PARC); Postscript; il primo personal computer per l'"automazione di ufficio" (Alto) dotato di una Graphical User Interface con desktop simile ad una scrivania, menù a tendina e uso estensivo di mouse; il concetto di remote procedure call.

ni di Smalltalk anche su macchine convenzionali, con risultati soddisfacenti. Non è mai stato definito uno standard per Smalltalk ed oggi esistono varie versioni, anche molto diverse del linguaggio.

**I linguaggi dichiarativi** Abbiamo già visto nel Capitolo 8 la differenza esistente, almeno sul piano formale, fra programmazione imperativa e programmazione dichiarativa. Dal punto di vista intuitivo, lo slogan, per così dire, della programmazione dichiarativa è che l'attività di programmazione deve concentrarsi sul "cosa" il calcolatore debba fare, lasciando che sia l'interprete del linguaggio a preoccuparsi del "come" raggiungere il risultato desiderato. Nella programmazione imperativa invece il programmatore deve specificare sia il "cosa" che il "come". Ovviamente questa è una visione ideale: demandare interamente all'interprete tutti gli aspetti relativi al "come" (e cioè, sostanzialmente, gestione della memoria e controllo di sequenza) senza che il programma fornisca alcuna indicazione in questo senso è molto penalizzante in termini di efficienza dei programmi. Nella realtà dunque, alle versioni "pure" dei linguaggi dichiarativi, che si conformano a questa visione, si aggiungono versioni "non pure" che aggiungono costrutti di natura imperativa per migliorare l'efficienza dei programmi ed anche per permettere l'uso di comandi (ad esempio l'assegnamento) ai quali molti programmati tradizionali sono abituati.

I linguaggi dichiarativi si dividono in linguaggi funzionali e linguaggi logici. Vediamo i due rappresentanti più importanti delle due classi, introdotti entrambi negli anni '70.

**ML** ML nacque come Meta Linguaggio (da cui il nome) per un sistema semiautomatico di dimostrazione di proprietà dei programmi e fu sviluppato dal gruppo di R. Milner a Edinburgo a partire dalla metà degli anni '70. Ben presto divenne un vero linguaggio di programmazione, utile soprattutto per la manipolazione di informazione simbolica.

Anche in ML, come in LISP, un programma consiste in un insieme di definizioni di funzioni. Alla parte puramente funzionale di ML vennero aggiunte varie caratteristiche imperative, in particolare il comando di assegnamento limitato alle cosiddette "reference cells" che possono essere considerate delle variabili (modificabili) che usano il reference model.

Il contributo più importante di ML riguarda sicuramente i tipi. Il linguaggio fu infatti dotato di un sistema di tipi sicuro, statico, che estendeva per vari versi il sistema di tipi di Pascal e superava le molte lacune del sistema di tipi di C. Innanzitutto, la nozione di sistema di tipi sicuro (*type safe*) che abbiamo già incontrato, ha una definizione rigorosa ben precisa che esclude (in modo dimostrabile) la possibilità di errori a run-time non segnalati che derivino da violazioni di tipo. Il type checker di ML determina staticamente il tipo per ogni espressione del linguaggio, e non vi è modo di cambiare tale tipo: se il type checker determina che un'espressione ha tipo intero allora siamo sicuri che ogni valutazione di tale espressione che abbia successo produrrà un intero.

Inoltre il sistema di tipi di ML supporta un meccanismo di inferenza di tipo: il programmatore può lasciare non specificate alcune informazioni di tipo ed il sistema, usando una forma di inferenza logica, deduce il tipo degli identificatori dal modo in cui essi sono usati. Anche se meccanismi similari furono studiati precedentemente nel contesto del lambda calcolo, ML fu il primo linguaggio ad essere stato dotato di un meccanismo di inferenza di tipo.

Infine ML permette il polimorfismo parametrico, ossia permette l'uso di variabili di tipo che poi possono essere consistentemente rimpiazzate da tipi concreti.

**PROLOG** Degli anni '70 anche la definizione di PROLOG, il primo linguaggio logico e ancora oggi presente in varie versioni e implementazioni.

Se, come abbiamo detto, alcune idee della programmazione logica possono essere ricondotte ai lavori di K. Gödel e J. Herbrand, le prime solide basi teoriche furono poste da A. Robinson che, negli anni '60, diede un contributo essenziale alla teoria della dimostrazione automatica. Robinson infatti fornì una definizione formale dell'algoritmo di unificazione e definì la *risoluzione*, un meccanismo di deduzione che usa l'unificazione e permette di dimostrare teoremi della logica del prim'ordine.

Data la sua semplicità la risoluzione si presta ad essere implementata per realizzare un dimostratore automatico di teoremi (della logica del prim'ordine), tuttavia questo non fornisce ancora un meccanismo di calcolo quale quello normalmente offerto dai linguaggi di programmazione: la dimostrazione di un teorema, di per sé, non produce un risultato "osservabile" che possa essere visto come il risultato della computazione. Per ottenere questa visione computazionale dell'attività di dimostrazione si dovettero aspettare altri 10 anni e una versione ristretta della risoluzione, detta risoluzione SLD proposta da R. Kowalski nel 1974. La risoluzione SLD, a differenza di quanto avviene con altri meccanismi di dimostrazione automatica, permette di dimostrare una formula calcolando esplicitamente i valori delle variabili che rendono vera la formula stessa e che quindi, al termine della deduzione, forniscono il risultato della computazione. Se Kowalski definì il modello teorico dobbiamo ricordare che in realtà il linguaggio PROLOG fu sviluppato da P. Rousset e A. Colmerauer che già dal 1970 lavoravano ad un formalismo per la manipolazione del linguaggio naturale basato su meccanismi di dimostrazione automatica di teoremi. Questi esperimenti portarono alla prima implementazione di PROLOG nel 1972 e quindi, dopo varie interazioni con R. Kowalski, alle versioni del 1973 che è in sostanza molto simile alle versioni attuali. Lo standard ISO del PROLOG è stato definito negli anni '90.

**I primi linguaggi concorrenti** Negli anni '70 si sono sviluppati i primi linguaggi concorrenti, sia in senso teorico che pratico. Dal punto di vista teorico particolarmente importante è stata l'introduzione dei cosiddetti calcoli di proces-

so o algebre di processo<sup>2</sup>. Questi formalismi permettono di modellare in modo matematicamente preciso i sistemi concorrenti usando pochi costrutti primitivi per esprimere l'interazione, la comunicazione (tipicamente mediante scambio di messaggi) e la sincronizzazione dei processi. Associati a questi calcoli si hanno delle leggi algebriche o comunque delle semantiche formali che permettono di verificare varie proprietà delle computazioni. I primi, fondamentali, contributi in questo settore furono quelli di Robin Milner, che nella seconda metà degli anni '70 introdusse il Calculus of Communicating Systems (CCS), e quelli di Charles Antony Richard Hoare che nel 1978 pubblicò il suo lavoro su Communicating Sequential Processes (CSP), un calcolo di processi che successivamente si sviluppò nel linguaggio *Occam*. In questi anni vi furono anche i primi importanti contributi teorici alla dimostrazione di proprietà quali la correttezza parziale, la mutua esclusione e l'assenza di deadlock per i sistemi concorrenti. Importanti in questo senso furono i lavori di Susan Owicky, David Gries e Leslie Lamport. Negli anni '70 furono sviluppati i monitor soprattutto grazie ad un famoso articolo di Hoare del 1974, anche se l'idea iniziale di incapsulare i dati per controllare l'accesso alle variabili condivise in un ambiente concorrente è attribuita a Dijkstra (1971) e a Per Brinch Hansen (1972). Quest'ultimo introdusse nel 1975 il *Concurrent Pascal*, il primo linguaggio concorrente che adottò i monitor. Brinch Hansen fu anche fra i primi, sempre in questi anni, a concepire l'idea della RPC (chiamata di procedura remota). Vari altri linguaggi di questo periodo usarono i monitor, fra questi *Modula*, sviluppato da Niklaus Wirth e *CSP/k*, sviluppato da Ric Holt. Infine va ricordato che gli anni '70 sono quelli del maggior sviluppo del sistema operativo UNIX, che conteneva chiamate di sistema per gestire la concorrenza quali *fork*, *wait* ed *exit*.

## 16.5 Anni '80

Gli anni '80 sono stati dominati dallo sviluppo del calcolatore per uso personale (Personal Computer o PC). Il primo PC commerciale si può forse considerare APPLE II, prodotto dalla Apple nel 1978. Anche se oggi, con la diffusione capillare dei dispositivi di calcolo in molti aspetti della vita quotidiana, il PC appare uno strumento indispensabile, va ricordato che all'epoca della sua introduzione i più rimasero scettici riguardo alle sue possibilità. Basti ricordare che, ancora nel 1977, K. Olson, presidente di Digital Eq. Co. (un'importante produttrice di minicomputer del tempo), affermava di non vedere alcun motivo per cui una persona dovesse voler avere un calcolatore a casa propria! Queste perplessità iniziali caddero quando, nel 1981, IBM lanciò il suo primo PC e Lotus creò il primo foglio elettronico. Quest'applicazione fece immediatamente comprendere a molti le possibilità di questa nuova tecnologia, che poi divenne di uso veramente generale a partire dal 1984. In quell'anno Apple immise sul mercato il PC Macintosh che aveva il primo sistema operativo con interfaccia grafica, basata su

<sup>2</sup>Quest'ultima terminologia tuttavia è stata introdotta successivamente, negli anni '80.

finestre, icone e mouse, del tutto analoga alle interfacce a finestre che usiamo oggi. Successivamente (negli anni '90) anche Microsoft introdusse il proprio sistema a finestre.

Il PC ha cambiato sostanzialmente il ruolo dei linguaggi di programmazione: lo sviluppo di sistemi per uso personale, che devono offrire interfacce grafiche di uso elementare, ha portato necessità di sviluppare facilmente sistemi grafici interattivi, necessari per gestire le interfacce a finestre. Tali sistemi sono ottenuti mediante programmi molto estesi e complessi e dunque per essi è essenziale la possibilità di poter riusare codice preesistente (eventualmente prodotto da altri). Questi sistemi hanno dunque fornito un campo d'applicazione ideale per i linguaggi orientati agli oggetti che, come abbiamo visto, offrono meccanismi naturali per il riuso del codice e per la strutturazione di sistemi vasti e complessi. In effetti, i protagonisti dei linguaggi degli anni '80 sono stati i linguaggi orientati agli oggetti, che hanno visto in questi anni un notevole sviluppo e le prime importanti applicazioni commerciali.

Sempre in questo decennio si sono sviluppati i primi sistemi embedded, ovvero sistemi costituiti da calcolatori cablati entro dispositivi fisici di cui devono effettuare il controllo (ad esempio motori, parti di impianti industriali, elettrodomestici ecc.). I sistemi embedded pongono molti problemi, primo fra tutti quello relativo all'affidabilità e alla correttezza dei programmi: per un programma che controlla i motori di un aereo, una situazione di errore non può, evidentemente, essere gestita in modo interattivo dal programmatore e la terminazione del programma non è un'opzione accettabile. Inoltre i sistemi embedded introducono anche problematiche relative ai tempi di risposta, problematiche affrontate nei cosiddetti linguaggi per sistemi in tempo reale (o real-time).

**C++** La prima versione di C++ fu definita da B. Stroustrup nel 1986 presso i Laboratori Bell dell'AT&T negli Stati Uniti, dopo vari anni di lavoro (e vari linguaggi definiti) per cercare di aggiungere classi ed ereditarietà al linguaggio C, senza per questo peggiorarne l'efficienza e senza compromettere la compatibilità con il linguaggio preesistente: C doveva rimanere un sottoinsieme di C++ e come tale doveva essere accettato dal compilatore di quest'ultimo. A questi obiettivi primari si aggiunse anche quello di migliorare il sistema di tipi di C.

Possiamo dire che, sostanzialmente, questi obiettivi sono stati raggiunti. Anche se esistono alcune inconsistenze fra C e C++, la maggior parte dei programmi C può essere tradotta da un compilatore C++. Riguardo all'efficienza è stato fatto uno sforzo notevole per ottenere un linguaggio in cui le caratteristiche non utilizzate in un programma non peggiorino l'efficienza del programma stesso. Questo significa, in particolare, che il sottoinsieme C di C++ non deve risentire in alcun modo della presenza degli oggetti e delle strutture necessarie per gestirli. Sempre nell'ottica dell'efficienza e della compatibilità con C che, come sappiamo, permette la gestione esplicita (da programma) della memoria, C++ non usa alcun metodo di garbage collection.

Il sistema dei tipi (statico) è stato migliorato e, in particolare, in C++ è possibile usare una forma generica di classe, detta template, che supporta una sorta di

### Ada Byron Lovelace

Il nome del linguaggio Ada è un tributo ad Ada Byron, contessa di Lovelace (1815-1852). Figlia del poeta Byron, fu una delle prime figure femminili nella storia del calcolo automatico, grande sostenitrice di C. Babbage (1792-1871), matematico all'Università di Cambridge e pionere delle macchine calcolatrici moderne. Babbage definì due macchine calcolatrici (quella "analitica" e quella per "differenze") che precorsero i tempi e risultarono di tale complessità tecnica da non poter essere realizzate. Nel 1842 il matematico italiano L. Menabrea pubblicò una memoria, in francese, sulla macchina analitica di Babbage. Ada Lovelace, nel 1843, tradusse tale memoria in inglese, aggiungendovi numerose note dalle quali risulta una lungimirante visione di una macchina calcolatrice per uso generale che permettesse "di sviluppare e tabulare una qualsiasi funzione .... la macchina [è] l'espressione di ogni funzione indefinita, di ogni grado di generalità e complessità".

polimorfismo parametrico. Una importante decisione progettuale è stata quella di trattare gli oggetti di C++ come generalizzazioni delle strutture (*struct*) di C, il che significa che gli oggetti possono anche essere allocati nei record di attivazione sulla pila e, a differenza di quello che avviene in Simula e in Java, essi possono essere manipolati direttamente invece che mediante i puntatori. Un assegnamento in C++ può quindi copiare fisicamente un oggetto nello spazio di memoria che era occupato da un altro oggetto, invece che modificare un puntatore.

Il meccanismo di lookup dei metodi in C++ risulta più semplice ed efficiente di quello di Smalltalk, dato che in C++ si possono usare informazioni fornite dal sistema di tipi statico, che invece non esiste in Smalltalk.

Infine, C++ permette anche di usare ereditarietà multipla, una caratteristica questa che pone vari problemi implementativi. La versione standard di C++ è stata approvata nel 1996.

**Ada** Un altro importante linguaggio definito negli anni '80 è Ada, realizzato grazie alla sponsorizzazione del Dipartimento della Difesa degli Stati Uniti. Il linguaggio Ada fu definito, in modo un po' inusuale, partendo da una gara fra vari gruppi di progettisti, sia accademici che industriali, per la realizzazione di un nuovo formalismo che soddisfacesse le esigenze del Dipartimento della Difesa. La gara fu vinta da J. Ichbiah nel 1979 con un linguaggio, basato su Pascal, che includeva molte nuove caratteristiche adatte per la programmazione di sistemi real-time ed embedded (ma non solo). La proposta di Ichbiah includeva i tipi di dato astratti, la nozione di task, meccanismi di controllo di natura temporale e meccanismi per l'esecuzione concorrente di task. In particolare quest'ultima caratteristica introdusse problematiche completamente nuove nell'ambito dei linguaggi di programmazione commerciali esistenti al tempo. L'idea di base dei task è semplice: se un task A chiama un task B, il task A continua ad essere eseguito durante l'esecuzione di B (nel caso dei normali sottoprogrammi o delle procedure, invece, ovviamente l'esecuzione di A è sospesa mentre si esegue B).

Questa esecuzione concorrente di due task, come abbiamo visto, pone problemi di sincronizzazione, comunicazione e gestione che richiedono opportuni costrutti linguistici ed adeguati meccanismi implementativi (ad esempio, il *rendez-vous*, introdotto ed estensivamente usato in questo linguaggio).

La versione standard di Ada fu definita nel 1983 anche se, a causa dei controlli per l'aderenza allo standard, i primi traduttori apparvero nel 1986, caso forse unico nel panorama dei linguaggi di programmazione.

**Gli sviluppi dei linguaggi concorrenti** Oltre ad Ada, gli anni '80 hanno visto l'affermarsi di altri linguaggi per la programmazione concorrente. Sulla base del modello teorico CSP, alla INMOS fu sviluppato il linguaggio *Occam*, prima per la programmazione di reti di Transputer e successivamente anche per altre piattaforme. Si tratta di un linguaggio il cui principio ispiratore, come suggerisce il nome<sup>3</sup>, è quello della semplicità. Occam fu il primo linguaggio ad usare esplicitamente i canali. In questi anni oltre al Transputer, primo microprocessore progettato specificatamente per essere usato in sistemi di calcolo parallelo, si svilupparono numerose architetture parallele quali le macchine con architettura a ipercubo della Intel, la Connection Machine e le macchine della Cray Research. Conseguentemente si svilupparono anche metodologie e linguaggi di programmazione specifici, anche se molti di essi ebbero una vita breve.

Nell'ambito delle algebre di processo Jan Bergstra e Jan Willem Klop svilupparono la Algebra of Communicating Processes (ACP) e introdussero esplicitamente questo termine nella letteratura.

Agli inizi degli anni '80 Gregory R. Andrews sviluppò Synchronizing Resources (SR), un linguaggio progettato esplicitamente per la programmazione concorrente che ha avuto un certo seguito nel mondo accademico. Di questi anni è anche Linda, un modello di computazione alternativo, definito da Davide Gelernter, nel quale la comunicazione e la sincronizzazione avvengono mediante una struttura condivisa dai processi detta blackboard.

Negli anni '80 furono definite anche varie estensioni concorrenti di linguaggi funzionali e logici, con lo scopo di ottenere un formalismo concorrente che mantenesse gli aspetti positivi della programmazione dichiarativa. Fra i linguaggi logici concorrenti definiti in questo periodo ricordiamo PARLOG, Concurrent Prolog e Guarded Horn Clauses (GHC). Quest'ultimo riveste un ruolo particolare in quanto fu uno dei protagonisti del progetto giapponese FGCS (Fifth Generation Computer Systems) che si sviluppò nel corso degli anni '80. Questo progetto ebbe finanziamenti colossali da parte del governo giapponese (complessivamente circa 400 milioni di dollari USA ai valori del 1992) e avrebbe dovuto portare alla realizzazione di una nuova generazione di computer paralleli, con prestazioni superiori dovute ad architetture assolutamente innovative che usavano il GHC come linguaggio macchina. In realtà, nonostante i numerosi importanti risultati scien-

tifici ottenuti, il progetto non fu un successo e le macchine costruite furono ben presto superate nelle prestazioni da altre con architetture più convenzionali.

**CLP** Negli anni '80 furono introdotti i linguaggi logici con vincoli, linguaggi che permettono di manipolare relazioni su opportuni domini e ai quali abbiamo accennato nel Capitolo 14. L'idea di aggiungere alla programmazione logica classica meccanismi di soluzione di vincoli fu sviluppata indipendentemente da tre gruppi di ricerca diversi. Colmerauer ed il suo gruppo a Marsiglia fu il primo a definire un linguaggio con vincoli nel 1982: si tratta di PROLOG II, un'estensione di PROLOG che permetteva di usare equazioni e *disequazioni* su termini (alberi razionali, ad essere precisi). Successivamente, a metà degli anni 80, il linguaggio fu ulteriormente esteso a PROLOG III che permetteva generici vincoli su stringhe, booleani e reali (limitatamente all'aritmetica lineare). Sempre in questi anni all'università di Monash (Australia) fu sviluppato il linguaggio CLP(R), con vincoli sui numeri reali, e Jaffar e Lassez definirono gli aspetti teorici del paradigma CLP (Constraint Logic Programming). In particolare fu mostrato come tutti i vari linguaggi logici con vincoli potevano essere visti come specifiche istanze di questo paradigma e come esso ereditava tutti i principali risultati della programmazione logica. Infine Dinbas, Van Hentenryck ed altri all'ECRC (Monaco) definirono CHIP, un'estensione di PROLOG che permetteva vari tipi di vincoli, ed in particolare vincoli su domini finiti.

## 16.6 Anni '90

Gli anni '90, come tutti sappiamo hanno visto l'affermazione di Internet e del World Wide Web, due strumenti che hanno cambiato profondamente molti aspetti dell'informatica e dunque anche dei linguaggi di programmazione. La possibilità di connettere in rete milioni di dispositivi di calcolo, di condividere dati e programmi che risiedono su macchine lontane migliaia di chilometri, di trasmettere dati sensibili e accedere ad informazioni riservate mediante canali condivisi da migliaia di utenti, ha introdotto innumerevoli problematiche di efficienza, di affidabilità, di correttezza e di sicurezza che coinvolgono tutto lo spettro dei livelli presenti in un sistema informatico: da quello dei protocolli di comunicazione di basso livello, fino ai linguaggi usati per le applicazioni finali. Dal punto di vista che qui ci interessa, e cioè quello dei linguaggi di programmazione, l'aspetto più rilevante di questi anni è stato sicuramente la definizione del linguaggio Java, che abbiamo già visto nel Capitolo 12 e di cui illustriamo alcuni aspetti salienti qui sotto. Sempre di questo periodo è la definizione di HTML (HyperText Markup Language), un linguaggio di marcatura per ipertesti definito da T. Berners-Lee nel 1989, usato per la definizione delle pagine web. Sia HTML che la sua evoluzione XML, linguaggio per la rappresentazione di dati semi-strutturati, per quanto importanti non sono trattati in questo testo in quanto non sono linguaggi di programmazione in senso stretto.

<sup>3</sup>Guglielmo di Occam, frate inglese del XIV secolo, con il suo celebre "rasoio" espresse un principio alla base del pensiero scientifico moderno: è inutile formulare più ipotesi di quelle strettamente necessarie a spiegare un dato fenomeno.

**Java** Il linguaggio orientato agli oggetti Java fu sviluppato da un gruppo (il *Green team*) guidato da James Gosling alla SUN. Il progetto iniziale, iniziato nel 1990 aveva lo scopo di definire un linguaggio, ottenuto a partire da una nuova implementazione di C++, da usarsi in piccoli apparati di calcolo, con potenza relativamente ridotta, connessi ad una rete e da usarsi collegati ad un televisore che doveva fungere da periferica di ingresso e uscita. Si trattava di dispositivi che dovevano realizzare una sorta di browser ante litteram, da usarsi con finalità simili alla moderna navigazione in rete, ma senza che fosse ancora disponibile tutta la tecnologia necessaria allo scopo. In effetti il linguaggio iniziale, già disponibile in una versione funzionante nel 1992, non ebbe grande seguito. Nel 1993 però, il rilascio di Mosaic, il primo browser per Internet, fece immediatamente capire al Green team che il linguaggio al quale stavano lavorando aveva grandi potenzialità nel mondo del Web. Difatti, la banda ridotta nelle comunicazioni con i computer domestici ed il grande numero di richieste a server contenenti pagine web particolarmente popolari rendevano penoso l'uso dei primi browser. Questi problemi potevano essere risolti, almeno parzialmente, spedendo in rete piccoli programmi, gli ormai famosi *applet*, da eseguirsi presso la macchina cliente dell'utente che aveva richiesto un certo servizio. In tal modo si diminuisce il carico del server al quale il servizio era stato richiesto. Il linguaggio da utilizzarsi per la realizzazione di tali applet doveva però soddisfare due requisiti fondamentali:

**Portabilità:** ovviamente quando si spedisce un programma ad una macchina remota, normalmente non si conosce l'architettura di tale macchina. È necessario dunque che su ogni macchina che possa agire da client sia implementata una versione del linguaggio in cui sono scritti gli applet, cosa non semplice se il linguaggio è particolarmente esteso e complicato.

**Sicurezza:** Per eseguire sulla propria macchina dei programmi ricevuti dalla rete sono necessarie delle precise garanzie sulla sicurezza e l'affidabilità di tali programmi.

Java fu dunque progettato tenendo presenti questi due requisiti di base, nel contesto del paradigma orientato agli oggetti. Le soluzioni ottenute oramai ci sono note: il primo problema venne risolto definendo la *Java Virtual Machine* (JVM) ed il relativo bytecode. Un programma Java viene tradotto (compilato) in un linguaggio intermedio, detto bytecode, la cui macchina astratta è appunto la JVM. Questa macchina, più semplice da realizzare della intera macchina Java, è implementata in modo interpretativo su ogni diversa macchina fisica. Il programma Java, tradotto in bytecode, può essere quindi spedito sulla rete per essere eseguito localmente nella macchina dell'utente. La reale applicabilità di quest'approccio fu dimostrata nel 1994, quando la Sun sviluppò il browser HotJava contenente la JVM, ma fu dal 1995 che Java iniziò la sua veloce diffusione, quanto la JVM fu incorporata nel browser Netscape (che a sua volta era la reincarnazione di Mosaic).

Il problema della sicurezza, invece, fu affrontato con varie tecniche, in primo luogo usando un sistema di tipi che garantisse la type safety: l'esecuzione di un programma Java non può causare a run-time errori di tipo non segnalati (almeno limitatamente alla parte sequenziale del linguaggio). La sicurezza rispetto ai tipi è ottenuta per mezzo di controlli di tipo fatti a tre livelli: il compilatore Java,

analogamente a quelli di altri linguaggi tipizzati, non permette la traduzione di programmi che violano il sistema di tipi di Java; il bytecode risultato della compilazione è anch'esso controllato da un type checker prima dell'esecuzione; infine, a run-time, l'interprete del bytecode effettua alcuni controllo di tipo che, per la loro natura, non possono essere fatti staticamente (ad esempio, il controllo sui limiti degli array).

Un'altra importante scelta progettuale di Java, effettuata anch'essa allo scopo di migliorare l'affidabilità e la semplicità del linguaggio, è la mancanza di gestione esplicita dei puntatori e la presenza del garbage collector per il recupero della memoria non più utilizzata. Così, anche se tutti gli oggetti Java sono accessibili mediante (una versione astratta di) puntatori e, ovviamente, vi è allocazione dinamica della memoria, non esiste il tipo "puntatore" ed il programmatore può manipolare i riferimenti agli oggetti solo indirettamente, mediante assegnamento e passaggio di parametri dove sono coinvolti oggetti. A questo proposito, ricordiamo che in Java solo i valori dei tipi di base interi, booleani e stringhe non sono oggetti e che, mentre le variabili di questi tipi di base usano il value model, le variabili che rappresentano oggetti usano il reference model. Il passaggio di parametri è sempre per valore (dove si passano oggetti, il valore passato è il riferimento all'oggetto, dato che le variabili in questo caso usano il reference mode) e la regola di scoping usata è quella statica.

Oltre al lookup dinamico dei metodi, tipico dei linguaggi orientati agli oggetti, Java permette anche il caricamento dinamico delle classi: se un programma, durante l'esecuzione, invoca un metodo di una classe non presente che, ad esempio, risiede in una locazione fisica diversa sulla rete, tale classe può essere caricata dinamicamente nella Java virtual machine. Questo approccio incrementale permette di iniziare l'esecuzione di un programma quando alcune sue parti sono ancora mancanti, cosa che nell'ambito dei browser ha indubbiamente un effetto pratico positivo.

Infine, come abbiamo visto nel Paragrafo 15.7, ricordiamo che Java permette l'esecuzione concorrente di più threads. Le primitive di sincronizzazione e comunicazione, necessarie per gestire l'esecuzione dei vari processi concorrenti, formano un'importante parte del progetto di Java e contribuiscono alla portabilità del linguaggio in quanto non fanno riferimento alle operazioni di sistema di una specifica piattaforma.

Tutte queste caratteristiche positive per la sicurezza e l'affidabilità dei programmi Java hanno un costo in termini di efficienza. La presenza di controlli di tipo a run-time, l'interprete del bytecode, il garbage collector e altro ancora infatti influenzano in modo significativo il tempo d'esecuzione dei programmi. Tuttavia questa inefficienza non è particolarmente importante per il dominio applicativo tipico dei programmi Java. In particolare, nell'ambito dei browser web, sicuramente i tempi d'attesa per l'uso della rete rendono trascurabili i tempi d'esecuzione degli applet Java.

**Le librerie per la concorrenza** Nonostante lo sviluppo di numerosi linguaggi specifici per la programmazione concorrente, come quelli citati nelle precedenti

sezioni, buona parte della programmazione concorrente attuale usa un linguaggio tradizionale sequenziale, aumentato da opportune librerie che realizzano le funzionalità necessarie per gestire la concorrenza. Buona parte di queste librerie sono state definite negli anni '90.

In particolare verso la metà degli anni 90 è stata definita, nell'ambito della famiglia di standard POSIX (Portable Operating System Interface for Unix) la libreria Pthreads. Si tratta di un insieme di routine C che permettono ad un normale programma C la programmazione multithreaded mediante una decina di funzioni per la gestione e la sincronizzazione dei thread.

Due package molto usati per la programmazione concorrente con scambio di messaggi sono Parallel Virtual Machine (PVM) e Message Passing Interface (MPI). Entrambe queste librerie permettono di realizzare i processi di un ambiente distribuito usando dei programmi scritti in un linguaggio sequenziale (tipicamente C o FORTRAN) i quali poi comunicano e si sincronizzano chiamando le funzioni di PVM o MPI. Le funzionalità offerte dalle due librerie sono simili, anche se PVM permette una maggiore flessibilità nella gestione di macchine eterogenee e nella gestione dei fault, mentre MPI permette una maggior varietà di primitive di comunicazione.

Package per la programmazione distribuita che supportano lo scambio di messaggi e l'invocazione remota di metodi sono disponibili anche in Java, come abbiamo visto alla fine del Paragrafo 15.7 parlando di `java.net`, `java.rmi` e `java.rmi.server`.

Nell'ambito della teoria un contributo importante degli anni '90 è stata la definizione del  $\pi$ -calcolo ad opera di Robin Milner, Joachim Parrow e David Walker. Si tratta di un calcolo che sviluppa il lavoro iniziato con il CCS introducendo la possibilità di descrivere sistemi mobili, la cui configurazione varia dinamicamente. Variazioni ed estensioni del  $\pi$ -calcolo sono oggi usate anche per descrivere sistemi biologici ed economici.

## 16.7 Sommario del capitolo

In questo capitolo conclusivo abbiamo cercato di vedere in prospettiva storica i più importanti linguaggi di programmazione definiti sino ad oggi, cercando di capire le ragioni delle varie scelte progettuali, i motivi del successo di alcuni linguaggi e quelli del fallimento di altri.

Ovviamente, un trattazione adeguata di queste questioni richiederebbe un nuovo libro, fra l'altro di non facile realizzazione, perché se non è difficile reperire la documentazione relativa ai vari linguaggi (anche "estinti"), è spesso estremamente complicato individuare le influenze reciproche fra i formalismi proposti.

Per quanto riguarda i possibili linguaggi sequenziali di domani, i Capitoli 12–14 forniscono delle indicazioni sui paradigmi sui quali si sta concentrando attualmente la ricerca: probabilmente i linguaggi di programmazione del futuro prossimo si svilupperanno in questi ambiti, cercando di migliorare i molti punti deboli dei linguaggi attuali (alcuni spunti di riflessione in questo senso sono stati indicati nel testo).

Vi sono poi altri contesti dai quali è possibile attendersi progressi significativi. Uno, forse il più importante, è quello dei linguaggi concorrenti. In particolare, come ricordato nel Capitolo 15, tecnologie quali quelle fornite dalle Service Oriented Architectures e dal Cloud Computing promettono sviluppi molto importanti sia in ambito teorico che pratico, con ricadute applicative di vasta portata. Ma è lecito aspettarsi novità importanti per i linguaggi anche da settori quali la bio-informatica e la computazione quantistica. Anche se la tesi di Church pare destinata a resistere alle sollecitazioni che vengono da queste nuove aree, è indubbio che i meccanismi di calcolo biologici o quantistici aprono prospettive sin qui inesplorate. Si tratta comunque di sviluppi che richiederanno tempi adeguati: probabilmente dovremo lasciare ad altri, forse a uno dei giovani lettori di questo testo, il compito di dar conto delle novità che verranno.

## 16.8 Nota bibliografica

La bibliografia su questo capitolo è, come si può facilmente intuire, sterminata: ogni singolo linguaggio di programmazione è stato oggetto di numerose pubblicazioni, da quelle manualistiche a quelle di carattere più teorico. La programmazione concorrente meriterebbe poi una bibliografia a parte, visto il numero continuamente crescente di pubblicazioni scientifiche e libri didattici dedicati all'argomento (si veda anche quanto detto nel Paragrafo 15.9). Per alcuni dei principali linguaggi attualmente in uso qui ricordiamo solo [50] per C, [56, 41] per Java, [96] per C++, [70] per ML e [94] per PROLOG.

Volendosi limitare a pubblicazioni riguardanti i linguaggi di programmazione in prospettiva storica, un'ottima fonte d'informazione è costituita dagli atti della conferenza di storia dei linguaggi di programmazione, organizzata dalla Association for Computing Machinery (ACM) americana, che ha avuto due edizioni nel passato, nel 1978 [103] e nel 1993 [2], e che sarà organizzata nuovamente nel 2007. Quasi tutti i linguaggi menzionati in questo capitolo sono oggetto di una nota introduttiva in tali atti, spesso realizzata dagli stessi progettisti del linguaggio. Anche gli Annals of the History of Computing, realizzati dalla IEEE, contengono molto materiale interessante. Un resoconto storico sul software degli anni '50 e '60 è contenuto in [24].

Vi sono anche numerosi libri che offrono un panorama sui vari linguaggi esistenti, ad esempio [46] e, per i linguaggi più vecchi, [86]. Alcuni testi di argomento simile al presente volume, quali [89], [88] ed il classico [81], contengono anche brevi descrizioni dei linguaggi più importanti. Inoltre vi sono vari testi che presentano la storia dei vari aspetti del calcolo automatico, come [104]. Può essere anche interessante consultare delle monografie dedicate ai pionieri dei moderni calcolatori elettronici perché, al di là degli aspetti biografici, forniscono interessanti spunti di riflessione sulle difficoltà incontrate da coloro che, in sostanza, inventarono una nuova disciplina. Un buon esempio in questo senso è [28].

Infine, per capire i diversi punti di vista, spesso contrastanti, degli artefici della storia dei linguaggi di programmazione è sicuramente utile consultare gli

articoli originali che costoro hanno scritto. I già citati articoli di Dijkstra [33, 32, 34, 35] e quello di Wirth [109] sono alcuni dei molti esempi disponibili.

## Bibliografia

- [1] H. Abelson e G. J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, Mass., 1996.
- [2] Association for Computing Machinery (ACM). *Proceedings of the second ACM SIGPLAN conference on History of programming languages*. ACM Press, 1993.
- [3] P. Aczel. An introduction to inductive definitions. In J. Barwise, editor, *Handbook of Mathematical Logic*, pages 739–782. North-Holland, Amsterdam, 1977.
- [4] A. V. Aho, , M. Lam, R. Sethi e J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. (Seconda edizione.). Addison-Wesley, 2007. (Trad. it. *Compilatori: Principi, tecniche e strumenti*, Pearson, 2009).
- [5] M. Aiello, A. Albano, G. Attardi e U. Montanari. *Teoria della computabilità, logica, teoria dei linguaggi formali*. ETS, 1976.
- [6] P. Ancilotti e M. Boari. *Programmazione concorrente e distribuita*. McGraw-Hill, 2007.
- [7] G.R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addison-Wesley, 1999.
- [8] A. W. Appel. *Modern compiler implementation in Java*. Cambridge University Press, 1998. Il testo esiste anche nelle versioni in C e ML.
- [9] K. Apt. *From Logic Programming to Prolog*. Prentice Hall, 1997.
- [10] K. Arnold, J. Gosling e D. Holmes. *The Java Programming Language, fourth edition*. Addison-Wesley Longman Publishing Co., Inc., Boston, 2005. (Trad. it. *Il linguaggio Java: Manuale ufficiale*, quarta edizione, Pearson-Addison Wesley, 2006).
- [11] J. Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Commun. ACM*, 21(8):613–641, 1978. Lezione per il conferimento del premio Turing.
- [12] J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden e M. Woodger. Report on the algorithmic language ALGOL 60. *Commun. ACM*, 3(5):299–314, 1960.
- [13] H. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. Elsevier, 1984.
- [14] G. Birtwistle, O. Dahl, B. Myhrtag e K. Nygaard. *Simula Begin*. Auerbach Press, Philadelphia, 1973.
- [15] C. Böhm e G. Jacopini. Flow diagrams, Turing machines and languages with only two formation rules. *Commun. ACM*, 9(5):366–371, 1966.

- [16] G. Bracha. Generics in the Java programming language. Technical report, Sun Microsystems, 2004. Disponibile on-line a [java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf](http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf).
- [17] M. Broy e E. Denert, editors. *Software pioneers: contributions to software engineering*. Springer-Verlag, Berlin, 2002.
- [18] R. Cailliau. How to avoid getting schlonked by Pascal. *SIGPLAN Not.*, 17(12):31–40, 1982.
- [19] L. Cardelli. Basic polymorphic typechecking. *Science of Computer Programming*, 8(2):147–172, 1987.
- [20] L. Cardelli. Type systems. In Allen B. Tucker, editor, *The Computer Science and Engineering Handbook*. CRC Press, Boca Raton, FL, 1997.
- [21] L. Cardelli e P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.
- [22] G. Castagna. Covariance and contravariance: Conflict without a cause. *ACM Trans. on Progr. Lang. and Systems*, 17(3):431–447, May 1995.
- [23] S. Ceri, G. Gottlob e L. Tanca. *Logic Programming and Databases*. Springer-Verlag, 1989.
- [24] P. E. Ceruzzi. *A History of modern Computing*. MIT Press, 1998.
- [25] A. Church. *The Calculi of Lambda Conversion*. Princeton Univ. Press, 1941.
- [26] K. L. Clark. Predicate logic as a computational formalism. Technical Report Res. Rep. DOC 79/59, Imperial College, Dpt. of Computing, London, 1979.
- [27] J. C. Cleaveland. *An Introduction to Data Types*. Addison-Wesley, Reading, Mass., 1986.
- [28] I. B. Cohen. *Howard Aiken: Portrait of a Computer Pioneer*. The MIT Press, 2000.
- [29] R. P. Cook e T. J. LeBlanc. A symbol table abstraction to implement languages with explicit scope control. *IEEE Trans. on Software Engineering*, 9(1):8–12, 1983.
- [30] G. Cousineau e M. Mauny. *The functional approach to programming*. Cambridge Univ. Press, 1998.
- [31] S. Crespi Reghizzi. *Formal Languages and Compilation*. Springer-Verlag, 2009.
- [32] E. W. Dijkstra. Co-operating sequential processes. In F. Genuys, editor, *Programming Languages*, pages 43–112. Academic-Press, London, 1968.
- [33] E. W. Dijkstra. Go to statement considered harmful. *Communications of the ACM*, 11(3):147–148, 1968.
- [34] E. W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1:115–138, 1971.
- [35] E. W. Dijkstra. Guarded commands, non-determinacy and a calculus for the derivation of programs. In F. L. Bauer e K. Samelson, editors, *Language Hierarchies and Interfaces*, volume 46 of *Lecture Notes in Computer Science*, pages 111–124. Springer, 1975.
- [36] H. Coelho e J. C. Cotta. *Prolog by example*. Springer-Verlag, 1988.

- [37] E. Eder. Properties of substitutions and unifications. *Journal of Symbolic Computation*, 1:31–46, 1985.
- [38] C. N. Fisher e R. J. LeBlanc. The implementation of run-time diagnostics in Pascal. *IEEE Trans. Software Eng.*, 6(4):313–319, 1980.
- [39] A. Goldberg e D. Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, 1983.
- [40] J. B. Goodenough. Exception handling: issues and a proposed notation. *Commun. ACM*, 18(12):683–696, 1975.
- [41] J. Gosling, B. Joy, G. Steele e G. Bracha. *The Java Language Specification*, 3/E. Addison Wesley, 2005. Disponibile on-line a <http://java.sun.com/docs/books/jls/index.html>.
- [42] J. Herbrand. *Logical Writings*. Reidel, Dordrecht, 1971. A cura di W.D. Goldfarb.
- [43] R. Hindley e P. Seldin. *Introduction to Combinators and Lambda-calculus*. Cambridge Univ. Press, 1986.
- [44] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.
- [45] J. E. Hopcroft, R. Motwani e J.D. Ullman. *Introduction to automata theory, languages, and computation*. Addison-Wesley, 2001. (Trad. it. *Automi, linguaggi e calcolabilità*, Addison-Wesley Italia, 2003).
- [46] E. Horowitz. *Programming Languages: A Grand Tour*. Computer Science Press, 1987.
- [47] E. Horowitz e S. Sahni. *Fundamentals of data structures in Pascal*. Freeman and Company, 1994.
- [48] K. Jensen e N. Wirth. *Pascal-User Manual and Report*. Springer-Verlag, Berlin, 1991.
- [49] R. Jones e R. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons, 1996.
- [50] B. W. Kernighan e D. M. Ritchie. *The C Programming Language*. Prentice Hall, 1988.
- [51] R. A. Kowalski. Predicate logic as a programming language. *Information Processing*, 74:569–574, 1974.
- [52] P. J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964.
- [53] C. Laneve. *La Descrizione Operazionale dei Linguaggi di Programmazione*. Franco Angeli, 1998.
- [54] D. Lea. *Concurrent Programming in Java: Design Principles and Patterns*, 2nd edition. Prentice Hall, 1999.
- [55] J. Levine. *Flex & Bison*. O'Reilly Media, 2009.
- [56] T. Lindholm e F. Yellin. *The Java Virtual Machine Specification*, 2nd edition. Sun and Addison-Wesley, 1999.
- [57] B. Liskov e J. Guttag. *Abstraction and Specification in Program Development*. MIT Electrical Engineering and Computer Science Series. MIT Press, 1986.

- [58] B. Liskov, A. Snyder, R. Atkinson e C. Schaffert. Abstraction mechanisms in CLU. *Commun. ACM*, 20(8):564–576, 1977.
- [59] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987. Seconda edizione.
- [60] D. B. Lomet. Scheme for invalidating references to freed storage. *IBM Journal of Research and Development*, 19(1):26–35, 1975.
- [61] D. B. Lomet. Making pointers safe in system programming languages. *IEEE Trans. Software Eng.*, 11(1):87–96, 1985.
- [62] M. D. MacLaren. Exception handling in PL/I. In *Proc. of an ACM conf. on Language design for reliable software*, pages 101–104, 1977.
- [63] K. Marriott e P.J. Stuckey. *Programming with Constraints*. MIT Press, 1998.
- [64] A. Martelli e U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4:258–282, 1982.
- [65] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, 3(4):184–195, 1960.
- [66] J. Meyer e T. Downing. *Java virtual machine*. O'Reilly, 1997.
- [67] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.
- [68] R. Milner, Joachim Parrow e David Walker. A calculus of mobile processes, Parts I and II. *Inf. Comput.*, 100(1):1–77, 1992.
- [69] R. Milner e M. Tofte. *Commentary on Standard ML*. MIT Press, 1991.
- [70] R. Milner, M. Tofte, R. Harper e D. MacQueen. *The Definition of Standard ML - Revised*. MIT Press, 1997.
- [71] J. C. Mitchell. Type systems for programming languages. pages 365–458, 1990.
- [72] C. W. Morris. Foundations of the theory of signs. In *Writings on the Theory of Signs*, pages 17–74. Mouton, The Hague, 1938. (Trad. it. *Lineamenti di una teoria dei segni*, P. Manzi, 1999).
- [73] J. Moses. The function of FUNCTION in LISP, or why the FUNARG problem should be called the environment problem. Technical report, MIT AI Memo 199, 1970. Disponibile on-line a <http://hdl.handle.net/1721.1/5854>.
- [74] K. Nygaard e O.-J. Dahl. The development of the SIMULA languages. In *HOPL-I: The first ACM SIGPLAN conference on History of programming languages*, pages 245–272, New York, NY, USA, 1978. ACM Press.
- [75] F. Pagan. *A practical guide to Algol 68*. John Wiley & Sons, London, 1976. Wiley Series in Computing.
- [76] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972.
- [77] S. Pemberton e M. Daniels. *Pascal Implementation: The P4 Compiler and Interpreter*. Ellis Horwood, 1982.
- [78] S. Peyton-Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987. Esaurito. Versione online disponibile a <http://research.microsoft.com/Users/simonpj/papers/slpl-book-1987/index.htm>.
- [79] B. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, Mass., 2002.
- [80] G. D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.
- [81] T.W. Pratt e M.V. Zelkowitz. *Programming languages: Design and Implementation*. Prentice-Hall, 2001. (quarta edizione).
- [82] R.S. Pressman. *Principi di ingegneria del software*. McGraw Hill, 2000. (terza edizione).
- [83] B. Randell e L. J. Russell (a cura di). *Algol 60 Implementation*. Academic Press, London, 1964.
- [84] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.
- [85] H. Rogers. *Theory of recursive functions and effective computability*. McGraw Hill (ristampato da MIT Press), 1967. (Trad. it. *Teoria delle funzioni ricorsive e della calcolabilità effettiva*, Tecniche Nuove, 1992).
- [86] J. Sammet. *Programming Languages: History and Fundamentals*. Prentice-Hall, 1969.
- [87] D. A. Schmidt. *The Structure of Typed Programming Languages*. MIT Press, Cambridge, Mass., 1994.
- [88] M. L. Scott. *Programming language pragmatics*. Morgan Kaufmann Publishers, 2000.
- [89] R. Sethi. *Programming Languages: Concepts and Constructs*. Addison-Wesley, 1996. (Trad. it. *Linguaggi di programmazione*, Zanichelli, 1994).
- [90] C. Shaffer. *A practical introduction to data structures and algorithm analysis*. Addison-Wesley, 1996.
- [91] S. Sippu e E. Soisalon-Soininen. *Parsing Theory*. Springer-Verlag, 1988. 2 voll.
- [92] R. L. Sites. ALGOL W reference manual. Technical report, Stanford, CA, USA, 1972.
- [93] R. B. Smith e D. Ungar. Programming as an experience: The inspiration for Self. In *ECCOOP '95: Proc. of the 9th European Conf. on Object-Oriented Programming*, pages 303–330, Berlin, 1995. Springer-Verlag.
- [94] L. Sterling e E. Shapiro. *The Art of Prolog*. MIT Press, 1986.
- [95] B. Stroustrup. Multiple inheritance for C++. In *Proc. of the Spring 1987 European Unix Users Group Conference*, Helsinki, 1987.
- [96] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, 1997. (Trad. it. *C++ Linguaggio, libreria standard, principi di programmazione*, Addison-Wesley, 2000).
- [97] A.S. Tannenbaum. *Structured computer organization*. Prentice-Hall, 1999.
- [98] A. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proc. London Math. Soc.*, 42:230–365, 1936. A Correction, ivi, 43 (1937), 544–546.
- [99] J. D. Ullman. *Elements of ML programming*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994.

- [100] D. Ungar e R. B. Smith. Self: The power of simplicity. In *OOPSLA '87: Conf. proc. on Object-oriented programming systems, languages and applications*, pages 227–242, New York, 1987. ACM Press.
- [101] A. van Wijngaarden, B. J. Mailloux, J. E. L. Peck, C. H. A. Koster, M. Sintzoff, C. H. Lindsey, L. G. L. T. Meertens e R. G. Fisker. Revised report on the algorithmic language ALGOL 68. *Acta Informatica*, 5:1–236, 1975.
- [102] T. R. Virgilio e R. A. Finkel. Binding strategies and scope rules are independent. *Computer Languages*, 7(2):61–67, 1982.
- [103] L. Wexelblat, editor. *Proceedings of the first ACM SIGPLAN conference on History of programming languages*. ACM Press, 1978.
- [104] M. R. Williams. *A History of Computing Technology*. Prentice-Hall, 1985. Edizione rivista: ACM Press, 1997.
- [105] P. R. Wilson. Uniprocessor garbage collection techniques. In *Proc. Int. Workshop on Memory Management*, LNCS 637. Springer-Verlag, 1992. Versione estesa disponibile on-line a <ftp://ftp.cs.utexas.edu/pub/garbage/bigsurv.ps>.
- [106] G. Winskel. *The Formal Semantics of Programming Languages*. MIT Press, 1993. (Trad. it. *La semantica formale dei linguaggi di programmazione*, UTET libreria, 1999).
- [107] N. Wirth. The programming language Pascal. *Acta Informatica*, 1(1):35–63, 1971. Ristampato in [17].
- [108] N. Wirth. The module: A system structuring facility in high-level programming languages. In *Language Design and Programming Methodology*, volume 79 of *LNCS*, pages 1–24. Springer, 1979.
- [109] N. Wirth. From programming language design to computer construction. *Communications of the ACM*, 28(2):159–164, 1985. Lezione per il conferimento del premio Turing.
- [110] N. Wirth. From Modula to Oberon. *Softw. Pract. Exper.*, 18(7):661–670, 1988.
- [111] N. Wirth e C. A. R. Hoare. A contribution to the development of ALGOL. *Commun. ACM*, 9(6):413–432, 1966.

## Indice analitico

- $\epsilon$ -produzione, 93  
 $\pi$ -calcolo, 556
- A-list, *vedi* scope dinamico  
Ackermann  
funzione di, 226  
ACP, 552  
Ada, 551  
ADT, *vedi* tipo di dato astratto  
Aiken, H., 536  
albero, 34  
di derivazione, 33  
sintattico, 203  
alfabeto, *vedi* linguaggio del prim'ordine  
ALGOL, 297, 540  
aliasing, 144, 147  
alto livello, *vedi* linguaggio  
ambiente, 146, 263  
creazione di associazione, 152  
disattivazione di associazione, 152  
distruzione di associazione, 152  
dominio semantico, 214  
globale, 150  
in C, 264  
locale, 150, 427  
non locale, 150, 246  
operazioni su, 151  
riattivazione di associazione, 152  
riferimento, 152  
ambiguità, *vedi* grammatica ambigua  
analisi  
lessicale, 43  
semantica, 45  
sintattica, 45  
analizzatore  
lessicale, 71
- sintattico, 90
- APL  
valutazione espressioni, 202  
applet, 554  
applicazione, 417, 420  
aritmetica  
dei puntatori, 306  
array, 298  
allocazione, 300, 307, 343  
calcolo indirizzo, 301  
controllo dei limiti, 300  
forma, 301  
in C, 307  
in Java, 303  
multidimensionale, 298  
slice, 299  
ASCC/MARK I, 536  
assegnamento, *vedi* comando  
assembly, 537  
associazione  
creazione, 152  
disattivazione, 152  
nome-oggetto denotabile, 145  
riattivazione, 152  
riferimento, 152  
tempo di vita, 153  
astrazione, 420  
funzionale, 246  
sui dati, 243, 279, 347  
sul controllo, 243  
attesa attiva, 503  
attuale, *vedi* parametro  
automa  
a pila, 85  
DFA minimo, 70  
finito, 56  
finito deterministico, 59

finito non deterministico, 58  
 LALR(1), 122  
 LR(0), 112  
 LR(1), 120  
 Böhm, C., 216  
 Babbage, C., 551  
 backtracking, 473  
 Backus, J., 541  
 barriere, 505  
 basso livello, *vedi* linguaggio  
 Bernstein, A., 544  
 beta, 421  
 binding, 145  
   deep, 259  
   shallow, 259  
 blackboard, 501  
 blocco, 148, 215  
   di procedura, 148  
   in-line, 148  
   protetto, 268  
   record di attivazione, 170  
 BNF, 31, 541  
 booleano, 287  
 Brinch Hansen, P., 510, 549  
 Byron Lovelace, Ada, 551  
 bytecode, *vedi* Java  
 C, 544  
   ambiente, 264  
   array, 307  
   enumerazioni, 290  
   passaggio per riferimento, 251  
   unione, 294  
 C++, 550  
   classe, 366  
   ereditarietà, 378  
   funzione virtuale, 382  
   passaggio per riferimento, 323  
   selezione dinamica dei metodi, 382  
   sottotipi, 378  
   template, 323  
 calcolabilità, 135  
 calcolo dei predicati, 454  
 Cambridge, *vedi* notazione  
 campo

di un record, 291  
 mascheramento, 371  
 canale, 515  
 carattere, 287  
 case, *vedi* comando  
 cast, 318, 319  
   non-converting type, 319  
   verso il basso, 388  
 catena dinamica, 172  
   puntatore di, 172  
 catena statica, *vedi* scope statico  
 cattura di variabile, 255  
 CCS, 533  
 Central Referencing Environment, *vedi*  
   tabella centrale dell'ambiente  
 chiamata di procedura, 175  
   prologo, 175  
   sequenza di chiamata, 175  
 CHIP, 553  
 chiusura, 256, 261, 265  
   di Kleene, 30, 53  
 Chomsky, N., 29, 93  
 Church, A., 140  
 classe, 366  
   astratta, 372  
   base fragile, 388  
   derivata, 370  
   implementazione, 385  
 clausola, 456, 458  
   come procedura, 469  
   corpo, 457  
   definita, 458  
   testa, 457  
   unitaria, 456  
 cloud computing, 497  
 CLP, 484, 553  
 COBOL, 542  
 codice spaghetti, 230  
 coercion, *vedi* coercizione  
 coercizione, 318, 321  
 collaterale, *vedi* effetto  
 Colmerauer, A., 548  
 comando, 208  
   ;, 215  
   case, 219  
   do, 223

for each, 228, 403  
 for, 223  
 goto, 215  
 if, 218  
 repeat, 223  
 while, 222  
 assegnamento, 210  
   in C, 212  
 blocco, 215  
 composto, 215  
 con guardia, 523  
 condizionale, 218  
   ambiguità sintattica, 39  
 iterativo, 222  
 iterazione determinata, 223  
 iterazione indeterminata, 222  
 sequenziale, 215  
 strutturato, 230  
 compatibilità  
   tra tipi, 283, 316  
 compattazione, *vedi* heap  
 compilatore, 15, 42  
 compilazione  
   tempo di, 145  
 compile-time, *vedi* compilazione  
 complesso, 288  
 composizione parallela, 525  
 comunicazione  
   asincrona, 502  
   meccanismi di, 499  
   memoria condivisa, 500  
   nei thread Java, 529  
   scambio di messaggi, 500  
   sincrona, 502, 518  
 concatenazione, 53  
 concorrenza, 493, 496  
   fisica, 496  
   logica, 496, 498  
 configurazione  
   di un PDA, 85  
 conflitto  
   in parser LR, 116  
 conflitto di nomi, 377  
 constraint, *vedi* vincolo  
 contatori dei riferimenti, 334  
 continuazione, 238  
 controllo  
   di tipo, 327  
   dinamico, 284  
   statico, 284  
 controllore dei tipi, 281, 327  
 controvariante, 407  
 conversione di tipo, 318  
   esplicita, 319  
   implicita, 318  
 coppia puntata, 199  
 corto circuito, *vedi* espressione  
 costruttore, 352, 374  
   di tipo, 312  
 covariante, 407  
 CRT, *vedi* tabella centrale dell'ambiente  
 CSP, 484, 513, 518, 533  
 Curry, H., 428  
 cut, *vedi* PROLOG  
 Dahl, O.-J., 543  
 dangling reference, 308, 330  
 Datalog, 483  
 deadlock, 504, 508, 519  
 deallocazione, 307, 329  
 definizione  
   induttiva, 233  
   regolare, 55  
 delega, 369  
 denotabile, 284  
   oggetto, 145  
 dereferenziazione  
   di puntatore, 305  
 derivazione, 32  
   sinistra, 84  
 DFA, 59  
   minimizzazione, 67  
 diagramma di transizione, 56  
 dichiarazione, 146  
   scope, 156  
 Dijkstra, E. W., 216, 507, 533, 557  
 discriminante, 294  
 dispatch, 380  
   multiplo, 384  
 display, *vedi* scope statico  
 do, *vedi* comando

don't know, *vedi* non determinismo  
 dope vector, 303  
 Dylan, 369

eager, *vedi* espressione, *vedi* valutazione per valore  
 eccezione, 266  
 implementazione, 271

Eckert, J.P., 535

EDSAC, 535

EDVAC, 535

effetto collaterale, 205

emulazione, 10, 12

ENIAC, 536

enumerazione, 289

equazione  
     fra termini, 462

equivalenza, 314  
     per nome, 313  
     strutturale, 313  
     tra tipi, 283, 313

ereditarietà, 375  
     con condivisione, 396  
     con replicazione, 395  
     di implementazione, 375  
     di interfaccia, 375  
     implementazione, 384  
     mix-in, 376  
     multipla, 377, 392

esecuzione  
     tempo di, 145

espressione, 198  
     notazione  
         infissa, 198  
         polacca, 199  
         polacca di Cambridge, 200  
         polacca inversa, 200  
         postfissa, 200  
         prefissa, 199  
     rappresentazione ad albero, 203  
     regolare, 53  
     sintassi, 198  
     valutazione, 201  
         albero, 203  
         APL, 202  
         associatività, 201

corto circuito, 207  
 delle sotto espressioni, 205  
 eager, 206  
 infissa, 201  
 lazy, 206  
 postfissa, 203  
 precedenza, 201  
 prefissa, 202  
 espressività  
     dell'iterazione determinata, 226  
 esprimibile, 284

fairness, 505

fallimento, *vedi* programma logico  
 fatto, *vedi* clausola  
 fattorizzazione sinistra, 99  
 Fibonacci, 231  
     successione di, 166  
 filosofi a cena, 508  
 firmware, 10  
 First, 102  
 float, 288  
 Follow, 103  
 for, *vedi* comando  
 for each, *vedi* comando  
 forma normale  
     di Chomsky, 93  
     di Greibach, 93  
 forma sentenziale, 84  
 formale, *vedi* parametro  
 formula, *vedi* linguaggio del prim'ordine  
     fortemente tipizzato, 283  
 FORTRAN, 537, 540  
 fragile  
     classe base, 388  
 frame, *vedi* record di attivazione  
 frammentazione, 179  
     esterna, 179  
     interna, 179  
 funarg problem, 275  
 funzione, 244  
     calcolabile, 139  
     come parametro, 258  
     come risultato, 264  
     covariante, 407

di ordine superiore, 258  
 parziale, 13  
 record di attivazione, 172  
 virtuale in C++, 382

Futamura  
     proiezione di, 26

Gödel, K., 139

garbage collection  
     copia, 339  
     mark and compact, 338  
     mark and sweep, 336  
     reference counts, 334  
     stop and copy, 339  
 garbage collector, 308, 332

Gauss-Jordan, 484

generico, 326, 398, 401  
     implementazione, 405

gerarchia di Chomsky, 89

gerarchia di macchine astratte, 21

gestore, 268

GHC, 552

goal, 456  
     valutazione, 471

Gosling, J., 553

goto, *vedi* comando  
 grammatica, 27  
     a struttura di frase, 89  
     ambigua, 38, 84, 125  
     aumentata, 113  
     contestuale, 41  
     dipendente dal contesto, 89  
 LALR(1), 122  
 libera da contesto, 30  
 LL(1), 105  
 LR(0), 116  
 LR(1), 120  
 LR(k), 119  
 regolare, 45, 65  
 SLR(1), 118

Greibach, S., 93

ground, *vedi* termine  
 guarded commanda, 523  
 guardia, 523

handle, 111

Hanoi, *vedi* torri di Hanoi  
 Haskell, 284, 313  
 heap, 177, 266  
     buddy system, 180  
     compattazione della memoria, 180  
     Fibonacci heap, 180  
     lista libera singola, 180  
     liste libere multiple, 180  
     blocchi  
         di dimensione fissa, 177  
         di dimensione variabile, 178  
     gestione, 177

Herbrand  
     universo di, 467

Herbrand, J., 450

Hoare, C. A. R., 510, 533, 549

Hoare, C.A.R., 540

Hopper, G., 542

HTML, 553

Ichbiah, J., 551

if, *vedi* comando  
 implementazione, 9, 28, 349  
     caso reale, 18  
     compilativa, 19  
     compilativa pura, 15  
     corretta, 353  
     di Pascal, 26  
     interpretativa, 19  
     interpretativa pura, 14  
 indecidibilità, 135  
     del controllo statico di tipo, 286  
     del problema della fermata, 137

independenza  
     dalla rappresentazione, 354

indice  
     tipo, 298

induzione, *vedi* principio d'induzione

inferenza di tipo, 327, 403

information hiding, 352, 359

interfaccia, 349

interleaving, 525

intero, 287

interprete, 3, 14  
     di una macchina astratta, 3

interruzione, 495

intervallo, 285, 290  
 item  
   LR(0), 112  
   LR(1), 119  
   nucleo LR(1), 122  
 iterazione, *vedi* comando determinata, *vedi* comando indeterminata, *vedi* comando

Jacopini, G., 216  
 Jaffar, J., 553  
 Java  
   Runnable, 526  
   Socket, 532  
   notifyAll, 531  
   notify, 531  
   run, 526  
   start, 526  
   synchronized, 529  
   wait, 531

array, 303  
 Byte Code, 21  
 bytecode, 554  
 classe, 366  
 ereditarietà, 376  
 generico, 401  
 interfaccia, 376  
 macchina intermedia, 21  
 RMI, 532  
 sottotipi, 376  
 storia, 553  
 thread, 526  
 wildcard, 405, 407

Java Virtual Machine, 21, 389, 554  
 Javascript, 369  
 JOLIE, 497  
 jump table, *vedi* tabella di salto  
 JVM, *vedi* Java Virtual Machine

Kay, A., 543, 546  
 Kleene, S., 139  
 Kowalski, R., 548

l-valore, 211, 304  
 lambda calcolo, 440  
 Landin, P., 240, 434

lazy, *vedi* espressione, *vedi* valutazione  
   lazy  
 legame  
   deep, 259  
   shallow, 259  
 lemma  
   pumping per liberi, 88  
   pumping per regolari, 77  
 lessema, 52  
 Lex, 71  
 Linda, 501, 552  
 linguaggio  
   accettato da un NFA, 59  
   applicativo, 214  
   assembly, 537  
   del prim'ordine, 454  
     alfabeto, 454  
     formula, 456  
     termine, 455  
     termine ground, 455  
   di alto livello, 5, 537  
   di basso livello, 5, 536  
   di ordine superiore, 258  
   di prima generazione, 536  
   di seconda generazione, 537  
   di terza generazione, 537  
   dichiarativo, 216  
   dipendente dal contesto, 89  
   formale, 30, 53  
   funzionale, 427  
   generato da una grammatica, 33  
   imperativo, 214, 216  
   libero, 83  
   macchina, 5  
   semidecidibile, 89  
 link di controllo, *vedi* puntatore di catena dinamica  
 link dinamico, *vedi* puntatore di catena dinamica  
 LISP, 199, 542  
   cons, 199  
   effetti collaterali, 433  
   garbage collector, 332  
 lista, 199  
   in PROLOG, 475  
   libera, 180

LL(k), 107  
 locks and keys, 331  
 logica del prim'ordine, 454  
 lookahead, 107  
 lookup dinamico dei metodi, 380  
 LR(k), 119  
 lucchetti e chiavi, 331  
 Lukasiewicz, W., 199  
 m.g.u., *vedi* unificatore più generale  
 macchina  
   analitica, 551  
   astratta, 2  
     gerarchia di, 21  
     hardware, 6  
     interprete, 3  
     memoria, 2  
     realizzazione, 10  
   di Turing, 18, 139  
   di von Neumann, 416  
   hardware, *vedi* macchina astratta  
 intermedia, 18, 19  
   Java, 21  
   Pascal, 21  
   per differenze, 551  
   SECD, 434  
 maniglia, 111  
 mark and compact, 338  
 mark and sweep, 336  
 MARK I, 536  
 mascheramento, 371  
 massimo parallelismo, 525  
 Mauchly, J., 535  
 McCarthy, J., 542  
 meccanismi di naming, 513  
 memoria, 165  
   dominio semantico, 214  
   gestione dinamica a pila, 168  
   gestione dinamica con heap, 176  
   gestione statica, 167  
 memorizzabile, 284  
 Menabrea, L., 551  
 method overriding, 371  
 metodo, 364  
   binario, 410  
   lookup dinamico, 380  
 ridefinizione, 371  
 statico, 368  
 microprogrammazione, 10  
 Milner, R., 533, 549  
 minimizzazione di DFA, 67  
 mix-in, 376  
 ML, 284, 313, 325, 417, 547  
   reference cell, 432  
   tipi ricorsivi, 312  
 Modula, 549  
 modulo, 354, 520  
 monitor, 510  
   signal, 512  
   wait, 512  
   variabile condizionale, 511, 512  
 monomorfo, 320  
 multimedodo, 384  
 mutua esclusione, 503  
   lock, 503  
 name clash, 377  
 naming, *vedi* ambiente  
 Naur, P., 540  
 negazione, 481  
   come fallimento, 481  
 NFA, 58  
 nome, 143  
 non determinismo, 472  
 notazione  
   polacca, 199  
   polacca di Cambridge, 200  
   polacca inversa, 200  
   postfissa, 200  
   prefissa, 199  
 Nygaard, K., 543  
 Occam, 515, 549, 552  
 occultamento  
   dell'informazione, 352, 359  
 oggetto, 364  
   allocazione, 368  
   denotabile, 145  
   denotabile, accesso, 153  
   denotabile, creazione, 152  
   denotabile, distruzione, 153  
   denotabile, modifica, 153

denotabile, tempo di vita, 153  
 implementazione, 384  
 opaco, *vedi* equivalenza per nome  
 operatore  
     associatività, 201  
     non definito, 206  
     precedenza, 201  
 ordinale, 291  
 ordine  
     di colonna, 300  
     di riga, 300  
 ordine superiore, 258, 420  
 overloading, 321, 381  
 overriding, 371  
     controvariante, 410  
 P-code, 21, 545  
 package, 354  
 paradigma  
     concorrente, 493  
     funzionale, 415  
     logico, 449  
     orientato agli oggetti, 359  
 parametro, 244  
     passaggio, 247  
     per costante, 250  
     per nome, 254  
     per riferimento, 249  
     per risultato, 252  
     per valore, 248  
     per valore-risultato, 253  
 PARLOG, 552  
 parser, 45, 90  
     bottom-up, 108  
     discesa ricorsiva, 101  
 LALR(1), 121  
 LR, 115  
 LR(1), 119  
 predittivo, 105  
 shift-reduce, 109  
 SLR(1), 117  
 top-down, 100  
 Pascal, 285  
     equivalenza per nome, 315  
     record variante, 293  
 Pascal, B., 545

passaggio  
     per riferimento, 323  
 passaggio dei parametri, *vedi* parametro  
 pattern, 52  
 pattern matching, 429  
 PDA, 85  
     deterministico, 87  
 Peano, G., 233  
 pigeonhole principle, 77  
 pila di sistema, 170  
 pointer reversal, 337  
 polacca, *vedi* notazione  
 polimorfismo, 320, 382  
     di inclusione, 324  
     di sottotipo, 324  
     di sottotipo nei linguaggi oo, 398  
     esplicito, 323  
     implementazione, 326  
     implicito, 323  
     parametrico, 321  
     universale, 321  
     universale di sottotipo, 324  
     universale parametrico, 321  
 porta, 514  
 pragmatica, 28, 49  
 predicato, 451  
     definizione di un, 457  
 preemption, 527  
 prefisso viabile, 111  
 preordine, 314  
 principio d'induzione, 233  
 principio della piccionaia, 77  
 problema  
     della fermata, 135  
     indecidibile, 138  
 procedura, *vedi* funzione  
     record di attivazione, 172  
 processo, 494  
     heavyweight, 494  
     lightweight, 494  
 produzione, 30  
      $\epsilon$ , 93  
 programma logico, 456  
     fallimento, 471  
     interpretazione dichiarativa, 468  
     interpretazione procedurale, 468

modello computazionale, 467  
 successo, 471  
 programmazione  
     concorrente, 493  
     distribuita, 498  
     in grande, 231, 354  
     logica, 449, 450  
     con vincoli, 484  
     interpretazione dichiarativa, 468  
     interpretazione procedurale, 468  
     modello computazionale, 467  
 multithreaded, 498  
 parallela, 496  
 sequenziale, 493  
 strutturata, 229  
 PROLOG, 478, 548  
     aritmetica, 479  
     backtracking, 473  
     controllo, 472  
     cut, 480  
     disgiunzione, 481  
     fallimenti, 474  
     modello computazionale, 467  
     negazione, 481  
     non determinismo, 472  
     occur check, 465  
     regola di selezione, 471  
     scelta delle clausole, 473  
     storia, 548  
 PROLOG II, 553  
 PROLOG III, 553  
 pumping lemma  
     linguaggi liberi, 88  
     linguaggi regolari, 77  
 puntatore, 304  
     rovesciamento, 337  
 puntatore di catena dinamica, 172  
 punto fisso, 444  
 query, *vedi* goal  
 r-valore, 211  
 RdA, *vedi* record di attivazione  
 reale, 288  
 receive, 500  
 record, 291  
 variante, 293, 329  
 record di attivazione, 170  
     per blocco in-line, 170  
     per funzioni, 172  
     per procedure, 172  
     puntatore di catena dinamica, 172  
     puntatore di catena statica, 183  
 redex, 421  
 reference  
     dangling, 308, 330  
 reference counts, 334  
 reference model, *vedi* variabile a riferimento  
 referencing, *vedi* ambiente  
 referencing environment, *vedi* ambiente  
 regola  
     beta, 421  
     di copia, 254, 421  
     di visibilità, 149  
 regolare  
     definizione, 55  
     espressione, 53  
 relazione, 314  
     ben fondata, 233  
 renaming, *vedi* sostituzione  
 rendez-vous, 519  
 repeat, *vedi* comando  
 residuo, 425  
 restrizione, *vedi* sostituzione  
 resumption  
     di un'eccezione, 269  
 ricorsione, 231  
     efficienza, 239  
     in coda, 233  
     sinistra, 98  
 rideonoma, *vedi* sostituzione  
 riduzione, 419  
 ripresa  
     di un'eccezione, 269  
 riscrittura, 419, 420  
 risolutore di vincoli, 484  
 risoluzione, 449  
     SLD, versione informale, 471  
 risposta calcolata, 470, 473  
 Ritchie, D., 544  
 Robinson, A., 450, 548

root set, 336  
Roussel, P., 548

rovesciamento dei puntatori, 337  
RPC, 519

run-time, *vedi* esecuzione

S-espressione, 199

scalare, 286

scanner, 71

Scheme, 284, 313

scope, 154

buchi nello, 149  
di una dichiarazione, 156

dinamico, 157

A-list, 189

CRT, 191

implementazione, 188

implementazione, 181

regole di, 154

statico, 155, 261

catena statica, 183

display, 186

implementazione, 181

segnatura, 349, 455

Self, 369

semafori, 507

P, 507

V, 507

semantica, 28, 47

denotazionale, 48

operazionale, 48

statica, 40, 45, 89

send, 500

sequenza di chiamata, 175

sequenziale, *vedi* comando

Service Oriented computing, 497

shadowing, 371

shape, 301

sicuro

rispetto ai tipi, 282, 283, 297, 328,  
407

side effect, *vedi* effetto collaterale

signature, *vedi* segnatura

simbolo

inutile, 94

Simula, 543

classe, 366  
simulazione, 11

sincronizzazione  
attesa attiva, 503  
basata sullo scheduler, 506

condizionale, 503, 512  
meccanismi di, 501

mutua esclusione, 503  
sintassi, 27, 28

astratta, 42  
espressione  
albero, 203

sistema  
di tipi, 283

SLD, *vedi* risoluzione  
slice, 299

Smalltalk, 546  
classe, 366

sottotipo, 372

SOA, 497

SOC, 497

sostituzione, 459

applicazione, 460  
composizione, 460

più generale, 461  
renaming, 461

restrizione, 461  
ridenomina, 461

sottoclasse, 370

sottoprogramma, *vedi* funzione

sottotipo, 317, 370

sovraaccaricamento, *vedi* overloading

spazio delle tuple, 501

specificia

di un ADT, 352

speedup, 496

SR, 552

static

in C, 246

stato, 415

stop and copy, 339

strategia, *vedi* valutazione

strategia di valutazione, 423

stream, 431

stringa, 53

strongly typed, 283

Stroustrup, B., 550  
stub, 521  
successo, *vedi* programma logico  
supporto a run time, 20

tabella  
centrale dell'ambiente, 191

di salto, 219  
di transizione, 58

LALR(1), 122

LL(1), 104

LR, 115

LR(0), 116

LR(1), 120

tag, 294

tail recursion, 233

template, 323

tempo

di vita, 153, 246, 266

teorema

di Böhm e Jacopini, 216

termine, 455

ground, 455

tesi di Church, 140

Thompson, K., 544

thread, 493

thunk, 264

tipizzazione

dinamica, 284

statica, 284

tipo, 279, 428

base di un intervallo, 290

compatibilità, 316

composto, 291

controllo, 327

controllore di, 281

conversione, 318

conversione esplicita, 319

delle funzioni, 312

di dato astratto, 348, 359

discreto, 291

equivalenza, 313

indice, 298

inferenza, 327, 403

intervallo, 285

monomorfo, 320

ordinale, 291  
ricorsivo, 310  
ricorsivo in ML, 312  
scalare, 286  
semplice, 286  
void, 289

token, 51

tombstone, 330

torri di Hanoi, 477

trasformazione di programmi, 24

trasparente, *vedi* equivalenza strutturale

Turing completo, 139

Turing, A., 138, 139

type checker, 281, 327

type safe, 282, 283, 328

uguaglianza

fra termini, 462

unificatore, 462

più generale, 462

unificazione, 431, 464

algoritmo Martelli Montanari, 464

teoria della, 457

unione, *vedi* variante, 329

universo di Herbrand, *vedi* Herbrand

unnaming, *vedi* ambiente

valore

denotabile, 214, 216, 284

esprimibile, 214, 216, 284

funzionale, 421

memorizzabile, 214, 216, 284

valutazione, 420, 421

applicativa, 424

by need, 425

eager, 424

lazy, 425

leftmost, 423

normale, 424

parziale, 24

per nome, 424

per valore, 424

strategia, 423

variabile, 209, 415, 457

a riferimento, 210, 305

catturata, 255

di classe, 368  
di istanza, 364  
mascheramento, 371  
esterna, 264  
globale  
allocazione statica, 167  
locale  
nel record di attivazione, 171  
logica, 457  
modificabile, 209  
nei linguaggi funzionali, 415  
nei linguaggi imperativi, 209  
nei linguaggi logici, 457  
static, 246  
variante, 293  
in C, 294  
in Pascal, 293  
varianza, 461  
vettore, *vedi* array  
vincolo, 484  
sintattico contestuale, *vedi* semantica statica  
visibilità, 149  
void, 289  
von Neumann, J., 416, 535  
vtable, 386  
  
Warren Abstract Machine, 488  
while, *vedi* comando  
Wilkes, M., 535  
Wirth, N., 540, 545, 549  
WS-BPEL, 497  
  
XML, 553  
  
zucchero sintattico, 199