

## CAPITOLO 7

# Livello del linguaggio assemblativo

Mentre nei Capitoli 4, 5 e 6 abbiamo illustrato tre diversi livelli che si trovano nella maggior parte degli odierni calcolatori, qui affronteremo principalmente un ulteriore livello, presente anch'esso su praticamente tutti i calcolatori moderni, quello del linguaggio assemblativo. La differenza principale che contraddistingue questo livello rispetto ai livelli di microarchitettura, ISA e macchina del sistema operativo è che il livello del linguaggio assemblativo è implementato mediante traduzione invece che interpretazione.

I programmi che si occupano di tradurre in un particolare linguaggio un programma scritto dall'utente in un altro linguaggio sono chiamati traduttori. Il linguaggio nel quale è scritto il programma originale viene chiamato linguaggio sorgente, e linguaggio destinazione quello nel quale viene convertito. Sia il linguaggio sorgente sia il linguaggio destinazione definiscono dei livelli. Se esistesse un processore in grado di eseguire direttamente i programmi scritti nel linguaggio sorgente, tradurre il programma originale nel linguaggio destinazione non sarebbe necessario.

Si usa la traduzione quando si ha a disposizione un processore (un processore hardware oppure un interprete) per il linguaggio destinazione, ma non per il linguaggio sorgente. Se la traduzione viene effettuata correttamente, l'esecuzione del programma tradotto fornisce esattamente lo stesso risultato che si otterrebbe dal programma sorgente se si avesse a disposizione un processore in grado di eseguirlo. Di conseguenza è possibile implementare un nuovo livello per il quale non esiste un processore, traducendo inizialmente i suoi programmi in un livello destinazione ed eseguendo poi i programmi del livello destinazione ottenuti dalla traduzione.

È importante notare la differenza che esiste fra traduzione, da una parte, e interpretazione, dall'altra. Nella traduzione il programma originale non viene eseguito direttamente, ma viene invece convertito in un programma equivalente chiamato **programma oggetto** o **programma eseguibile binario** la cui esecuzione è portata avanti solo dopo che la traduzione è stata completata. La traduzione viene realizzata in due passi distinti:

1. generazione di un programma equivalente nel linguaggio destinazione;
2. esecuzione del programma appena generato.

Questi due passi non sono simultanei: il secondo non inizia finché il primo non sia stato completato. Al contrario, nell'interpretazione esiste un unico passo: l'esecuzione del programma sorgente originale. Non occorre generare, in una fase iniziale, alcun programma equivalente, anche se a volte il programma sorgente può essere convertito in una forma intermedia (per esempio il bytecode di Java) per facilitarne l'interpretazione.

Mentre si esegue il programma oggetto sono coinvolti solamente tre livelli: il livello di microarchitettura, il livello ISA e il livello macchina del sistema operativo. Di conseguenza, durante l'esecuzione è possibile trovare nella memoria del calcolatore tre diversi programmi: il programma oggetto dell'utente, il sistema operativo e il microprogramma (se esiste), mentre è svanita qualsiasi traccia del programma sorgente originale. Il numero di livelli presenti al momento dell'esecuzione può dunque essere diverso dal numero di livelli presenti prima di effettuare la traduzione. Occorre tuttavia tener presente che, mentre noi definiamo un livello in base alle istruzioni e ai costrutti linguistici resi disponibili ai suoi programmatori (e non solo in base alle tecniche implementative), altri autori distinguono nettamente i livelli implementati dagli interpreti che lavorano a tempo di esecuzione dai livelli implementati mediante la traduzione.

## 7.1 Introduzione al linguaggio assemblativo

A seconda della relazione che intercorre tra il linguaggio sorgente e il linguaggio destinazione è possibile dividere sommariamente i traduttori in due gruppi. Quando il linguaggio sorgente è essenzialmente una rappresentazione simbolica di un linguaggio macchina numerico, il traduttore è chiamato **assemblatore** e il linguaggio sorgente è chiamato **linguaggio assemblativo**. Quando il linguaggio sorgente è un linguaggio ad alto livello come Java oppure C e il linguaggio destinazione è un linguaggio macchina numerico oppure una sua rappresentazione simbolica: in questo caso il traduttore viene chiamato **compilatore**.

### 7.1.1 Che cos'è un linguaggio assemblativo

Un puro linguaggio assemblativo è un linguaggio nel quale ciascuna istruzione produce esattamente un'istruzione macchina. In altre parole esiste una corrispondenza uno-a-uno tra le istruzioni macchina e le istruzioni del programma assemblativo. Se ciascuna linea del linguaggio assemblativo contiene esattamente un'istruzione macchina, un programma assemblativo di  $n$  linee genererà un programma in linguaggio macchina di  $n$  parole.

La ragione dell'utilizzo del linguaggio assemblativo al posto del linguaggio macchina (in binario o esadecimale) è che è molto più facile programmare utilizzando il linguaggio assemblativo. L'uso di una forma simbolica per i nomi e gli indirizzi, al posto della rappresentazione binaria oppure ottale, costituisce un'enorme differenza. La maggior parte dei programmatori è in grado di ricordare che ADD, SUB, MUL e DIV sono le abbreviazioni di aggiungi, sotrai, moltiplica e dividi, mentre pochi riescono a ricordarsi i corrispondenti valori numerici utilizzati dalla macchina. Il programmatore in lin-

guaggio assemblativo deve ricordarsi solamente i nomi simbolici, dato che sarà compito dell'assemblatore tradurli in istruzioni macchina.

La stessa osservazione vale per gli indirizzi. Un programmatore può assegnare nomi simbolici alle locazioni di memoria e lasciare all'assemblatore il compito di determinare i corrispondenti valori numerici. Programmando direttamente in linguaggio macchina è invece obbligatorio lavorare sempre con i valori numerici degli indirizzi. Per questi motivi al giorno d'oggi nessuno programma più in linguaggio macchina, anche se alcuni decenni fa, prima che venissero inventati gli assemblatori, era necessario farlo.

Il linguaggio assemblativo, oltre alla corrispondenza uno-a-uno tra le sue istruzioni e le istruzioni macchina, gode di un'altra proprietà che lo contraddistingue dai linguaggi ad alto livello. Il programmatore in linguaggio assemblativo ha accesso a tutte le funzionalità e a tutte le istruzioni disponibili nella macchina di destinazione, mentre il programmatore in linguaggio ad alto livello non ha la stessa libertà di accesso. Se per esempio la macchina di destinazione ha un bit di overflow, un programma assemblativo può testarlo, mentre un programma Java non lo può fare direttamente. Un programma assemblativo, diversamente da uno ad alto livello, può eseguire qualsiasi istruzione dell'insieme delle istruzioni della macchina di destinazione. In poche parole, tutto ciò che può essere fatto in linguaggio macchina, può anche essere fatto in linguaggio assemblativo. Al contrario, un programmatore in linguaggio ad alto livello non ha a disposizione tutte le istruzioni, i registri e le funzionalità della macchina. I linguaggi per la programmazione di sistemi, come C, sono spesso trasversali rispetto a questi due tipi, dato che la loro sintassi è ad alto livello, ma forniscono allo stesso tempo alcune delle possibilità di accesso alla macchina tipiche dei linguaggi assemblativi.

Un'ultima differenza che vale la pena rendere più esplicita è che i programmi in linguaggio assemblativo vengono scritti per una specifica famiglia di macchine, mentre un programma scritto in un linguaggio ad alto livello può essere eseguito su molte macchine diverse. La possibilità di portare il software da una macchina a un'altra rappresenta un grande vantaggio per molte applicazioni.

### 7.1.2 Perché usare il linguaggio assemblativo

Programmare in linguaggio assemblativo è difficile. Non ci si faccia illusioni. Non è un lavoro per programmatore pavidi e debolucci. Inoltre scrivere un programma in linguaggio assemblativo richiede molto più tempo che scriverlo in un linguaggio ad alto livello, e anche la correzione degli errori e l'aggiornamento del codice sono operazioni molto più complesse.

Ma allora, perché mai si dovrebbe programmare in linguaggio assemblativo? Esistono due motivi: le prestazioni e le possibilità di accesso alla macchina. Prima di tutto un esperto programmatore in linguaggio assemblativo, lavorando molto duramente, può in alcuni casi creare codice più piccolo e veloce di quanto possa fare un programmatore in linguaggio ad alto livello. Per alcune applicazioni la velocità e la dimensione rappresentano due fattori critici; in questa categoria ricadono per esempio il codice di una smart card o di una RFID card, i driver dei dispositivi, le librerie per la manipolazione di stringhe, le routine del BIOS, i cicli interni di applicazioni in tempo reale le cui prestazioni hanno un'importanza fondamentale.

In secondo luogo, alcune procedure devono accedere in modo completo all'hardware (cosa che in genere è impossibile con i linguaggi ad alto livello). In questa categoria ritroviamo per esempio i gestori a basso livello degli interrupt e delle eccezioni dei sistemi operativi e i controllori dei dispositivi di molti sistemi integrati che funzionano in tempo reale.

Un compilatore deve produrre un risultato utilizzabile da un assemblatore oppure eseguire esso stesso il processo di assemblaggio. Comprendere il linguaggio assemblativo è quindi essenziale per capire come lavorano i compilatori.

In secondo, lo studio del linguaggio assemblativo permette di avere una visione più chiara del funzionamento delle macchine reali. Per gli studenti di architettura degli elaboratori scrivere anche poche righe di codice assemblativo è l'unico modo per avere consapevolezza di che cosa sono realmente le macchine a livello di microarchitettura.

### 7.1.3 Formato delle istruzioni del linguaggio assemblativo

La struttura delle istruzioni di un linguaggio assemblativo rispecchia la struttura delle istruzioni macchina, delle quali le prime forniscono una rappresentazione simbolica; ciononostante i linguaggi assemblativi di macchine diverse e di livelli differenti hanno un numero sufficiente di somiglianze da permettere di parlare di linguaggio assemblativo in termini generali. La Figura 7.1 mostra alcuni frammenti del linguaggio assemblativo x86 che esegue l'istruzione  $N = I + J$ . Nell'esempio le istruzioni sopra la linea vuota eseguono la computazione e quelle al di sotto sono invece i comandi che indicano all'assemblatore di riservare spazio di memoria per le variabili  $I$ ,  $J$  e  $N$  e non rappresentano alcuna istruzione macchina.

Etichetta	Codice operativo	Operandi	Commenti
FORMULA:	MOV ADD MOV	EAX,I EAX,J N,EAX	; registro EAX = I ; registro EAX = I + J ; N = I + J
I	DD	3	; riserva 4 byte inizializzati a 3
J	DD	4	; riserva 4 byte inizializzati a 4
N	DD	0	; riserva 4 byte inizializzati a 0

Figura 7.1 Esecuzione di  $N = I + J$  su x86.

Esistono differenti assemblatori per la famiglia Intel (per esempio x86), ciascuno dei quali ha una propria sintassi. Nel corso di questo capitolo utilizzeremo per i nostri esempi il linguaggio assemblativo Microsoft MASM. Ci sono diversi assemblatori anche per ARM, ma la sintassi è simile e quella di x86 e per questa ragione ci limiteremo a un solo esempio.

Le istruzioni del linguaggio assemblativo sono composte da quattro parti: un campo etichetta, un campo per l'operazione (codice operativo o *opcode*), un campo per gli operandi e un campo per i commenti. Le etichette sono utilizzate per fornire nomi sim-

bolici agli indirizzi, e sono necessarie per definire le destinazioni alle quali portano le istruzioni che effettuano salti e per poter accedere alle parole di dati memorizzate mediante nomi simbolici. Se un'istruzione è etichettata, l'etichetta inizia (di solito) nella colonna 1.

L'esempio della Figura 7.1 ha quattro etichette: FORMULA, I, J e N. Si noti che MASM richiede che l'etichetta termini con il carattere "due punti" quando è utilizzata per il codice, ma non quando è utilizzata per i dati. Non è un aspetto fondamentale, comunque, e altri assemblatori si comportano in maniera differente. Non c'è alcun aspetto dell'architettura sottostante che suggerisca una scelta piuttosto che un'altra. Un vantaggio offerto dalla notazione che usa i "due punti" è che un'etichetta può comparire, da sola, su una linea, mentre il codice operativo viene scritto nella prima colonna della linea successiva. Questo stile si rivela in un certo modo utile per i compilatori. Senza i "due punti" sarebbe impossibile distinguere un'etichetta da un codice operativo, nel caso in cui entrambi occupino, da soli, una linea del codice assemblativo. Usando i "due punti" si elimina questa potenziale ambiguità.

Una caratteristica infelice di alcuni assemblatori limita a sei o a otto il numero di caratteri di un'etichetta, mentre la maggior parte dei linguaggi ad alto livello permette di utilizzare nomi di lunghezza arbitraria. Nomi lunghi e scelti in modo appropriato rendono i programmi molto più leggibili e facili da capire da chi non ne è l'autore.

Ogni macchina è dotata di registri che hanno bisogno di un nome. I nomi assegnati a registri dell'x86 sono EAX, EBX, ECX e così via.

Il campo Codice operativo contiene un'abbreviazione simbolica del codice operativo oppure un comando per l'assemblatore stesso. La scelta di un nome appropriato è solo una questione di gusti e spesso i progettisti di differenti linguaggi assemblativi compiono scelte diverse. I progettisti dell'assemblatore MASM hanno deciso di usare MOV sia per caricare un registro dalla memoria sia per memorizzare un registro, ma avrebbero potuto scegliere di utilizzare MOVE oppure LOAD e STORE.

I programmi in linguaggio assemblativo hanno spesso bisogno di riservare spazio per le variabili. I progettisti del linguaggio MASM hanno scelto di usare DD (Define Double), perché nell'8088 la parola era di 16 bit.

Nelle istruzioni del linguaggio assemblativo il campo Operandi viene utilizzato per specificare gli indirizzi e i registri utilizzati come operandi dall'istruzione macchina. Nel caso di un'istruzione per la somma di interi il campo operandi indica quali elementi devono essere sommati fra loro. Nel caso di un'istruzione di salto indica invece a quale indirizzo deve portare la diramazione. Gli operandi possono essere registri, costanti, locazioni di memoria, e così via.

Il campo Commenti è il luogo in cui i programmati possono inserire utili spiegazioni sul funzionamento del programma a beneficio di altri programmati che in seguito lo potrebbero utilizzare o modificare (o a beneficio del programmatore stesso alcuni anni dopo aver scritto il codice). Un programma in linguaggio assemblativo senza una tale documentazione è praticamente incomprensibile, spesso anche per il suo stesso autore. Il campo dei commenti è destinato esclusivamente all'uso da parte dell'uomo e non ha alcun effetto sul processo di assemblaggio né tantomeno sul programma che verrà generato.

### 7.1.4 Pseudoistruzioni

Un linguaggio assemblativo, oltre a specificare quali istruzioni macchina devono essere eseguite, può anche contenere dei comandi indirizzati all'assemblatore stesso, per richiedere, per esempio, di allocare una certa quantità di memoria oppure per passare a una nuova pagina del listato. I comandi diretti all'assemblatore sono chiamati **pseudoistruzioni** o anche **direttive dell'assemblatore**. Nella Figura 7.1 abbiamo già visto una pseudoistruzione molto comune: DD. Nella Figura 7.2 ne sono elencate altre, tratte dall'assemblatore Microsoft MASM per gli x86.

Pseudoistruzione	Significato
SEGMENT	Inizia un nuovo segmento (testo, dati e così via) con certi attributi
ENDS	Termina il segmento corrente
ALIGN	Controlla l'allineamento della successiva istruzione o dei successivi dati
EQU	Definisce un nuovo simbolo uguale a una data espressione
DB	Allocà spazio per memorizzare uno o più byte (inizializzati)
DW	Allocà spazio per memorizzare uno o più elementi di dati (inizializzati) a 16 bit (parola)
DD	Allocà spazio per memorizzare uno o più elementi di dati (inizializzati) a 32 bit (parola doppia)
DQ	Allocà spazio per memorizzare uno o più elementi di dati (inizializzati) a 64 bit (parola quadrupla)
PROC	Inizia una procedura
ENDP	Termina una procedura
MACRO	Inizia una definizione di macro
ENDM	Termina una definizione di macro
PUBLIC	Esporta un nome definito nel modulo
EXTERN	Importa un nome da un altro modulo
INCLUDE	Preleva e include un altro file
IF	Inizia un assemblaggio condizionale in base a una data espressione
ELSE	Inizia un assemblaggio condizionale se la condizione IF precedente è falsa
ENDIF	Termina un assemblaggio condizionale
COMMENT	Definisce un nuovo carattere di inizio commento
PAGE	Genera un'interruzione di pagina nel listato
END	Termina il programma assemblativo

Figura 7.2 Alcune pseudoistruzioni dell'assemblatore MASM.

La pseudoistruzione SEGMENT inizia un nuovo segmento, mentre ENDS lo termina. È consentito iniziare un segmento di testo, contenente codice, poi un segmento dati e tornare successivamente al segmento testo, e così via.

ALIGN forza l'indirizzo della linea successiva, contenente in genere dati, a essere un multiplo dell'argomento della pseudoistruzione. Per esempio se il segmento contiene già 61 byte di dati, l'indirizzo allocato subito dopo la pseudoistruzione ALIGN 4 sarà 64.

EQU assegna un nome simbolico a un'espressione. Per esempio, dopo la pseudoistruzione

BASE EQU 1000

il simbolo BASE può essere usato in qualsiasi punto al posto di 1000. L'espressione che segue EQU può comprendere alcuni simboli definiti dal programmatore oltre a operatori aritmetici e di altro tipo, come mostra il seguente esempio

LIMIT EQU 4 \* BASE + 2000

La maggior parte degli assemblatori, compreso MASM, richiede che un simbolo sia definito prima di essere utilizzato in un'espressione come quella appena vista.

Le successive quattro pseudoistruzioni, DB, DW, DD e DQ, allocano memoria rispettivamente per una o più variabili della dimensione di 1, 2, 4 oppure 8 byte. Per esempio,

TABLE DB 11, 23, 49

alloca spazio per 3 byte e li inizializza rispettivamente ai valori 11, 23 e 49. Essa definisce inoltre il simbolo TABLE e lo associa all'indirizzo in cui è memorizzato il valore 11.

Le pseudoistruzioni PROC e ENDP definiscono rispettivamente l'inizio e la fine delle procedure del linguaggio assemblativo, che hanno la stessa funzione delle procedure definite in altri linguaggi di programmazione. Analogamente le direttive MACRO e ENDM delimitano l'area in cui compare la definizione di una macroistruzione (argomento trattato più avanti nel corso del capitolo).

Le due successive pseudoistruzioni, PUBLIC e EXTERN, controllano la visibilità dei simboli. È una pratica comune scrivere i programmi in più file e spesso capita che una procedura contenuta in un file richiami una procedura o acceda ai dati definiti in un altro file. La pseudoistruzione PUBLIC permette di esportare un simbolo che deve essere reso disponibile anche ad altri file, rendendo così possibili i riferimenti tra file diversi. Analogamente, per impedire che l'assemblatore segnali un errore quando incontra un simbolo che non è ancora stato definito, si può dichiarare un simbolo come EXTERN. Questa pseudoistruzione indica all'assemblatore che la definizione del simbolo compare in un altro file. I simboli che non sono dichiarati mediante alcuna delle due pseudoistruzioni precedenti sono visibili soltanto localmente, all'interno dello stesso file. Ciò significa che l'utilizzo, per esempio, di FOO in più file non genererà alcun conflitto, dato che ciascuna definizione è locale al file in cui si trova.

La direttiva INCLUDE fa sì che l'assemblatore prelevi un altro file e lo includa interamente all'interno di quello corrente. Spesso tali file comprendono definizioni, macroistruzioni e altri elementi che sono richiesti in più file.

Molti assemblatori, compreso MASM, supportano l'assemblaggio condizionale. Per esempio,

```

WORDSIZE EQU 32
IF WORDSIZE GT 32
WSIZE: DD 64
ELSE
WSIZE: DD 32
ENDIF

```

alloca una singola parola a 32 bit e chiama *WSIZE* il suo indirizzo. La parola è inizializzata a 64 oppure a 32, in base al valore di *WORDSIZE* (in questo caso 32). In genere si utilizza questo metodo per allocare una variabile quando occorre scrivere un programma che possa essere eseguito da macchine a 32 bit o a 64 bit. Inserendo all'interno di *IF* e *ENDIF* tutte le porzioni del codice dipendenti dalla macchina, e cambiando successivamente una singola definizione, *WORDSIZE*, il programma può essere automaticamente assemblato per entrambe le dimensioni di parola. Grazie a questo approccio è possibile mantenere un solo programma sorgente per più macchine destinazione (di tipo diverso), rendendo di conseguenza più facile lo sviluppo e il mantenimento del software. In molti casi tutte le definizioni dipendenti dalla macchina, come *WORDSIZE*, sono raggruppate in un singolo file, di cui vengono create versioni diverse per ogni tipo di macchina. In questo modo, includendo semplicemente il corretto file delle definizioni, è possibile utilizzare facilmente il programma per macchine diverse.

La pseudoistruzione *COMMENT* permette all'utente di modificare il delimitatore dei commenti, che inizialmente è impostato al carattere “punto e virgola”. *PAGE* è utilizzato per controllare il listato che l'assemblatore può generare su richiesta. Infine *END* segna la fine del programma. Oltre a quelle appena elencate, in MASM esistono molte altre pseudoistruzioni. Altri assemblatori per x86 hanno invece un insieme diverso di pseudoistruzioni; infatti queste non sono dettate dall'architettura della macchina, ma dai gusti di chi ha scritto l'assemblatore.

## 7.2 Macroistruzioni

Spesso i programmatore di linguaggi assemblativi hanno bisogno di ripetere più volte all'interno di un programma alcune sequenze d'istruzioni. Il modo più ovvio per farlo consiste nel riscrivere ogni volta che sia necessario. Tuttavia se una sequenza è lunga, o se deve essere utilizzata molte volte, scriverla ripetutamente è un'operazione tediosa.

Un approccio alternativo consiste nel trasformare la sequenza in una procedura e nel richiamarla quando è richiesta. Questa strategia ha lo svantaggio di dover eseguire, ogni volta che occorre utilizzare la sequenza, un'istruzione di chiamata di procedura e una per restituirla il controllo. Se le sequenze sono corte, per esempio di due istruzioni, ma sono usate frequentemente, il lavoro aggiuntivo determinato dalla chiamata alla procedura può rallentare notevolmente il programma. Le *macro*, abbreviazione usuale di macroistruzioni, forniscono una soluzione facile ed efficiente al problema di ripetere un'identica, o quasi, sequenza d'istruzioni.

### 7.2.1 Definizione, chiamata ed espansione di macro

Una definizione di macro è un modo per assegnare un nome a una porzione di testo. Dopo che una macro è stata definita, il programmatore può scriverne il nome al posto della porzione di programma corrispondente. Una macro in realtà non è altro che un'abbreviazione per una porzione di testo. La Figura 7.3(a) mostra un programma in linguaggio assemblativo per x86 che scambia due volte il contenuto delle variabili *p* e *q*. Queste sequenze possono essere definite come macro, come mostra la Figura 7.3(b). Dopo aver definito la macro, ogni occorrenza di *SWAP* viene sostituita dalle quattro linee:

```

MOV EAX,P
MOV EBX,Q
MOV Q,EAX
MOV P,EBX

```

Il programmatore ha definito *SWAP* come un'abbreviazione delle quattro istruzioni.

<pre> MOV EAX,P MOV EBX,Q MOV Q,EAX MOV P,EBX </pre>	<pre> SWAP MACRO MOV EAX,P MOV EBX,Q MOV Q,EAX MOV P,EBX ENDM </pre>
(a)	(b)

Figura 7.3 Codice in linguaggio assemblativo per scambiare due volte *P* e *Q*. Senza (a) e con (b) macro.

Anche se assemblatori diversi utilizzano notazioni leggermente differenti, in tutti i casi una definizione di macro è composta dalle seguenti parti principali:

1. un'intestazione che indica il nome della macro che si sta definendo;
2. il testo che costituisce il corpo della macro;
3. una pseudoistruzione che segna la fine della definizione (per esempio, *ENDM*).

Quando l'assemblatore incontra una definizione di macro, la salva nella tabella delle definizioni di macro per poterla utilizzare successivamente. Da quel momento in poi ogni volta che il nome della macro appare come codice operativo, l'assemblatore la sostituisce con il corpo della macro. L'uso del nome di una macro come codice operati-

vo viene detta **chiamata di macro**, mentre la sua sostituzione con il corpo della macro è chiamata **espansione di macro**.

L'espansione della macro viene effettuata durante il processo assemblativo e non durante l'esecuzione del programma. Questo è un punto importante. Il programma della Figura 7.3(a) e quello della Figura 7.3(b) produrranno esattamente lo stesso codice in linguaggio macchina. Guardando solamente il programma tradotto in linguaggio macchina, è impossibile stabilire se, durante la generazione, siano state utilizzate o meno delle macro. Il motivo è che, una volta completata l'espansione delle macro, l'assemblatore elimina le loro definizioni, di cui non rimane alcuna traccia nel programma generato.

Le chiamate di macro non devono essere confuse con le chiamate di procedure. La differenza principale è che una chiamata di macro è un'istruzione diretta all'assemblatore, affinché sostituisca il nome della macro con il suo corpo. Una chiamata di procedura è invece un'istruzione macchina inserita nel programma oggetto e che sarà eseguita in un secondo momento per chiamare la procedura. La Figura 7.4 confronta le chiamate di macro con le chiamate di procedura.

Domanda	Macro	Procedura
Quando viene effettuata la chiamata?	Durante l'assemblaggio	Durante l'esecuzione
Il corpo è inserito nel programma oggetto in ogni punto in cui avviene una chiamata?	Sì	No
L'istruzione per la chiamata di procedura è inserita nel programma oggetto ed eseguita successivamente?	No	Sì
Occorre usare un'istruzione di ritorno dopo aver effettuato la chiamata?	No	Sì
Quante copie del corpo appaiono nel programma oggetto?	Una per chiamata di macro	Una

Figura 7.4 Confronto tra le chiamate di macro e le chiamate di procedura

Concettualmente conviene pensare che il processo assemblativo avvenga in due fasi. Nella prima vengono salvate tutte le definizioni delle macro e vengono espanso le loro chiamate. Nella seconda il testo risultante viene elaborato come se fosse il programma originale. Vedendolo in questo modo, il programma sorgente viene letto e poi trasformato in un altro programma nel quale sono state rimosse tutte le definizioni delle macro e le chiamate sono state sostituite dai loro corpi. L'output risultante è un programma in linguaggio assemblativo che non contiene alcuna macro e che può essere passato all'assemblatore.

È importante tenere a mente che un programma è semplicemente una stringa di caratteri che comprende lettere, cifre, spazi, segni di punteggiatura e "ritorni a capo" (passaggio a una nuova linea). L'espansione di una macro consiste nel sostituire una parte di questa stringa con una nuova stringa di caratteri. L'uso delle macro è una tecnica per manipolare stringhe di caratteri, senza preoccuparsi del loro significato.

### 7.2.2 Macro con parametri

Le macro descritte precedentemente sono utilizzabili in sorgenti di breve lunghezza in cui si ripete più volte una sequenza d'istruzioni perfettamente identica. Tuttavia è frequente il caso in cui un programma contiene varie sequenze d'istruzioni che non sono del tutto identiche. Nell'esempio della Figura 7.5(a) la prima sequenza scambia *P* con *Q*, mentre la seconda *R* con *S*.

<pre> MOV EAX,P MOV EBX,Q MOV Q,EAX MOV P,EBX   MOV EAX,R MOV EBX,S MOV S,EAX MOV R,EBX </pre>	(a)	<pre> CHANGE MACRO P1, P2 MOV EAX,P1 MOV EBX,P2 MOV P2,EAX MOV P1,EBX ENDM   CHANGE P, Q   CHANGE R, S </pre>	(b)
--	-----	---	-----

Figura 7.5 Sequenze d'istruzioni quasi identiche. Senza (a) e con (b) macro.

Gli assemblatori di macro gestiscono queste situazioni consentendo che le definizioni delle macro specifichino dei **parametri formali** e che le chiamate forniscano dei **parametri attuali**. Quando una macro viene espansa i parametri formali che appaiono nel suo corpo vengono sostituiti dai corrispondenti parametri attuali. I parametri attuali sono specificati nel campo operandi della chiamata di macro. La Figura 7.5(b) mostra il programma della Figura 7.5(a) riscritto utilizzando una macro con due parametri. *P1* e *P2* sono i parametri formali. Nel momento in cui la macro viene espansa, ogni occorrenza di *P1* nel corpo della macro viene sostituita dal secondo parametro attuale. Nella chiamata di macro

CHANGE P, Q

*P* è il primo parametro attuale e *Q* è il secondo. I programmi prodotti dalle due parti della Figura 7.5 sono quindi identici: contengono esattamente le stesse istruzioni con gli stessi operandi.

### 7.2.3 Caratteristiche avanzate

La maggior parte dei processori di macro mette a disposizione un gran numero di caratteristiche avanzate per rendere più facile la vita dei programmatori in linguaggio assemblativo. In questo paragrafo daremo un'occhiata ad alcune caratteristiche avanzate di

MASM. Un problema che sorge usando gli assemblatori che supportano le macro è la duplicazione delle etichette. Supponiamo che una macro contenga un'istruzione di diramazione condizionale e un'etichetta che rappresenta la destinazione del salto. Se la macro viene chiamata due o più volte, l'etichetta verrà duplicata causando un errore del linguaggio assemblativo. Una soluzione consiste nell'obbligare il programmatore a fornire tramite parametro una diversa etichetta ogni volta che chiama la macro. La soluzione utilizzata da MASM è invece quella di permettere la dichiarazione di etichetta LOCAL, e lasciare che l'assemblatore generi un'etichetta diversa per ogni espansione della macro. Altri assemblatori utilizzano la regola secondo la quale le etichette numeriche sono automaticamente definite come locali.

MASM e la maggior parte degli assemblatori permettono di definire macro all'interno di altre macro. Questa funzionalità risulta particolarmente utile se usata insieme all'assemblaggio condizionale. In genere la stessa macro viene definita in entrambi i rami di un costrutto IF, come il seguente:

```

M1 MACRO
  IF WORDSIZE GT 16
M2  MACRO
...
ENDM
ELSE
M2  MACRO
...
ENDM
ENDIF
ENDM

```

In entrambi i casi la macro *M2* viene definita, ma la sua definizione dipende dal fatto che il programma venga assemblato su una macchina a 16 bit oppure su una a 32 bit. Se *M1* non viene chiamata, *M2* non viene definita.

Infine, le macro possono chiamare altre macro, comprese loro stesse. Se una macro è ricorsiva, cioè se richiama se stessa, deve passare un parametro modificato a ogni espansione e che fa terminare la ricorsione quando raggiunge un determinato valore. Se così non fosse, l'assemblatore potrebbe entrare in un ciclo infinito, e in tal caso l'utente dovrebbe interrompere esplicitamente l'assemblatore.

#### 7.2.4 Implementazione delle funzionalità macro negli assemblatori

Per implementare una funzionalità macro un assemblatore deve essere in grado di eseguire due funzioni: salvare le definizioni delle macro ed espandere le loro chiamate. Esaminiamo una alla volta queste due operazioni.

L'assemblatore deve mantenere una tabella con tutti i nomi delle macro e, associato a ogni nome, un puntatore alla definizione precedentemente memorizzata, in modo che sia possibile recuperarla quando è necessario. Alcuni assemblatori hanno una tabella separata per i nomi delle macro, mentre altri sono dotati di un'unica tabella in cui sono mantenute, oltre alle macro, anche tutte le istruzioni macchina e le pseudoistruzioni.

Quando s'incontra una definizione di macro si aggiunge nella tabella un elemento che indica il nome della macro, il numero di parametri formali e un puntatore a un'altra tabella, la tabella delle definizioni delle macro, che conterrà il suo corpo. In questa fase si costruisce anche una lista dei parametri formali da utilizzare nell'elaborazione della definizione. Viene quindi letto il corpo della macro e memorizzato all'interno della tabella delle definizioni. I parametri formali che s'incontrano all'interno del corpo sono indicati mediante simboli speciali. Come esempio mostriamo la rappresentazione interna della definizione della macro *CHANGE*, utilizzando il punto e virgola come "ritorno a capo" e la "e commerciale" (&) come simbolo dei parametri formali:

```
MOV EAX,&P1; MOV EBX,&P2; MOV &P2,EAX; MOV &P1,EBX;
```

Nella tabella delle definizioni delle macro il corpo di ciascuna macro è semplicemente una stringa di caratteri.

Durante il primo passo del processo di assemblaggio, vengono cercati i codici operativi e le macro vengono espanso. Ogni volta che s'incontra una definizione di macro la si memorizza nella tabella delle macro. Quando viene chiamata una macro l'assemblatore interrompe temporaneamente la lettura dal dispositivo di input e inizia a leggere il corpo della macro precedentemente memorizzato. I parametri formali presenti al suo interno vengono poi sostituiti dai parametri attuali specificati dalla chiamata. La presenza del simbolo & prima dei parametri formali permette all'assemblatore di riconoscerli più facilmente.

### 7.3 Processo di assemblaggio

Nei paragrafi successivi descriveremo brevemente come funziona un assemblatore. Anche se ogni macchina ha il proprio linguaggio assemblativo, i diversi processi di assemblaggio sono tra loro sufficientemente simili da poterci permettere una descrizione generale.

#### 7.3.1 Assemblatori a due passate

Dato che ogni programma in linguaggio assemblativo consiste in una serie d'istruzioni scritte su un'unica riga, di primo acchito potrebbe sembrare naturale avere un assemblatore che legge un'istruzione, la traduce in linguaggio macchina e scrive su file il codice generato, oltre a produrre, se richiesto, un altro file contenente il listato. Questo procedimento andrebbe ripetuto fino alla traduzione dell'intero programma. Purtroppo però questa strategia non funziona.

Consideriamo il caso in cui la prima istruzione sia una diramazione verso *L*. L'assemblatore non può assemblare questa istruzione finché non venga a conoscenza dell'indirizzo dell'istruzione *L*. Dato che questa istruzione potrebbe trovarsi verso la fine del programma l'assemblatore è obbligato a leggere praticamente tutto il programma per trovare il suo indirizzo. Il fatto di utilizzare un simbolo, *L*, prima ancora che sia stato definito, costituisce il problema dei riferimenti in avanti; in altre parole si effettua un riferimento a un simbolo la cui definizione ha luogo in un punto successivo del programma.

È possibile gestire i riferimenti in avanti in due modi. Primo, l'assemblatore può leggere il programma sorgente due volte. Ciascuna lettura del programma sorgente è chiamata **passata** e i traduttori che leggono due volte il programma di input sono detti a **due passate**. Durante la prima lettura questi assemblatori raccolgono in una tabella tutte le definizioni dei simboli, comprese le etichette delle istruzioni. Prima di iniziare la seconda passata si conoscono quindi i valori di tutti i simboli. In tal modo non rimane alcun riferimento in avanti ed è possibile leggere ogni istruzione, assemblarla e generare il codice corrispondente. Questo approccio richiede una passata aggiuntiva per elaborare il programma di input, ma è concettualmente semplice.

Un secondo approccio consiste nel leggere il file in ingresso una sola volta, convertirlo in un formato intermedio e memorizzare in una tabella della memoria questa forma intermedia. In seguito viene effettuata una seconda passata sulla tabella, invece che sul codice sorgente come avveniva nel primo approccio. Se è disponibile una sufficiente quantità di memoria (fisica o virtuale) questo approccio risparmia il tempo dovuto alle operazioni di I/O. Se non viene richiesta la generazione del listato, è possibile ridurre al minimo indispensabile questa forma intermedia.

In entrambi gli approcci, la prima passata ha anche il compito di salvare tutte le definizioni di macro e di espanderle quando si incontrano le loro chiamate. Generalmente quindi la definizione dei simboli e l'espansione delle macro avvengono in un'unica passata.

### 7.3.2 Prima passata

La principale funzione della prima passata è quella di costruire la cosiddetta **tabella dei simboli**, contenente i valori di tutti i simboli. Un simbolo è un'etichetta, oppure un valore, al quale è stato assegnato un nome simbolico attraverso una pseudoistruzione, come

```
BUFSIZE EQU 8192
```

Per assegnare nel campo etichetta di un'istruzione un valore a un simbolo, l'assemblatore deve conoscere quale indirizzo avrà tale istruzione durante l'esecuzione del programma. Durante il processo di assemblaggio l'assemblatore mantiene una variabile, chiamata **ILC** (*Instruction Location Counter*, "contatore di locazioni delle istruzioni"), per tener traccia dell'indirizzo che l'istruzione che sta assemblando avrà a tempo di esecuzione. All'inizio della prima passata la variabile ha valore 0 e, come mostra la Figura 7.6, ogni volta che viene elaborata un'istruzione la variabile viene incrementata della sua lunghezza. L'esempio mostrato nella figura è relativo a x86.

Nella maggior parte degli assemblatori la prima passata utilizza tre tabelle interne per i simboli, le pseudoistruzioni e i codici operativi. Se necessario viene mantenuta anche la tabella dei letterali (si veda in seguito la definizione). Come mostra la Figura 7.7, la tabella dei simboli ha una linea per ciascun identificatore (simbolo). I simboli vengono definiti quando sono utilizzati come etichette oppure mediante una definizione esplicita (per esempio, EQU). Ogni elemento della tabella dei simboli contiene il simbolo stesso (o un puntatore a esso), il suo valore numerico e, in alcuni casi, altre informazioni.

Etichetta	Codice operativo	Operandi	Commenti	Lunghezza	ILC
MARIA:	MOV	EAX, I	EAX = I	5	100
	MOV	EBX, J	EBX = J	6	105
ROBERTA:	MOV	ECX, K	ECX = K	6	111
	IMUL	EAX, EAX	EAX = I x I	2	117
	IMUL	EBX, EBX	EBX = J x J	3	119
	IMUL	ECX, ECX	ECX = K x K	3	122
MARIANNA:	ADD	EAX, EBX	EAX = I x I + J x J	2	125
	ADD	EAX, ECX	EAX = I x I + J x J + K x K	2	127
STEFANIA:	JMP	DONE	salto a DONE	5	129

Figura 7.6 Il contatore di locazioni delle istruzioni (ILC) tiene traccia degli indirizzi delle locazioni di memoria in cui le istruzioni saranno caricate. In questo esempio le istruzioni prima di MARIA occupano 100 byte.

Identificatore	Valore	Altre informazioni
MARIA	100	
ROBERTA	111	
MARIANNA	125	
STEFANIA	129	

Figura 7.7 Tabella dei simboli per il programma della Figura 7.6.

Queste informazioni aggiuntive possono comprendere:

1. la lunghezza del campo dati associato al simbolo;
2. i bit di rilocazione (il simbolo cambia valore se il programma è caricato in un indirizzo diverso rispetto a quello che ha assunto l'assemblatore);
3. se il simbolo sia o meno accessibile dall'esterno della procedura.

La tabella dei codici operativi (la Figura 7.8 ne mostra una parte) contiene un elemento per ciascun codice simbolico (mnemonico) del linguaggio. Ciascun elemento contiene il codice simbolico, due operandi, il valore numerico del codice operativo, la lunghezza dell'istruzione e un numero di tipo (che permette di raggruppare i codici operativi a seconda del numero e tipo di operandi).

Se per esempio un'istruzione ADD contiene come primo operando EAX e come secondo una costante a 32 bit (immed32), si utilizza il codice operativo 0x05 e la lunghezza dell'istruzione è di 5 byte. In realtà le costanti che possono essere espresse con 8 o 16 bit utilizzano codici operativi diversi, ma non sono mostrati nella figura. Se ADD viene utilizzata con due registri come operandi, allora la sua lunghezza risulta di 2 byte e il suo codice operativo è 0x01. La classe dell'istruzione 19 (il valore è arbitrario) dovrebbe essere assegnata a tutte le combinazioni codice operativo-operando che seguono le stesse regole. Le istruzioni facenti parte di questa classe dovrebbero quindi essere elab-

borate allo stesso modo dell'istruzione ADD che usa due registri come operandi. La classe dell'istruzione designa in realtà una procedura dell'assemblatore, richiamata per elaborare tutte le istruzioni di un certo tipo.

Codice operativo	Primo operando	Secondo operando	Codice esadecimale	Lunghezza dell'istruzione	Classe dell'istruzione
AAA	—	—	37	1	6
ADD	EAX	immed32	05	5	4
ADD	reg	reg	01	2	19
AND	EAX	immed32	25	5	4
AND	reg	reg	21	2	19

Figura 7.8 Parte della tabella dei codici operativi per un assemblatore x86.

Alcuni assemblatori permettono ai programmatore di scrivere istruzioni che usano l'indirizzamento immediato anche se nel linguaggio destinazione non esiste alcuna istruzione corrispondente. Per gestire queste istruzioni, chiamiamole *pseudoimmediate*, l'assemblatore alloca alla fine del programma la memoria per l'operando immediato e genera un'istruzione che fa riferimento a esso. Per esempio il mainframe IBM 360 e i suoi successori non hanno istruzioni immediate, ma i programmatore possono comunque scrivere

L 14,=F'5'

per caricare il registro 14 con una parola costante il cui valore è 5. In questo modo, il programmatore non è obbligato a scrivere esplicitamente una pseudoistruzione per allocare una parola inizializzata a 5, indicando la sua etichetta e poi utilizzandola nell'istruzione L. Le costanti per le quali l'assemblatore riserva automaticamente la memoria sono chiamate **letterali**. I letterali, oltre a risparmiare un po' di scrittura di codice al programmatore, migliorano anche la leggibilità del programma rendendo visibile il valore delle costanti all'interno delle istruzioni. La prima passata dell'assemblatore dovrebbe costruire una tabella di tutti i letterali utilizzati nel programma. Le nostre tre macchine di esempio sono dotate d'istruzioni immediate e quindi i loro assemblatori non utilizzano i letterali. Oggi giorno è abbastanza comune che un assemblatore sia dotato d'istruzioni letterali, ma una volta non era così. È probabile che l'uso molto diffuso dei letterali abbia reso evidente agli occhi dei progettisti delle macchine che l'indirizzamento immediato sia una buona idea. Se però è necessario utilizzare i letterali, allora durante l'assemblaggio occorre mantenere una tabella in cui viene aggiunto un elemento ogni volta che si incontra un letterale. Dopo la prima passata questa tabella viene ordinata e gli elementi duplicati vengono rimossi.

La Figura 7.9 mostra una procedura che potrebbe servire come base per la prima passata di un assemblatore. Lo stile di programmazione è autoespli- cativo: i nomi delle procedure sono stati scelti per fornire una buona indicazione della loro funzione. L'aspetto più importante è che la Figura 7.9 rappresenta abbastanza bene la struttura della

prima passata. La procedura è sintetica, può essere compresa facilmente, e rende evidente quale deve essere il passo successivo, ovvero la scrittura del corpo delle procedure richiamate al suo interno.

```

public static void pass_one( ) {
    // Questa procedura è uno schema della prima passata di un semplice assemblatore.
    boolean more_input = true; // indicatore che interrompe la prima passata
    String line, symbol, literal, opcode; // campi dell'istruzione
    int location_counter, length, value, type; // variabili generiche
    final int END_STATEMENT = -2; // la fine dell'input

    location_counter = 0; // assembila la prima istruzione in 0
    initialize_table( ); // inizializzazione generale

    while(more_input) { // more_input viene impostato a falso da END
        line = read_next_line( ); // preleva una linea di input
        length = 0; // # byte nell'istruzione
        type = 0; // di quale tipo (formato) è l'istruzione

        if (line_is_not_comment(line)) { // alla linea è associata un'etichetta?
            symbol = check_for_symbol(line); // se sì, si registra il simbolo e il valore
            if (symbol != null)
                enter_new_symbol(symbol, location_counter);
            literal = check_for_literal(line); // la linea contiene un letterale?
            if (literal != null) // se sì, lo si inserisce nella tabella
                enter_new_literal(literal);

            // Si determina ora il tipo del codice operativo. _1 indica un codice operativo inesistente
            opcode = extract_opcode(line); // localizza il nome mnemonico del codice operativo
            type = search_opcode_table(opcode); // trova il formato, p.es. OP REG1,REG2
            if (type < 0) // se non è un codice operativo, è una pseudoistruzione?
                type = search_pseudo_table(opcode);
            switch(type) { // determina la lunghezza di questa istruzione
                case 1: length = get_length_of_type1(line); break;
                case 2: length = get_length_of_type2(line); break;
                // altri casi
            }
        }

        write_temp_file(opcode, length, line); // informazioni utili per la seconda passata
        location_counter = location_counter + length; // aggiorna loc_ctr
        if (type == END_STATEMENT) { // l'input è terminato?
            more_input = false; // se sì, si eseguono delle operazioni finali
            rewind_temp_for_pass_two( ); // come riportare al punto di inizio il file temporaneo
            sort_literal_table( ); // e ordinare la tabella dei letterali
            remove_redundant_literals( ); // e rimuovere da questa i duplicati
        }
    }
}

```

Figura 7.9 Prima passata di un semplice assemblatore.

Alcune di queste procedure saranno relativamente corte, come *check\_for\_symbol* che restituisce una stringa di caratteri rappresentante un simbolo, se è presente, oppure null

in caso contrario. Altre procedure, come *get\_length\_of\_type1* e *get\_length\_of\_type2* potrebbero invece essere più lunghe e richiamare altre procedure. In generale il numero di tipi non sarà ovviamente limitato a due, ma dipenderà dal linguaggio che si sta assemblando e da quanti tipi d'istruzioni è composto.

Oltre a facilitarne la stesura, strutturare i programmi in questo modo presenta altri vantaggi. Se più persone lavorano alla scrittura dell'assemblatore le varie procedure possono essere distribuite fra tutti i programmatore. Inoltre tutti i dettagli (più difficili) riguardanti la lettura dell'input rimangono nascosti nella procedura *read\_next\_line*. Se dovessero cambiare, per esempio a causa di una modifica del sistema operativo, sarebbe necessario modificare solamente una delle procedure ausiliarie, mentre non occorrerebbe effettuare alcuna modifica alla procedura *pass\_one*.

Durante la prima passata viene analizzata ogni linea per trovare il codice operativo, determinare il suo tipo e calcolare la lunghezza dell'istruzione. Queste informazioni sono necessarie anche per la seconda passata e quindi può essere utile trascriverle esplicitamente in modo da evitare, nella passata successiva, di analizzare nuovamente la linea da zero. Tuttavia, riscrivere il file di input genera un maggior numero di operazioni di I/O. La scelta tra effettuare più operazioni di I/O per eliminare una seconda analisi delle istruzioni oppure effettuare meno operazioni di I/O, ma più analisi delle linee di codice, è condizionata da vari fattori, tra cui la velocità della CPU rispetto a quella del disco e l'efficienza del file system. In questo esempio scriveremo un file temporaneo contenente il tipo, il codice operativo, la lunghezza e l'effettiva linea di input. La seconda passata legge questa linea al posto di quella presente nel file di input.

La prima passata termina quando viene letta la pseudoistruzione END. Se necessario, a questo punto possono essere ordinate le tabelle dei simboli e dei letterali. Quest'ultimi, dopo essere stati ordinati, possono essere controllati al fine di individuare, ed eventualmente rimuovere, gli elementi doppi.

### 7.3.3 Seconda passata

La funzione della seconda passata è la generazione del programma oggetto e l'eventuale stampa del listato. Oltre a ciò la seconda passata deve generare delle informazioni richieste dal linker per collegare in un unico file eseguibile le procedure assemblate in momenti distinti. La Figura 7.10 mostra una bozza della procedura che implementa la seconda passata.

Le operazioni compiute dalla seconda passata sono più o meno simili a quelle della prima: le linee vengono lette ed elaborate una alla volta. Dato che all'inizio di ogni linea abbiamo scritto (sul file temporaneo) il tipo, il codice operativo e la lunghezza, queste informazioni vengono lette in modo da risparmiare parte della fase di analisi dell'input. Il lavoro principale della generazione del codice è realizzato dalle procedure *eval\_type1*, *eval\_type2*, e così via. Ciascuna di queste funzioni gestisce un particolare schema, per esempio un codice operativo e due registri come operandi, genera il codice binario corrispondente all'istruzione e infine lo restituisce all'interno della variabile *code*. Questo codice viene poi scritto su file. Con ogni probabilità la funzione *write\_output* accumula in memoria il codice binario generato e lo scrive su file in grosse porzioni per ridurre il traffico dati verso il disco.

```

public static void pass_two() {
    // Questa procedura è uno schema della seconda passata di un semplice assemblatore.
    boolean more_input = true;           // indicatore che interrompe la seconda passata
    String line, opcode;                // campi dell'istruzione
    int location_counter, length, type; // variabili varie
    final int END_STATEMENT = -2;       // segnala la fine dell'input
    final int MAX_CODE = 16;             // numero massimo di byte di codice per istruzione
    byte code[] = new byte[MAX_CODE];   // contiene il codice generato da un'istruzione

    location_counter = 0;               // assembla la prima istruzione in 0

    while (more_input) {                // more_input viene impostato a falso da END
        type = read_type();            // preleva il campo tipo della linea successiva
        opcode = read_opcode();        // preleva il campo codice operativo della linea successiva
        length = read_length();        // preleva il campo della lunghezza della linea successiva
        line = read_line();            // preleva la linea attuale dell'input

        if (type != 0) {                // il tipo 0 è per le linee di commento
            switch(type) {              // genera il codice di output
                case 1: eval_type1(opcode, length, line, code); break;
                case 2: eval_type2(opcode, length, line, code); break;
                // altri casi
            }
        }
        write_output(code);           // scrive il codice binario
        write_listing(code, line);    // stampa una linea del listato
        location_counter = location_counter + length; // aggiorna loc_ctr
        if (type == END_STATEMENT) {    // l'input è terminato?
            more_input = false;        // se sì, si eseguono alcune operazioni finali
            finish_up();               // varie e conclusione
        }
    }
}

```

Figura 7.10 Seconda passata di un semplice assemblatore.

L'istruzione sorgente originale e il codice oggetto generato da questa (in formato esadecimale) possono quindi essere stampati oppure memorizzati in un buffer per rimandarne la stampa. Dopo aver modificato ILC la seconda passata preleva l'istruzione successiva.

Finora si è assunto che il programma sorgente non contenga alcun errore. Chiunque abbia mai scritto un programma, in qualsiasi linguaggio, sa bene quanto sia realistica una simile affermazione. Alcuni fra gli errori più comuni sono:

1. un simbolo è stato utilizzato senza essere stato definito;
2. un simbolo è stato definito più di una volta;
3. il nome nel campo del codice operativo non è un codice operativo lecito;
4. un codice operativo non è seguito da un numero sufficiente di operandi;
5. un codice operativo è seguito da un numero eccessivo di operandi;
6. un numero contiene un carattere non valido, per esempio 143G6;

7. un registro è stato utilizzato in modo scorretto (per esempio, una diramazione verso un registro);
8. manca la pseudoistruzione END.

I programmati sono piuttosto ingegnosi nel trovare sempre nuovi tipi di errori da commettere. Quelli dovuti ai simboli non definiti spesso sono originati da errori di battitura e quindi un assemblatore astuto potrebbe provare a sostituire il simbolo non definito con quello che, fra quelli che sono stati correttamente definiti, più gli assomiglia. Per la maggior parte degli altri errori però c'è ben poco da fare. Quando un assemblatore incontra un'istruzione errata si limita a stampare un messaggio di errore e cerca di continuare l'assemblaggio.

### 7.3.4 Tabella dei simboli

Durante la prima passata l'assemblatore accumula informazioni sui simboli e i loro valori. Tali informazioni devono essere memorizzate nella tabella dei simboli per essere utilizzate durante la seconda passata. Ora prenderemo brevemente in esame alcuni tra i vari modi di organizzare la tabella dei simboli. In tutti i casi si cerca di simulare una memoria associativa, che concettualmente non è altro che un insieme di coppie (simbolo, valore). Dato un simbolo, la memoria associativa deve fornire il valore corrispondente.

La tecnica più semplice consiste nell'implementare la tabella dei simboli come un vettore di coppie, in cui il primo elemento è il simbolo (oppure un puntatore a esso) e il secondo è il valore (oppure un puntatore a esso). Quando si vuole recuperare un simbolo, la routine della tabella dei simboli effettua semplicemente una ricerca lineare all'interno dell'array finché non trova l'elemento desiderato. Questo metodo è facile da programmare, ma allo stesso tempo è lento, dato che per ciascuna ricerca occorre esaminare, in media, metà della tabella.

Un altro modo per organizzare la tabella dei simboli è quello di ordinarla rispetto ai simboli e di utilizzare un algoritmo di ricerca dicotomica per cercare il simbolo desiderato. Questo algoritmo funziona confrontando il simbolo con l'elemento centrale della tabella. Se il simbolo precede alfabeticamente l'elemento centrale della tabella, occorrerà continuare la ricerca nella prima metà della tabella. Se invece il simbolo segue l'elemento centrale, occorrerà cercarlo nella seconda metà della tabella. Se l'elemento centrale della tabella è uguale al simbolo cercato, la procedura termina.

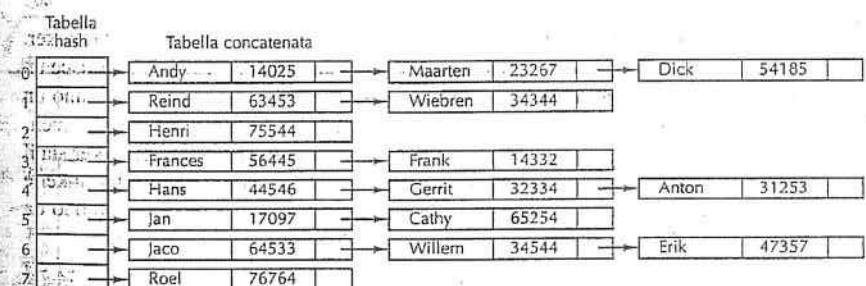
Assumendo che l'elemento centrale non sia uguale al simbolo cercato, possiamo comunque sapere in quale metà della tabella individuarlo. Si può quindi applicare nuovamente la ricerca dicotomica nella metà corretta della tabella; ciò permetterà di trovare il simbolo oppure di conoscere in quale quarto della tabella si trovi. Applicando ricorsivamente l'algoritmo è possibile completare una ricerca all'interno di una tabella di  $n$  elementi in circa  $\log_2 n$  tentativi. Questa tecnica è ovviamente molto più veloce rispetto alla ricerca lineare, ma richiede che sia conservato l'ordine degli elementi della tabella.

Un modo completamente differente per simulare una memoria associativa è la tecnica conosciuta con il nome di codifica hash o hashing. In questo approccio occorre definire una funzione "hash" che mappa i simboli nell'intervallo di interi compreso tra 0 e  $k - 1$ . Una possibile funzione consiste nel moltiplicare tra loro i codici ASCII dei

caratteri del simbolo, ignorando un eventuale overflow, e considerando come risultato il resto della divisione per  $k$  di questo valore oppure il risultato della divisione rispetto a un numero primo. In realtà può andar bene qualsiasi funzione dell'input che fornisca una distribuzione uniforme dei valori hash. I simboli possono essere memorizzati in una tabella composta da  $k$  secchielli (*bucket*) numerati da 0 a  $k - 1$ . Ogni posizione  $i$  della tabella hash punta a una lista concatenata in cui sono memorizzate le coppie (simbolo, valore) il cui valore hash del simbolo vale  $i$ . Con  $n$  simboli e una tabella hash con  $k$  posizioni, una lista avrà una lunghezza media pari a  $n/k$ . Se si sceglie  $k$  approssimativamente uguale a  $n$ , allora in media è possibile localizzare i simboli con un unico passo di ricerca. Modificando  $k$  è possibile ridurre la dimensione della tabella a spese però della velocità dell'operazione di ricerca. La codifica hash è illustrata nella Figura 7.11.

Andy	14025	0
Anton	31253	4
Cathy	65254	5
Dick	54185	0
Erik	47357	6
Frances	56445	3
Frank	14332	3
Gerrit	32334	4
Hans	44546	4
Henri	75544	2
Jan	17097	5
Jaco	64533	6
Maarten	23267	0
Reind	63453	1
Roel	76764	7
Willem	34544	6
Wiebren	34344	1

(a)



(b)

Figura 7.11 Codifica hash. (a) Simboli, valori e codici hash derivati dai simboli. (b) Tabella hash a otto elementi con liste concatenate di simboli e valori.

## 7.4 Collegamento e caricamento

La maggior parte dei programmi è composta da più di una procedura. Generalmente i compilatori e gli assemblatori traducono una procedura alla volta e memorizzano su disco il risultato della traduzione. Prima che il programma possa essere eseguito è necessario recuperare tutte le procedure e collegarle fra loro in modo appropriato. Inoltre, in assenza di memoria virtuale, occorre caricare in memoria centrale il programma ottenuto dal collegamento delle procedure. I programmi che eseguono questi passi sono chiamati in vari modi, tra cui *linker*, *linking loader* e *linkage editor*. La traduzione completa di un programma sorgente richiede due passi distinti, com'è mostrato nella Figura 7.12:

1. compilazione o assemblaggio dei file sorgente;
2. collegamento dei moduli oggetto.

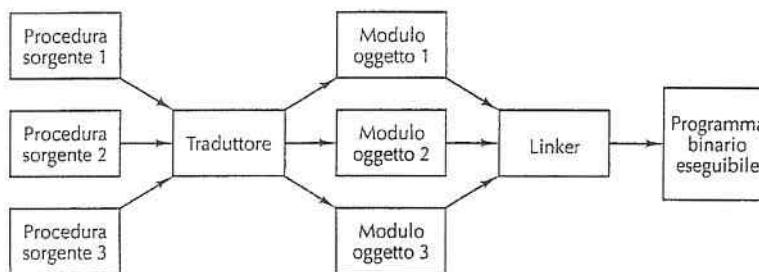


Figura 7.12 Generazione tramite un linker di un programma eseguibile binario a partire da un gruppo di file sorgente tradotti indipendentemente.

Il primo passo viene eseguito dal compilatore o dall'assemblatore, e il secondo dal linker. La traduzione che trasforma una procedura sorgente in un modulo oggetto costituisce un cambiamento di livello, dato che il linguaggio sorgente e quello destinazione hanno un diverso insieme d'istruzioni e usano una notazione differente. Il processo di collegamento invece non rappresenta un cambiamento di livello, dato che sia l'input del linker sia il suo output sono due programmi per la stessa macchina virtuale. La funzione del linker è quella di unire le procedure tradotte separatamente e di collegarle tra loro in modo da poterle eseguire come un'unica unità chiamata **programma eseguibile binario**.

Nei sistemi Windows i moduli oggetto sono caratterizzati dall'estensione *.obj* e i programmi eseguibili binari dall'estensione *.exe*. In UNIX i moduli oggetto hanno invece l'estensione *.o*, mentre i programmi eseguibili binari non hanno alcuna estensione.

Esiste un buon motivo per cui i compilatori e gli assemblatori traducono ciascun file sorgente come un'entità separata. Se un compilatore, o un assemblatore, dovesse leggere una serie di procedure sorgente e produrre direttamente un programma in linguaggio macchina pronto per essere eseguito, allora la modifica anche solo di un'istruzione in una delle procedure sorgente richiederebbe una nuova traduzione di tutte le altre procedure.

Se si usa la tecnica dei moduli oggetto separati (mostrata nella Figura 7.12) è necessario ritradurre soltanto la procedura modificata e non quelle rimaste invariate, anche se occorre ricollegare nuovamente tra loro i moduli oggetto. Di solito la fase di collegamento è molto più veloce della traduzione e permette quindi di risparmiare una grande quantità di tempo durante lo sviluppo di un programma. Questo guadagno risulta particolarmente importante quando si hanno centinaia o migliaia di moduli.

### 7.4.1 Compiti del linker

Nel processo assemblativo il contatore di locazioni delle istruzioni viene impostato a 0 all'inizio della prima passata. Questo passo corrisponde ad assumere che il modulo oggetto sarà collocato, durante l'esecuzione, all'indirizzo (virtuale) 0. La Figura 7.13 mostra quattro moduli oggetto per una macchina generica. In questo esempio ciascun modulo inizia con un'istruzione BRANCH che porta, all'interno del modulo, all'istruzione MOVE.

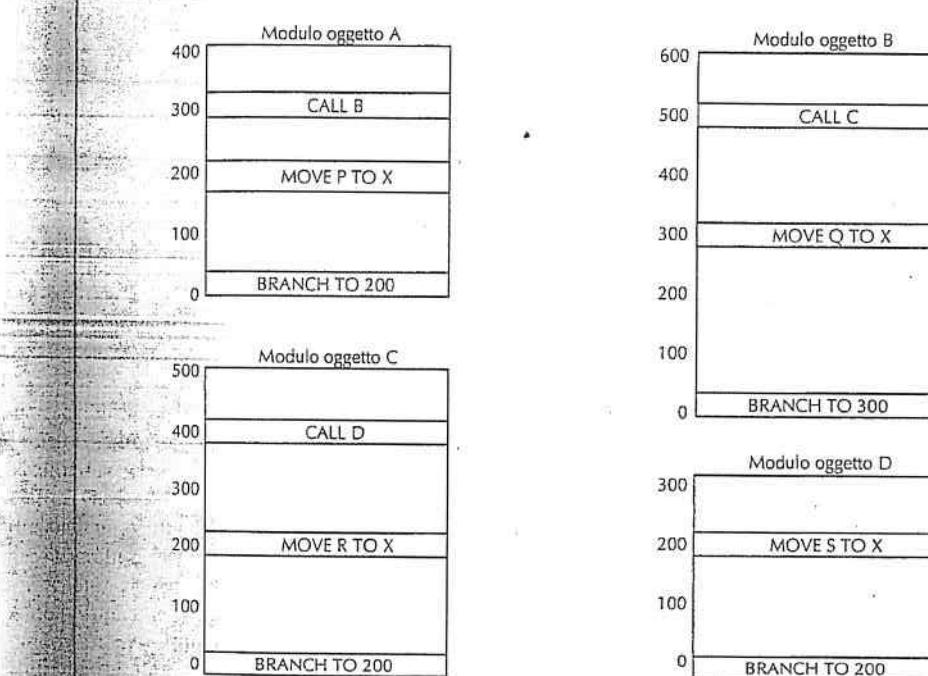


Figura 7.13 Ogni modulo ha il proprio spazio d'indirizzi che inizia da 0.

Per poter eseguire il programma il linker porta in memoria centrale i moduli oggetto al fine di formare l'immagine del programma eseguibile binario, come mostra la Figura 7.14(a).

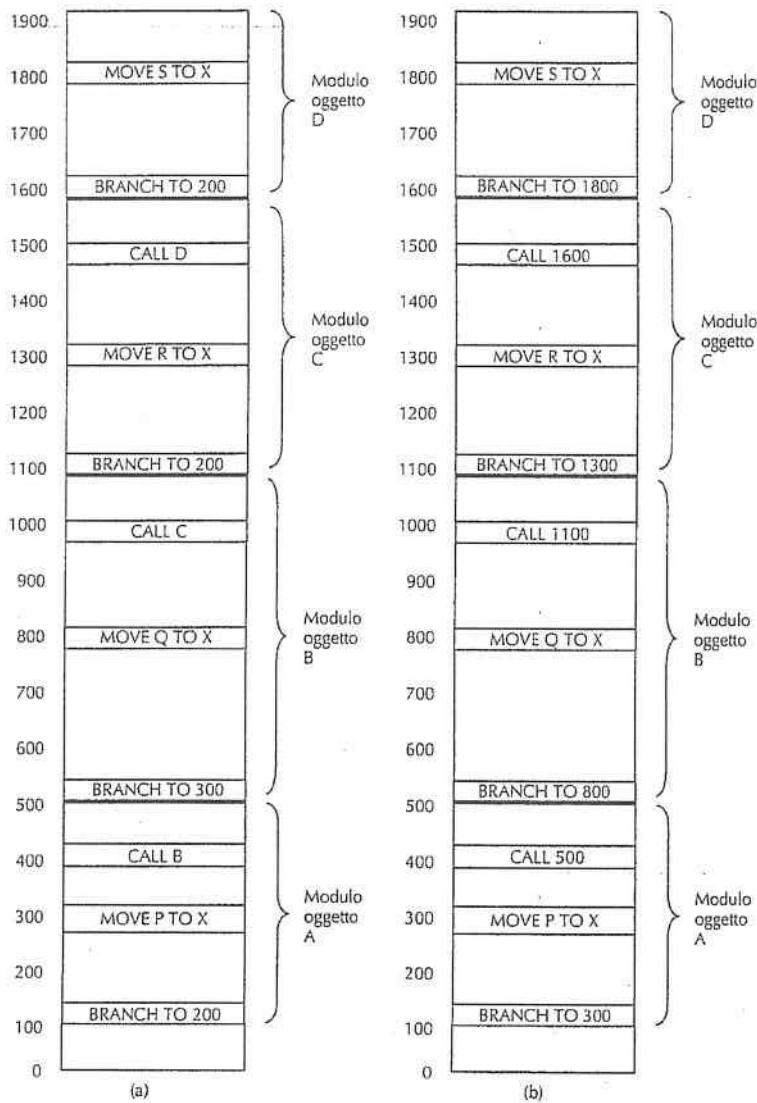


Figura 7.14 (a) I moduli oggetto della Figura 7.13 dopo essere stati posizionati nell'immagine binaria, ma prima di essere rilocati e collegati. (b) Gli stessi moduli oggetto dopo aver effettuato il collegamento e la rilocazione.

L'idea è di creare all'interno del linker un'immagine esatta dello spazio d'indirizzamento virtuale del programma eseguibile e di posizionare tutti i moduli oggetto nelle loca-

zioni corrette. Se non c'è sufficiente memoria per formare l'immagine è possibile utilizzare un file del disco. In genere una piccola sezione della memoria a partire dall'indirizzo zero viene utilizzata per i vettori di interrupt, la comunicazione con il sistema operativo, la cattura dei puntatori non inizializzati e altri scopi. Per questo motivo i programmi di solito iniziano a partire da una locazione maggiore di 0. Nella figura abbiamo (arbitrariamente) fatto iniziare i programmi all'indirizzo 100.

Il programma della Figura 7.14(a) non è ancora pronto per essere eseguito, anche se è caricato all'interno dell'immagine del file eseguibile binario. Consideriamo che cosa succederebbe se l'esecuzione cominciasse con l'istruzione che si trova all'inizio del modulo A. Il programma non effettuerrebbe correttamente il salto all'istruzione MOVE, dato che quella istruzione si trova ora all'indirizzo 300. In realtà tutte le istruzioni che fanno riferimento alla memoria fallirebbero per la stessa ragione. Occorre chiaramente trovare una soluzione al problema.

Il problema della rilocazione si verifica perché i moduli della Figura 7.13 hanno spazi degli indirizzi separati. Su una macchina con uno spazio degli indirizzi segmentato, come x86, ogni modulo potrebbe teoricamente avere il proprio spazio degli indirizzi all'interno del proprio segmento. In ogni caso l'unico sistema operativo per x86 che supporta questa possibilità è OS/2; tutte le versioni di Windows e di Unix supportano solamente uno spazio degli indirizzi lineare e quindi tutti i moduli oggetto devono essere uniti fra loro al suo interno. Inoltre neanche le istruzioni per la chiamata di procedure della Figura 7.14(a) funzionano correttamente. Nella figura si vede che all'indirizzo 400 il programmatore prova a chiamare il modulo oggetto B; l'assemblatore però non ha modo di sapere quale indirizzo inserire nell'istruzione CALL B, dato che ciascuna procedura è stata tradotta indipendentemente. L'indirizzo del modulo oggetto B non è infatti conosciuto fino al momento del collegamento. Questo è il problema dei riferimenti esterni. I due problemi possono essere risolti in modo semplice dal linker.

Il linker unisce gli spazi degli indirizzi separati dei diversi moduli oggetto all'interno di un unico spazio lineare degli indirizzi seguendo questi quattro passi:

1. costruisce una tabella di tutti i moduli oggetto e delle loro lunghezze;
2. in base a questa tabella assegna un indirizzo di partenza per ciascun modulo;
3. cerca tutte le istruzioni che fanno riferimento alla memoria e aggiunge a ciascuna di loro una costante di rilocazione uguale all'indirizzo di partenza del suo modulo;
4. cerca tutte le istruzioni che fanno riferimento ad altre procedure e inserisce in quei punti gli indirizzi delle procedure corrispondenti.

La tabella seguente fornisce nome, lunghezza e indirizzo iniziale di tutti i moduli della Figura 7.14.

Modulo	Lunghezza	Indirizzo iniziale
A	400	100
B	600	500
C	500	1100
D	300	1600

### 7.4.2 Struttura di un modulo oggetto

Spesso i moduli oggetto sono composti da sei parti, come mostra la Figura 7.15. La prima parte contiene il nome del modulo, informazioni necessarie al linker (come le lunghezze delle singole parti del modulo), e in alcuni casi la data di assemblaggio.

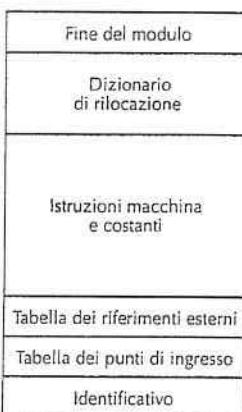


Figura 7.15 Struttura interna di un modulo oggetto prodotto da un traduttore.  
Il campo *identificativo* è il primo.

La seconda parte è una lista di simboli definiti all'interno del modulo al quale possono fare riferimento altri moduli; i simboli sono associati ai valori corrispondenti. Se per esempio il modulo consiste di una procedura chiamata *bigbug*, il primo elemento della tabella conterrà la stringa di caratteri "bigbug" seguita dall'indirizzo nel quale si trova la procedura. Il programmatore in linguaggio assemblativo utilizza la pseudoistruzione PUBLIC della Figura 7.2 per indicare quali simboli devono essere dichiarati come **punti d'ingresso**.

La terza parte del modulo oggetto comprende una lista di simboli utilizzati nel modulo, ma definiti in un altro, e anche una lista che indica quali simboli sono utilizzati dalle varie istruzioni macchina. Questa lista permette al linker di inserire gli indirizzi corretti nelle istruzioni che usano simboli esterni. Una procedura può chiamare altre procedure, tradotte indipendentemente, dichiarando come esterni i loro nomi. Il programmatore in linguaggio assemblativo indica quali simboli devono essere dichiarati come **simboli esterni** utilizzando una pseudoistruzione (per esempio la EXTERN della Figura 7.2). In alcuni calcolatori i punti d'ingresso e i riferimenti esterni sono combinati in un'unica tabella.

La quarta parte del modulo oggetto è costituita dal codice assemblato e dalle costanti. Questa è l'unica parte del modulo oggetto che verrà caricata in memoria per essere eseguita. Le altre parti saranno invece utilizzate dal linker ed eliminate prima che inizi l'esecuzione.

La quinta parte è il dizionario di rilocazione. Com'è possibile vedere nella Figura 7.14 occorre aggiungere una costante di rilocazione alle istruzioni contenenti dei riferimenti a indirizzi di memoria. Il linker, ispezionando la quarta parte del modulo, non ha modo di sapere quali parole di dati contengano istruzioni macchina e quali contengano costanti; questa tabella gli fornisce le informazioni sugli indirizzi che devono essere rilocati. Queste informazioni potrebbero formare una tabella di bit, con un bit per ogni indirizzo potenzialmente da rilocare, oppure una lista esplicita degli indirizzi da rilocare.

La sesta parte è composta da un identificatore della fine del modulo, dall'indirizzo a partire dal quale bisogna iniziare l'esecuzione e, in alcuni casi, da una checksum per rilevare gli errori che possono essere avvenuti durante la lettura del modulo.

La maggior parte dei linker richiede due passate. Durante la prima il linker legge tutti i moduli oggetto e costruisce una tabella con i nomi e le lunghezze dei moduli, oltre a una tabella globale di simboli contenenti tutti i punti d'ingresso e i riferimenti esterni. Durante la seconda passata i moduli oggetto vengono letti, rilocati e collegati uno alla volta.

### 7.4.3 Rilocazione a tempo del binding e dinamica

In un sistema multiprogrammato un programma può essere letto e portato nella memoria centrale, eseguito per un breve periodo di tempo, riportato su disco e poi riletto per essere eseguito un'altra volta. In un sistema di grandi dimensioni, quando è attivo un numero elevato di programmi, è difficile avere la certezza che il programma venga letto e caricato ogni volta nelle stesse locazioni.

La Figura 7.16 mostra che cosa succederebbe se il programma già rilocato della Figura 7.14(b) venisse caricato all'indirizzo 400 invece che all'indirizzo 100 in cui il linker l'aveva inserito originariamente. Tutti gli indirizzi di memoria risulterebbero errati e, per di più, non si potrebbero riutilizzare le informazioni di rilocazione, cancellate molto tempo prima. Anche se queste fossero ancora disponibili, il costo per rilocare tutti gli indirizzi ogni volta che un programma viene riportato in memoria centrale sarebbe troppo alto.

Il problema di spostare programmi che sono già stati collegati e rilocati è profondamente connesso al momento in cui viene completato il collegamento finale tra i nomi simbolici e gli indirizzi assoluti della memoria fisica. Un programma appena scritto contiene dei nomi simbolici per gli indirizzi di memoria, per esempio *BR L*. Il momento nel quale si determina l'effettivo indirizzo della memoria centrale corrispondente a *L* è chiamato **tempo del binding** ("tempo del collegamento").

Esistono almeno sei possibilità:

1. quando il programma viene scritto;
2. quando il programma viene tradotto;
3. quando il programma viene collegato, ma prima che sia caricato;
4. quando il programma viene caricato;
5. quando viene caricato un registro base utilizzato per l'indirizzamento;
6. quando viene eseguita l'istruzione contenente l'indirizzo.

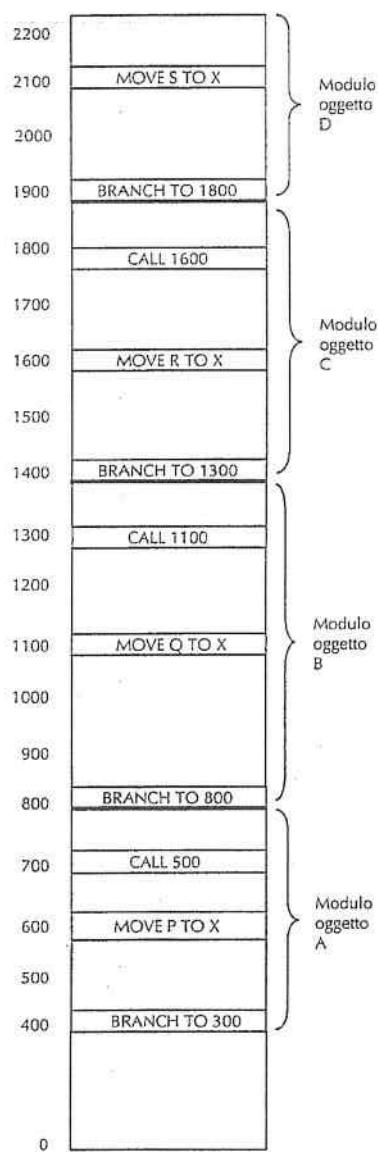


Figura 7.16 Programma binario rilocato della Figura 7.14(b) spostato di 300 posizioni.  
Molte istruzioni si riferiscono ora a indirizzi di memoria errati.

Se un'istruzione contenente un indirizzo di memoria viene spostata dopo il collegamento, il suo riferimento alla memoria risulta errato (assumendo che anche l'oggetto al quale fa riferimento sia stato spostato). Se il traduttore produce come output un eseguibile binario, il momento del collegamento coincide con quello della traduzione e in questo caso il programma deve essere eseguito all'indirizzo assunto dal traduttore. Il metodo descritto nel paragrafo precedente lega i nomi simbolici agli indirizzi assoluti durante la fase di collegamento; questa è la ragione per cui gli spostamenti dei programmi effettuati dopo il collegamento falliscono (come mostra l'esempio della Figura 7.16).

La questione solleva due tipi di problemi. Il primo riguarda il momento in cui i nomi simbolici vengono legati agli indirizzi virtuali, mentre il secondo riguarda il momento in cui gli indirizzi virtuali vengono legati agli indirizzi fisici. Il collegamento si può dire completato solo dopo che queste due operazioni sono state terminate. Quando il linker riunisce i diversi spazi degli indirizzi crea in realtà uno spazio virtuale di indirizzi. La rilocazione e il collegamento hanno l'effetto di collegare i nomi simbolici con specifici indirizzi virtuali. Questa osservazione è vera a prescindere dal fatto che venga o meno utilizzata la memoria virtuale.

Assumiamo per il momento che lo spazio degli indirizzi della Figura 7.14(b) sia paginato. È chiaro che gli indirizzi virtuali corrispondenti ai nomi simbolici *A*, *B*, *C* e *D* sono già stati determinati, anche se i loro indirizzi fisici dipenderanno, in ogni momento in cui verranno utilizzati, dal contenuto della tabella delle pagine. Un programma eseguibile binario è in realtà un collegamento tra nomi simbolici e indirizzi virtuali.

Qualsiasi meccanismo che permetta di modificare facilmente la corrispondenza degli indirizzi virtuali negli indirizzi fisici della memoria faciliterà lo spostamento dei programmi all'interno della memoria principale, anche dopo che essi sono stati legati a uno spazio degli indirizzi virtuali. Uno di questi meccanismi è la paginazione. Dopo che un programma è stato spostato all'interno della memoria centrale occorre modificare solamente la sua tabella delle pagine e non il programma stesso.

Una seconda tecnica consiste nell'uso di un registro di rilocazione durante l'esecuzione (come avveniva per il CDC 6600 e i suoi successori). Sulle macchine che utilizzano questa tecnica il registro punta sempre all'indirizzo fisico della memoria in cui inizia il programma corrente. Il registro di rilocazione viene sommato via hardware a tutti gli indirizzi, prima che essi siano inviati alla memoria. L'intero processo di rilocazione risulta così trasparente ai programmi utente. Quando viene spostato un programma il sistema operativo deve aggiornare il registro di rilocazione. Questo meccanismo è però meno generale rispetto alla paginazione, dato che l'intero programma deve essere spostato come una singola unità (oppure come due, nel caso in cui vi siano registri di rilocazione separati per il codice e per i dati, come nel processore Intel 8088).

Sulle macchine in grado di effettuare riferimenti alla memoria relativi al contatore d'istruzioni è possibile ricorrere a un terzo meccanismo. Grazie a questa funzionalità è possibile sfruttare il fatto che molte istruzioni di salto sono relative al valore di PC. Ogni volta che si sposta un programma nella memoria centrale occorre modificare solamente il PC. Un programma in cui tutti i riferimenti alla memoria sono relativi al PC oppure sono assoluti (per esempio, gli indirizzi assoluti dei registri dei dispositivi di I/O) è detto *indipendente dalla posizione*. Una procedura indipendente dalla posizione può essere

collocata in qualsiasi punto dello spazio degli indirizzi virtuali senza bisogno di effettuare la rilocazione.

#### 7.4.4 Collegamento dinamico

Nella strategia di collegamento descritta nel Paragrafo 7.4.1 tutte le procedure che un programma potrebbe richiamare sono collegate prima che abbia inizio l'esecuzione. Se si completano tutti i collegamenti prima dell'esecuzione, non si trae pieno vantaggio dalla memoria virtuale, di cui un calcolatore può essere dotato. Molti programmi contengono procedure, chiamate soltanto in particolari, e infrequentemente, circostanze. I compilatori, per esempio, hanno alcune procedure per tradurre istruzioni utilizzate raramente, e altre il cui compito è quello di gestire condizioni di errore che si verificano di rado.

Un modo più flessibile per collegare procedure compilate separatamente è quello di collegarle nel momento in cui ciascuna procedura viene caricata. Questo processo è conosciuto con il nome di **collegamento dinamico**. Il primo calcolatore che ha impiegato questa tecnica è stato MULTICS, la cui pionieristica implementazione è, sotto alcuni aspetti, ancora ineguagliata. Nei paragrafi successivi analizzeremo come avviene il collegamento dinamico in alcuni sistemi.

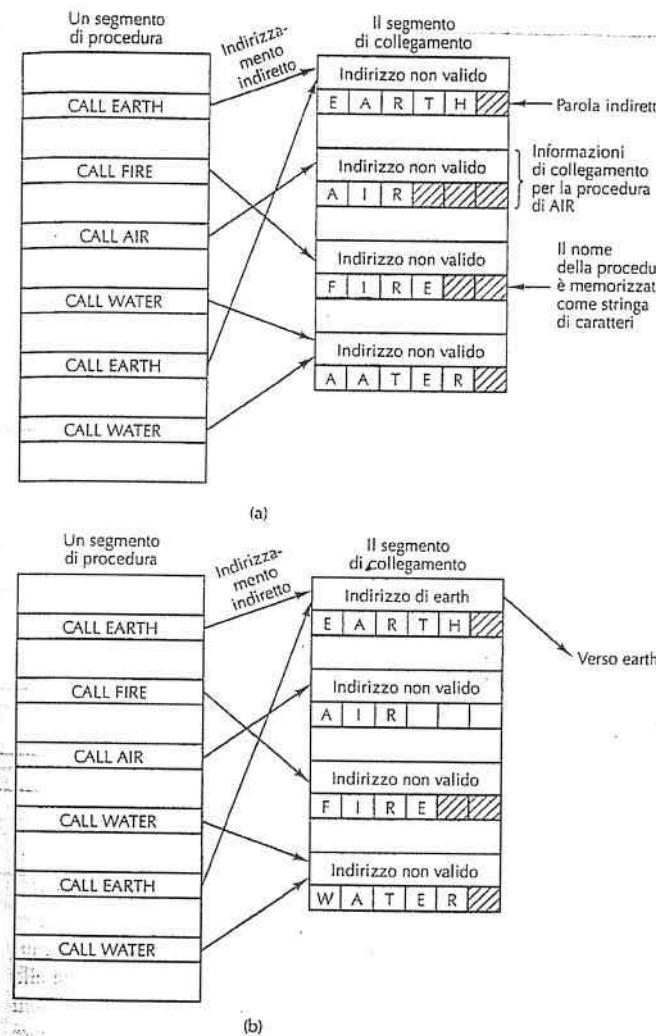
#### Collegamento dinamico in MULTICS

L'implementazione di MULTICS del collegamento dinamico associa a ciascun programma un segmento di collegamento contenente un blocco d'informazioni per ciascuna procedura del programma. Questo blocco d'informazioni inizia con una parola riservata per l'indirizzo virtuale della procedura, seguita dal nome della procedura, memorizzato come stringa di caratteri.

Quando si utilizza il collegamento dinamico le chiamate di procedura nel linguaggio sorgente vengono tradotte in istruzioni che indirizzano, in modo indiretto, la prima parola del blocco di collegamento (la Figura 7.17(a) ne mostra un esempio). Il compilatore riempie questa parola o con un indirizzo non valido oppure con una speciale stringa di bit che determina un'eccezione.

Quando si richiama una procedura che fa parte di un altro segmento, il tentativo di indirizzare la parola non valida solleva nel linker dinamico un'eccezione. Il linker legge quindi la stringa di caratteri che si trova nella parola che segue l'indirizzo non valido e cerca una procedura con questo nome all'interno della directory dell'utente. Viene quindi assegnato un indirizzo virtuale a questa procedura, di solito all'interno del proprio segmento privato; inoltre, come mostra la Figura 7.17(b), questo indirizzo virtuale viene scritto nel segmento di collegamento, al posto dell'indirizzo non valido. Successivamente viene rieseguita l'istruzione che ha provocato l'errore di collegamento e ciò permette al programma di continuare la propria esecuzione dal punto in cui si trovava prima che venisse sollevata l'eccezione.

Tutti i successivi riferimenti alla procedura avverranno senza provocare errori di collegamento, dato che la prima parola del segmento di collegamento contiene ora un indirizzo virtuale valido. Da ciò ne consegue che il linker dinamico viene invocato solamente la prima volta in cui viene richiamata una procedura e non durante le chiamate successive.



**Figura 7.17** Collegamento dinamico. (a) Prima che EARTH sia chiamata. (b) Dopo che EARTH è stata chiamata e collegata.

#### Collegamento dinamico in Windows

Tutte le versioni del sistema operativo Windows, compreso NT, supportano e sfruttano pienamente il collegamento dinamico. Allo scopo viene utilizzato un formato di file chiamato **DLL** (*Dynamic Link Library*, "libreria a collegamento dinamico"). Le DLL

possono contenere procedure, dati oppure entrambi, e sono comunemente utilizzate per permettere a due o più processi di condividere librerie di dati e procedure. Molte DLL hanno l'estensione *.dll*, ma vengono utilizzate anche estensioni diverse, come *.drv* (per le librerie di driver) e *.fon* (per le librerie di caratteri).

Nella sua forma più comune una DLL è una libreria composta da un gruppo di procedure che possono essere caricate in memoria e alle quali possono accedere più processi nello stesso istante. La Figura 7.18 rappresenta due programmi che condividono un file DLL contenente quattro procedure, *A*, *B*, *C* e *D*. Nell'illustrazione il programma 1 usa la procedura *A* e il programma 2 usa la *C*, anche se i due programmi avrebbero comunque potuto utilizzare la stessa procedura.

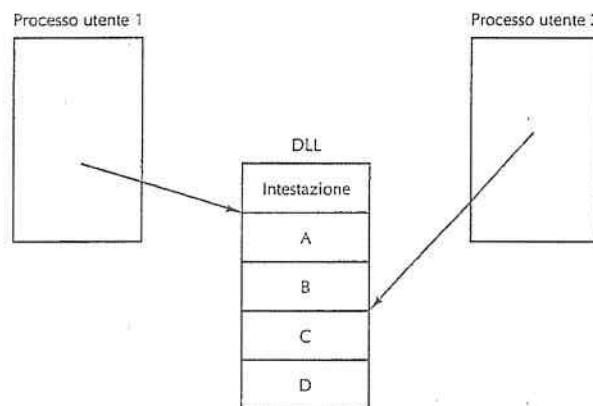


Figura 7.18 Un file DLL usato da due processi.

Il linker costruisce una DLL a partire da un insieme di file di input. La costruzione di una DLL è in pratica molto simile alla creazione di un programma eseguibile binario, tranne che per uno speciale indicatore che viene specificato al linker in modo che crea una DLL. Di solito le DLL vengono costruite a partire da gruppi di librerie di procedure che ci si aspetta verranno utilizzate da più processi. Le procedure d'interfaccia alla libreria delle chiamate di sistema di Windows e le librerie grafiche di grandi dimensioni sono due esempi di DLL molto comuni. L'uso delle DLL ha il vantaggio di risparmiare spazio in memoria e sul disco. Se una libreria usata molto frequentemente venisse collegata staticamente a ciascun programma che ne fa uso, essa comparirebbe all'interno di molti programmi binari eseguibili sia in memoria sia sul disco, sprecando una gran quantità di spazio. Facendo ricorso alle DLL ciascuna libreria è presente una sola volta in memoria e una sola sul disco.

Oltre a risparmiare spazio questo approccio agevola l'aggiornamento delle procedure, anche dopo che i programmi che le utilizzano sono stati compilati e collegati. Per pacchetti software commerciali, di cui raramente gli utenti hanno a disposizione il codi-

ce, l'uso delle DLL permette al venditore di correggere i bug presenti nelle librerie semplicemente ridistribuendo via Internet nuovi file DLL, senza dover cambiare in alcun modo il programma binario principale.

La differenza principale tra una DLL e un eseguibile binario è che la DLL non può essere eseguita da sola (non ha al suo interno un programma principale). Inoltre essa contiene nella sua intestazione informazioni di tipo diverso e possiede delle procedure aggiuntive che non sono legate alle procedure della libreria. È presente per esempio una procedura che viene automaticamente chiamata ogni volta che un nuovo processo è collegato alla DLL e un'altra che viene automaticamente chiamata quando un processo viene invece scollegato. Queste procedure possono allocare o deallocare aree di memoria o gestire altri tipi di risorse richieste dalla DLL.

Ci sono due modi per collegare un programma a una DLL. Nel primo, chiamato **collegamento implicito**, il programma dell'utente viene collegato a un file speciale chiamato **libreria importata**, generato da un programma di utilità che estrae alcune informazioni dalla DLL. La libreria importata costituisce il collante che permette al programma utente di accedere alla DLL. Un programma utente può essere collegato a più librerie importate. Quando viene caricato un programma che utilizza il collegamento implicito, Windows lo esamina per vedere quali DLL utilizza e controlla che tutte siano presenti in memoria. Quelle che non lo sono vengono caricate immediatamente (ma non necessariamente nella loro interezza, dato che sono paginate). Vengono apportati alcuni cambiamenti alle strutture dati delle librerie importate in modo da poter localizzare le procedure richiamate (queste modifiche sono analoghe a quelle viste nella Figura 7.17). Inoltre esse devono essere mappate nello spazio degli indirizzi virtuali del programma. A questo punto il programma utente è pronto per essere eseguito e può chiamare le procedure presenti nella DLL come se fossero staticamente legate al programma stesso.

L'alternativa a questo tipo di collegamento è, ovviamente, il **collegamento esplicito**. Questo approccio non richiede l'uso di librerie importate e non obbliga a caricare le DLL nello stesso momento in cui viene caricato il programma dell'utente. Al contrario il programma dell'utente effettua a tempo di esecuzione una chiamata esplicita per leggere a sé una DLL e successivamente effettua altre chiamate per ottenere gli indirizzi delle procedure di cui ha bisogno. Dopo aver recuperato questi indirizzi il programma può chiamare le procedure. Quando ha finito di utilizzarle effettua una chiamata finale per scollegarsi dalla DLL. Quando anche l'ultimo processo si è scollegato da una DLL questa può essere rimossa dalla memoria.

È importante comprendere che, a differenza dei thread e dei processi, in una DLL le procedure non hanno una propria identità. Queste procedure vengono eseguite nel thread del chiamante e utilizzano lo stack del chiamante per le proprie variabili locali. Possono avere alcuni dati statici specifici di un processo (così come dati condivisi), ma si comportano esattamente come le procedure collegate staticamente. L'unica vera differenza riguarda il modo in cui viene effettuato il collegamento.

### Collegamento dinamico in UNIX

Il sistema UNIX ha un meccanismo che è essenzialmente simile alle DLL di Windows e che si basa su quella che viene chiamata **libreria condivisa**. Analogamente ai file DLL una libreria condivisa è un file archivio contenente procedure o moduli di dati presenti

in memoria a tempo di esecuzione e può essere collegata contemporaneamente a più processi. La libreria standard di C e gran parte del codice relativo alla comunicazione via rete sono esempi di librerie condivise.

UNIX supporta esclusivamente il collegamento implicito e quindi una libreria condivisa consiste in due parti: statica (collegata staticamente al file eseguibile) e destinazione (richiamata a tempo di esecuzione). Anche se alcuni dettagli presentano delle differenze, i concetti sono essenzialmente uguali a quelli delle DLL.

## 7.5 Riepilogo

Anche se la maggior parte dei programmi può, e dovrebbe, essere scritta in un linguaggio ad alto livello, in alcune situazioni è necessario utilizzare, almeno in parte, il linguaggio assemblativo. Candidati alla scrittura in linguaggio assemblativo sono i programmi per calcolatori dotati di poche risorse, come le smart card e i processori integrati in dispositivi come le radiosveglie. Un programma in linguaggio assemblativo è una rappresentazione simbolica di un programma scritto nel linguaggio macchina di un livello inferiore. Esso viene tradotto nel linguaggio macchina mediante un programma chiamato assemblatore.

Per le applicazioni che richiedono un'alta velocità di esecuzione si sceglie di scrivere inizialmente l'intero programma in un linguaggio ad alto livello, e poi si determinano i punti in cui viene spesa la maggior parte del tempo; quindi si riscrivono in linguaggio assemblativo solo le porzioni di codice usate in modo più intenso. Questo approccio ha dei vantaggi rispetto a scrivere l'intero programma in codice assemblativo, dato che in genere solo una piccola frazione del codice è in realtà responsabile della maggior parte del tempo di esecuzione.

Molti assemblatori permettono l'uso delle macro, le quali consentono al programmatore di attribuire nomi simbolici a sequenze di codice usate frequentemente. In un secondo momento questi nomi vengono automaticamente sostituiti con il codice corrispondente. Di solito le macro possono anche essere parametrizzate in modo semplice. Le macro sono implementate mediante algoritmi di elaborazione di stringhe di caratteri.

La maggior parte degli assemblatori prevede due passate. La prima si occupa di costruire una tabella per le etichette, i letterali e gli identificatori dichiarati esplicitamente. I simboli possono essere memorizzati in modo disordinato e reperiti mediante ricerche lineari, oppure possono essere tenuti in ordine e poi individuati con una ricerca dicotomica. Un'altra possibilità per memorizzare e cercare i simboli consiste nell'uso di una tabella hash. Se i simboli non devono essere cancellati durante la prima passata, questo metodo solitamente risulta il migliore. La seconda passata genera il codice. Alcune pseudoistruzioni vengono eseguite durante la prima passata, altre durante la seconda.

I programmi assemblati indipendentemente possono essere collegati fra loro per formare un codice binario che può essere eseguito. Questo lavoro è realizzato dal linker. I suoi compiti principali sono la rilocazione e il collegamento dei nomi. Il collegamento dinamico è una tecnica grazie alla quale alcune procedure non sono collegate fino al momento in cui non vengono effettivamente richiamate. Le DLL di Windows e le librerie condivise di UNIX usano il collegamento dinamico.

### PROBLEMI

1. In un dato programma il 2% del codice è responsabile del 50% del tempo di esecuzione. Si confrontino le seguenti strategie relative ai tempi di programmazione e di esecuzione. Si assuma che il programma richieda 100 mesi/uomo di lavoro per essere programmato in C; la scrittura del codice assemblativo è invece 10 volte più lenta, ma il programma ottenuto risulta quattro volte più efficiente.
  - a. Intero programma in C.
  - b. Intero programma in linguaggio assemblativo.
  - c. Prima tutto in C, poi il 2% più critico riscritto in linguaggio assemblativo.
2. Le considerazioni che valgono per gli assemblatori a due passate, valgono anche per i compilatori?
  - a. Si assume che i compilatori producano moduli oggetto, non codice assemblativo.
  - b. Si assume che i compilatori producano linguaggio assemblativo simbolico.
3. La maggior parte degli assemblatori per x86 ha l'indirizzo destinazione come primo operando e l'indirizzo sorgente come secondo operando. Quali problemi si sarebbero dovuti risolvere per utilizzare gli operandi in posizione invertita?
4. È possibile assemblare in due passate il seguente programma? EQU è una pseudoistruzione che mette in relazione l'etichetta con l'espressione che si trova nel campo dell'operando.
 

```
P EQU Q
Q EQU R
R EQU S
S EQU 4
```
5. La Società di Calcolatori DaDueSoldi sta pianificando di produrre un assemblatore per un calcolatore con parole a 48 bit. Per contenere i costi il responsabile del progetto, il Dott. Paperone, ha deciso di limitare la lunghezza degli identificatori in modo che possano essere memorizzati in una sola parola. Paperone ha dichiarato che gli identificatori possono consistere solamente in lettere, tranne la Q che è vietata (per superstizione: infatti è la 17ª lettera dell'alfabeto inglese). Qual è la massima lunghezza dei simboli? Si descriva un proprio schema di codifica.
6. Qual è la differenza tra istruzioni e pseudoistruzioni?
7. Qual è la differenza (se c'è) tra il contatore di locazioni delle istruzioni (ILC) e il contatore d'istruzioni (PC)? Dopo tutto entrambi tengono traccia della successiva istruzione in un programma.
8. Si mostri la tabella dei simboli dopo che si sono incontrate le seguenti istruzioni per x86. La prima istruzione è assegnata all'indirizzo 1000.
 

EVEREST:	POP BX	(1 BYTE)
K2:	PUSH BP	(1 BYTE)
WHITNEY:	MOV BP,SP	(2 BYTES)
MCKINLEY:	PUSHX	(3 BYTES)
FUJI:	PUSH SI	(1 BYTE)
KIBO:	SUB SI,300	(3 BYTES)
9. È possibile immaginare delle circostanze nelle quali il linguaggio assemblativo permetta che un'etichetta sia identica a un codice operativo (per esempio, MOV come etichetta)? Si motivi la risposta.
10. Si mostri i passi necessari per cercare, utilizzando la ricerca dicotomica, la parola Berkeley all'interno della seguente lista: Ann Arbor, Berkeley, Cambridge, Eugene, Madison, New Haven, Palo Alto, Pasadena, Santa Cruz, Stony Brook, Westwood e Yellow Springs. Quando si calcola l'elemento centrale di una lista con un numero pari di elementi si utilizzi l'elemento che si trova subito dopo il punto medio.
11. È possibile utilizzare la ricerca dicotomica su una tabella la cui dimensione è un numero primo?