Capitolo 4: I linguaggi evoluti

4.1 Introduzione

I linguaggi evoluti sono insiemi di regole sintattiche che consentono di specificare operazioni eseguibili da una generica CPU. Rispetto ai linguaggi macchina, i linguaggi evoluti hanno il vantaggio di rendere possibile l' utilizzo di nomi simbolici per le variabili e per le subroutines; inoltre mettono a disposizione del programmatore un insieme di istruzioni elementari molto più ampio di quello disponibile in un normale linguaggio macchina, ed hanno l' ulteriore vantaggio di essere totalmente indipendenti nella sintassi e nella funzionalità dal tipo di CPU utilizzata. Quindi, grazie all' utilizzo di linguaggi evoluti e di sistemi operativi diventa possibile scrivere programmi in modo del tutto indipendente dal tipo di computer utilizzato.

Naturalmente, siccome alla fine ogni particolare CPU non puó che eseguire programmi scritti nel suo linguaggio macchina, dovrà essere svolta una operazione di traduzione che porti dal programma scritto in linguaggio evoluto al codice eseguibile scritto in linguaggio macchina.

4.2 Compilazione e Link

L' operazione di traduzione di un programma in linguaggio evoluto in uno in linguaggio macchina si chiama compilazione, e viene eseguita per mezzo di un programma specifico detto "compilatore". I compilatori sono specifici, oltre che del linguaggio evoluto utilizzato, anche del tipo di CPU utilizzata, della quale devono conoscere il linguaggio macchina; sono invece di solito indipendenti dai dettagli di configurazione del computer utilizzato.

Il codice in linguaggio macchina generato da un compilatore non è tuttavia eseguibile dalla CPU, e prende il nome di "codice oggetto". Il codice oggetto è in effetti solo la traduzione in linguaggio macchina del programma effettivamente scritto dal programmatore in linguaggio evoluto: mancano tutte quelle routines che permettono di realizzare in linguaggio macchina le operazioni elementari in linguaggio evoluto, come pure mancano tutte le routines specifiche del sistema operativo che rendono possibile il dialogo con le periferiche. Ad esempio, in qualunque linguaggio evoluto la stampa sul monitor del valore di una variabile reale chiamata `A` si può eseguire con una semplice istruzione del tipo `PRINT A`. Dal punto di vista della CPU questo significa leggere la memoria alla locazione corrispondente alla variabile `A`, decodificarla tenendo conto della codifica utilizzata, ricavare una rappresentazione corrispondente in base 10, scriverla su una stringa e passare questa stringa al driver del monitor perché la visualizzi. Queste operazioni vengono eseguite da routines specifice del linguaggio e del sistema operativo che devono essere prelevate da opportune librerie. L' operazione di completamento del codice oggetto con le opportune routines prelevate dalle librerie di linguaggio o di sistema operativo si chiama "link". Durante tale operazione possono essere aggiunte al codice oggetto anche routines prelevate da librerie utente.

Osserviamo infine che nel sistema operativo UNIX le operazioni di compilazione e link vengono di solito eseguite dallo stesso comando (**cc** per il C); è però possibile, usando opportune opzioni dei comandi (vedere Par.5.2), eseguirle separatamente.

4.3 Elementi di un linguaggio evoluto

In un linguaggio evoluto si possono distinguere diverse componenti:

- Sintassi: indica l' insieme delle regole generali che governano la scrittura del programma.

- Struttura: indica il modo in cui è possibile suddividere un programma complesso in sottoprogrammi e fornisce le regole per far comunicare tra loro i sottoprogrammi.
- Variabili: sono le regole per la definizione e l' utilizzo delle aree di memoria che contengono i dati da elaborare.
- Controllo di Flusso: sono le regole per eseguire test, cicli etc.
- Funzioni: sono i comandi per eseguire operazioni complesse, sia di tipo aritmetico che di input/output.

Nel seguito ci occuperemo degli aspetti generali (cioè sostanzialmente indipendenti dal linguaggio) di variabili e struttura. Sintassi, Controllo di Flusso e Funzioni sono invece specifiche dei diversi linguaggi; per il C, un breve riassunto di queste caratteristiche si trova al Capitolo 5

4.4 Le variabili

Una variabile è una porzione di memoria a cui il compilatore assegna un nome. È importante ricordare che il nome ha un senso solo nell' ambito del programma in linguaggio evoluto; dopo la compilazione resteranno solo gli indirizzi numerici di memoria, ed il programma eseguibile non saprà più nulla dei nomi. Soltanto nel caso la compilazione venga eseguita con l' opzione **-g** al codice eseguibile verrà aggiunta una tabella di corrispondenza nome-indirizzo che consentrà al "debugger simbolico" (Par 3.9) di accedere, nel corso dell' esecuzione del programma, alle variabili utilizzando il loro nome.

Nel seguito esamineremo le caratteristiche principali delle variabili.

4.4.1 Tipo e dimensione di una variabile

Si dice "tipo" di una variabile il metodo di codifica utilizzato per scrivere i dati in memoria. Come abbiamo visto esistono codifiche adatte ai numeri interi, ai numeri reali, alle variabili alfanumeriche e così via; i diversi linguaggi utilizzanno diverse parole chiave per indicare lo stesso tipo di codifica; si veda Tab. 5.2 per i nomi utilizzati dal C.

Il tipo di codifica utilizzato determina anche la lunghezza in bits di un elemeto di quel tipo: ad esempio un reale in singola precisione è scritto su 32 bits, ovvero su 4 bytes, mentre un reale in doppia precisione occupa 64 bits, ovvero 8 bytes. Oltre alla lunghezza di un singolo elemento i linguaggi evoluti consentono di creare vettori e matrici di singoli elementi di un dato tipo. La dimensione di una variabile sarà quindi data dal numero complessivo di elementi del vettore o della matrice; vale la pena di ricordare che la lunghezza totale occupata in memoria da una variabile sarà data dal prodotto della dimensione per la lunghezza in bytes del singolo elemento.

4.4.2 Posizione in memoria degli elementi di una matrice

Gli elementi di un vettore vengono sempre posti in memoria in locazioni consecutive. Per quel che concerne le matrici e i vettori a più indici invece la disposizione degli elementi in memoria è meno ovvia; in particolare in C la disposizione degli elementi avviene nell' ordine $i[0][0]$, $i[0][1]$, ..., $i[0][n-1]$, $i[1][0]$, ... L' ordine esatto della disposizione delle matrici in memoria è importante, in quanto molto spesso i computers accedono molto più velocemente a blocchi contigui di memoria che non a elementi sparsi; quindi quando si realiza un loop che legge o scrive gli elementi di un vettore è bene cercare di assecondare l' ordine naturale della posizione dei singoli elementi: basterà ricordare che in C devono scorrere più velocemente gli indici più a destra.

4.4.3 Tipi composti

I tipi composti sono agglomerati di variabili standard (sia scalari che vettori o matrici). Una volta definiti (vedere Par. 5.12), i tipi composti possono essere utilizzati, come quelli predefiniti, per definire variabili scalari, vettori o matrici. Pure una variabile appartenente a un tipo composto può essere passata come argomento ad una subroutine specificandone semplicemente il nome: per questo motivo i tipi composti consentono di semplificare molto la scrittura dei programmi ed il trasferimento di dati tra diverse routines e tra i programmi ed i files di dati. La sola limitazione, superata solo nei linguaggi "object oriented" (come il C++), è che non si possono definire operazioni sui tipi composti, ma si deve sempre ricorrere alle operazioni di base facendole agire sui singoli "campi" che definiscono il tipo composto.

4.5 Allocazione statica ed automatica della memoria

Con allocazione della memoria si indica l' operazione che il compilatore esegue quando riserva lo spazio di memoria necessario per ospitare una variabile.

L' allocazione statica viene eseguita al momento della compilazione, e di conseguenza non può essere alterata nel corso dell' esecuzione. Questo metodo di allocazione viene usato normalmente dai compilatori per le variabili globali di un programma: queste variabili vengono allocate una volta per tutte e rimangono nella medesima posizione di memoria durante tutta l' esecuzione del programma. In questo modo possono essere utilizzate con sicurezza da tutte le componenti del programma.

L' allocazione statica è svantaggiosa per variabili che non sono necessarie durante tutto lo svolgimento del programma, ma solo durante fasi particolari, in quanto mantiene lo spazio relativo occupato inutilmente. In questi casi viene usata l' allocazione automatica, che consiste nel riservare lo spazio in memoria al momento in cui la variabile deve essere utilizzata (durante l' esecuzione del programma quindi) e nel rilasciarlo quando la variabile non serve più. Questa tecnica è utilizzata normalmente per le variabili locali delle subroutine, che vangono allocate al momento della chiamata della routine e buttate via quando la routine termina l' esecuzione. Di conseguenza non si deve mai fare affidamento sul fatto che una variabile locale conservi il suo valore tra una chiamata e la successiva, a meno che non si utilizzino dichiarazioni particolari (Par. 5.10) che forzano l' allocazione statica anche per le variabili locali.

4.6 I puntatori

I metodi di allocazione che abbiamo visto fino ad ora sono eseguiti e comandati dal compilatore, vuoi in fase si compilazione (allocazione statica) vuoi in fase di esecuzione (allocazione automatica). In alcuni casi tuttavia è necessario poter intervenire direttamente sul meccanismo di allocazione delle variabili: il caso tipico è quello che si presenta quando si devono disporre n dati in un vettore, ma n non è noto al momento della compilazione ma viene deciso volta per volta al momento dell' esecuzione del programma. In questo caso si vorrebbe decidere la lunghezza del vettore, e quindi eseguire l' allocazione, al momento dell' esecuzione.

Lo strumento che consente di intervenire sul processo di allocazione è il "puntatore". Si tratta di una normale variabile, di solito allocata staticamente, che contiene, invece di un dato, l' indirizzo in memoria di un' altra variabile.

Quamdo si accede ad una variabile per mezzo di un puntatore il sistema prima legge il contenuto del puntatore, quindi lo utilizza per trovare la posizione in memoria sulla quale effettuare l' operazione richiesta. La variabile "puntata" dal puntatore viene detta "target".

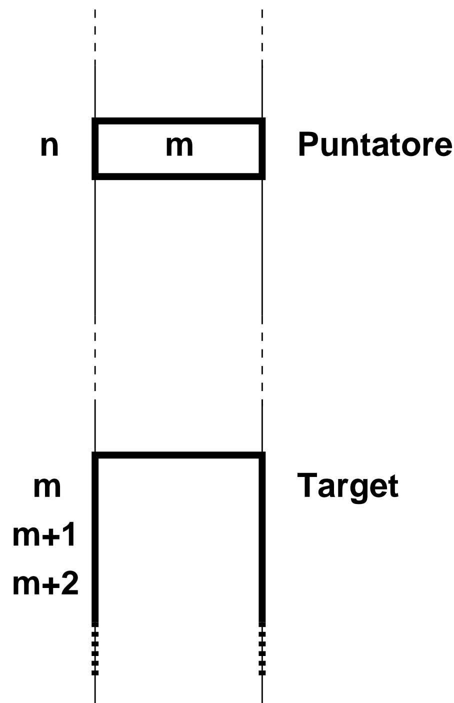

Figura 4.1:

4.6.1 Allocazione dinamica della memoria

Grazie all' utilizzo dei puntatori e di speciali routines del sistema operativo che consentono, nel corso dell' esecuzione di un programma, di cercare aree di memoria libere e di riservarle ad un particolare uso, è possibile realizzare la allocazione dinamica delle variabili in memoria.

La sola operazione necessaria in fase di compilazione consiste nel definire un puntatore. Durante l' esecuzione del programma, una volta deciso quanto spazio è necessario per la variabile che vogliamo allocare, dovremo solo chiedere al sistema operativo di trovare una zone di memoria sufficientemente grande e dovremo scrivere nel puntatore l' indirizzo iniziale di questa zona. Ciò fatto potremo accedere alla nostra variabile tramite il puntatore come se si trattasse di una variabile allocata in modo statico; la variabile sarà locale o globale a seconda che il puntatore, e quindi l' indirizzo in esso contenuto, siano leggibili o no da diverse routines. Quando la nostra variabile non ci servirà più potremo rilasciare lo spazio ed azzerare il puntatore, o eventualmente potremo riutilizzare il puntatore per puntare a un target di dimensione differente: in questo modo sarà possibile, ad esempio, incrementare progressivamente, secondo le esigenze che si presentano nel corso dell' esecuzione, la dimensione di un vettore allocato dinamicamente.

Malgrado gli enormi vantaggi di versatilità e flessibilità che presenta, l' allocazione dinamica scarica completamente sul programmatore la responsabilità di controllare la corretta gestione della memoria. Per questo motivo va utilizzata solo quando realmente necessario e facendo la dovuta attenzione.

Al Paragrafo 5.13 sono descritte le tecniche usate in C per realizzare praticamente l' allocazione dinamica. Riassumiamo qui solo i concetti fondamentali. Il C utilizza simboli specifici per il puntatore ed il target (se `p` è un puntatore `*p` è il suo target. I puntatori si comportano all' incirca come variabili intere, per cui le operazioni sui puntatori si possono eseguire (quasi) come su normali interi positivi. Ogni puntatore è specifico del tipo del target (i puntatori a

interi sono diversi dai puntatori a float ad esempio) e può essere usato indifferentemente per puntare a scalari o a vettori; nel secondo caso la notazione per l' accesso al vettore è identica a quella che si usa per i vettori allocati staticamente.

4.7 Programmazione strutturata

Con il termine "programmazione strutturata" si intende la suddivisione di un programma complicato in sottoprogrammi aventi ciascuno una funzionalità ben definita. Questo metodo di programmazione consente di semplificare notevolmente le fasi di messa a punto di un programma, dal momento che i vari pezzi possono essere provati e messi a punto indipendentemente, anche da persone differenti. Inoltre sottoprogrammi scritti in un particolare contesto possono poi essere riutilizzati, sostanzialmente senza modifiche in situazioni diverse.

Un altro vantaggio della programmazione strutturata è la possibilità di essere applicato in una progettazione del software di tipo top-down, nella quale cioè il progetto complessivo viene suddiviso in parti più semplici, per ciascuna delle quali è definita una "interfaccia" con le altre, e la procedura è ripetuta iterativamente per ciascun sottoprogetto fino ad arrivare a blocchi elementari. Il vantaggio della progettazione top-down è quello di fornire descrizioni schematiche a diversi livelli di progetti complessi, consentendo a chi lavora in un particolare settore e quindi conosce i dettagli di quella parte, di avere delle descrizioni funzionali di massima du tutte le rimanenti componenti senza per altro dover vemire a conoscenza di alcun dettagio realizzativo.

Gli elementi dei linguaggi di programmazione che consentono la realizzazione dei blocchi elementari di un programma strutturato sono le subroutines.

4.8 Subroutines

Una subroutine è un insieme di istruzioni che vengono eseguite in sequenza; quando la sequenza termina, o quando viene incontrata un' istruzione specifica detta "return" la subroutine termina e il programma riprende dall' istruzione successiva a quella che ne ha causato l' attivazione (istruzione che appartiene per definizione ad un' altra subroutine). Una subroutine particolare è il "main program". Il main program non viene chiamato da nessuna subroutine, ma viene attivato al momento in cui il programma parte; quando la sequenza di istruzioni del main program termina anche il programma termina.

Ogni subroutine può definire al suo interno variabili locali. Tali variabili sono accessibili solo alla subroutine che le dichiara e, in assenza di differenti direttive, sono allocate in modo automatico, per cui vanno perdute non appena l' esecuzione della subroutine termina.

Nella logica della programmazione strutturata, le routines devono comunicare tra loro, scambiandosi dati e risultati delle elaborazioni, e questo non può essere fatto usando variabili locali. Ogni subroutine possiede quindi un insieme di parametri, che vengono usualmente specificati tra parentesi dopo il nome; si tratta variabili fittizie o "dummy", per le quali cioè non esiste una allocazione di memoria. Ciò nonostante possono essere utilizzate nell' ambito della subroutine come normali variabili: infatti, al momento della chiamata, il programma chiamante sepcificherà quali delle sue variabili locali devono essere passate come parametri, e quando la subroutine utilizzerà i nomi delle variabili dummy lavorerà in realtà su queste variabili.

In questo modo si può fare sì che due subroutines comunichino tra loro; in più si può ottenere che una subroutine esegua le operazioni per cui è stata progettata su variabili diverse semplicemente passando parametri differenti in successive chiamate: la subroutine non dovrà sapere nulla dei veri nomi delle variabili su cu lavora, in quanto utilzzerà sempre i nomi delle variabili dummy.

Bisogna sempre fare molta attenzione a che il programma chiamante passi come parametri delle variabili identiche, per tipo e dimensione, a quelle specificate nella definizione della subroutine

come variabili fittizie; in caso contrario i dati scritti in memoria dal programma chiamante saranno decodificati in modo errato dalla subroutine ed i risultati saranno pure errati.

La sintassi per la definizione e la chiamata delle subroutines è descritta al Par. 5.6.

4.8.1 Passaggio dei parametri alle subroutines

In C il passaggio dei parametri alle subroutines avviene "by value": per ogni parametro passato alla subroutine viene eseguita una copia, e questa copia è resa accessibile come variabile dummy; il sottoprogramma può quindi leggere i parametri, ma eventuali modifiche verranno eseguite sulla copia, e non saranno quindi visibili al programma chiamante quando la subroutine sarà terminata. Un modo per ovviare a questa difficoltà consiste nel passare come parametro non una variabile ma un puntatore ad essa (passaggio "by reference"): in questo modo verrà eseguita una copia del puntatore, ma la subroutine, leggendo l' indirizzo della variabile, potrà comunque leggerla e modificarne il contenuto. Questa tecnica viene implicitamente usata per i vettori, in quanto in C il nome di un vettore non seguito da parentesi quadre indica il puntatore al vettore.

4.8.2 Functions

Una variante delle subroutine sono le functions, che si usano di solito per implementare funzioni matematiche o comunque operazioni assimilabili a funzioni matematiche. Le functions posseggono un parametro dummy aggiuntivo, che è associato al nome stesso della function e che viene sempre e solo passato (by value) dalla function al programma chiamante. In questa maniera il nome di una function seguito dalla lista dei parametri posta tra parentesi può essere utilizzato dal programma chiamante all' interno di espressioni aritmetiche o di istruzioni di assegnazione o di test, esattamente come se si trattasse di una normale variabile; tuttavia mentre una normale variabile viene semplicemente letta dalla memoria e usata nell' espressione, una function viene chiamata come una subroutine, le vengono passati i parametri e viene eseguito il codice eseguibile in essa contenuto che termina con la assegnazione della variabile dummy di ritorno; questo valore viene poi sostituito nell' espressione. Di conseguenza l' utilizzo di functions in espressioni, specie all' interno di cicli ripetuti molte volte, può risultare ben più "CPU time consuming" dell' uso di una normale variabile.

Va notato che in C di fatto esistono solo le functions. Una subroutine è vista come una function che ritorna una variabile di tipo `void`.

4.8.3 Ricursività

La ricursività è la capacità da parte di una function di richiamare se stassa. Da quello che abbiamo detto sull' allocazione automatica delle variabili locali si intuische che linguaggi come il C ammettono la ricursività. In effetti ad ogni chiamata (o auto–chiamata) viene creato un nuovo insieme di variabili locali, e di conseguenza si cancellano le possibili interferenze tra due istanze della stessa function contemporaneamente presenti nello stack delle subroutines. Va però osservato che le variabili globali o le variabili locali statiche sono potenzialmente fonti di interferenza tra chiamate ricursive, e vanno usate con molta attenzione nel caso si implementino algoritmi ricursivi.

4.8.4 Moduli e variabili globali

L' utilizzo dei parametri come metodo di comunicazione tra le routines può diventare scomodo se si ha un gruppo di routines che lavorano su un numero elevato di variabili; in questo caso si preferisce ricorrere alle variabili globali. Si tratta di variabili, allocate staticamente, che sono visibili non ad una sola routine ma a gruppi di routines.

Il meccanismo più sicuro per l' implementazione delle variabili globali consiste nella realizzazione di un modulo, ovvero di un gruppo di routines: ogni routine definirà al suo interno le proprie variabili locali, ma tutte le variabili definite entro il modulo ma fuori da ogni routine saranno automaticamente considerate comuni a tutto il modulo. Le regole per la realizzazione dei moduli si trovano al Par. 5.5.

Vale la pena di osservare come l' approccio delle variabili globali cancelli l' indipendenza delle subroutines dai nomi delle variabili su cui esse agiscono: i nomi delle variabili globali sono fissi, e quindi una subroutine che agisce su una variabile globale agisce sempre e solo su quella (a meno che la variabile globale non sia un puntatore che viene ridefinito dinamicamente nel corso dell' esecuzione...)

Se le routines singole sono quindi adatte ad implementare blocchi funzionali semplici, che, nell' ambito di un programma possono essere applicati a diversi insiemi di dati, i moduli vanno utilizzati per per implementare blocchi complessi, che agiscono su un' unica base di dati.

4.8.5 La programmazione object–oriented

I moduli sono collezioni di procedure che agiscono su una base di dati fissata ed univoca. In alcune circostanze questa limitazione (che pure può essere aggirata con un uso sapiente di puntator ed allocazione dinamica) risulta fastidiosa. È il caso delle interfacce grafiche a finestre: ogni finestra corrsiponde a dati diversi, ma i programmi di manipolazione sono similari; si deisdererebbe quindi un accoppiamento naturale tra dati e subroutines, del tipo che tutte le volta che viene creata la struttura dati corrispondente ad una finestra venisse pure creata una copia delle routines di manipolazione specifiche per quella particolare finestra.

Questa èla logica dei linguaggi di programmazione object-oriented (come ad esempio il C++). In questi linguaggi è possibile definire dei tipi composti (detti classi) che contengono, oltre che variabili, subroutines e functions. Ogni volta che si crea una variabile del tipo composto (nel linguaggio OO, ogni volta che si istanzia una classe) vengno anche create copie dedicate delle funzioni. Questo incapsulamento del codice risulta molto utile in tutte quelle circostanze in cui sia necessario, all' interno di un singolo programma, di realizzare molte ripetizioni di algoritmi identici o molto simili agenti su basi di dati differenti.

4.8.6 Interfacce con librerie di routines

Le routines che appartengono ad uno stesso modulo si conoscono tra loro, e di conseguenza si possono chiamare senza bisogno di particolari definizioni; inoltre il compilatore sarà in grado di controllare se le variabili dummy ed i parametri effettivamente passati sono definiti in modo coerente. Si dice in questo caso che l' interfaccia tra le varie routines di un modulo è definita implicitamente.

In alcuni casi tuttavia si devono chiamare routines per le quali non è disponibile una interfaccia implicita: in tal caso si deve creare una interfaccia esplicita, che, nella versione minimale, deve contenere la definizione del tipo dei parametri di ritorno di tutte le function che si desidera utilizzare. Tale definizione è identica a quella di una variabile, con in aggiunta l'opzione "extarnal" (vedere Par. 5.6).

L' interfaccia minimale tuttavia non consente al compilatore di controllare la coerenza dei parametri passati alle subroutines; se si desidera (ed è molto consigliabile) rendere possibile questo test si deve creare una interfaccia completa che specifichi, per ogni subroutine o function, tipo e dimensione di tutti i parametri. La sintassi è descritta al Paragrafo 5.7.