

```

d) currentPtr = startPtr;
   while (currentPtr != NULL) {
      printf("Lastname = %s\nGrade = %.2f\n",
             currentPtr->lastName, currentPtr->grade);
      currentPtr = currentPtr->nextPtr;
   }
e) currentPtr = startPtr;
   while (currentPtr != NULL) {
      tempPtr = currentPtr;
      currentPtr = currentPtr->nextPtr;
      free(tempPtr);
   }
   startPtr = NULL;

```

12.5 La visita simmetrica è:

11 18 19 28 32 40 44 49 69 71 72 83 92 97 99

La visita anticipata è:

49 28 18 11 19 40 32 44 83 71 69 72 97 92 99

La visita differita è:

11 19 18 32 44 40 28 69 72 71 92 99 97 83 49

Esercizi

12.6 Scrivete un programma che colleghi due liste concatenate di caratteri. Il programma dovrà includere la funzione **concatenate**, che prenda come argomenti i puntatori a entrambe le liste e concatensi la seconda alla prima.

12.7 Scrivete un programma che unisca due liste ordinate di valori interi in una singola lista ordinata di valori interi. La funzione **merge** dovrà ricevere i puntatori al primo nodo di ognuna delle liste da unire, e dovrà restituire un puntatore al primo nodo della lista unita.

12.8 Scrivete un programma che inserisca ordinatamente in una lista concatenata 25 interi casuali compresi tra zero e 100. Il programma dovrà calcolare la somma degli elementi e la loro media con un valore in virgola mobile.

12.9 Scrivete un programma che crei una lista concatenata di 10 caratteri e, in seguito, create una copia della lista con gli elementi in ordine inverso.

12.10 Scrivete un programma che prenda in input una linea di testo, e utilizzi una pila per visualizzare la linea in ordine inverso.

12.11 Scrivete un programma che usi una pila per determinare se una stringa è palindroma (ovverosia, se possa essere letta allo stesso modo in entrambi i sensi). Il programma dovrà ignorare gli spazi e la punteggiatura.

12.12 Le pile sono usate dai compilatori come supporto per il processo di valutazione delle espressioni e di generazione del codice in linguaggio macchina. In questo e nel prossimo esercizio, esamineremo con cura il modo in cui i compilatori valutano delle espressioni aritmetiche che consistano solamente di costanti, operatori e parentesi.

Gli esseri umani scrivono generalmente delle espressioni come **3 + 4 e 7 / 9**, in cui l'operatore (+ o /, in questo caso) è scritto tra i suoi operandi, ovverosia in *notazione infissa*. I computer "preferiscono" invece la *notazione polacca inversa*, con l'operatore scritto a destra dei suoi due operandi. Con la

notazione polacca inversa, le due precedenti espressioni infisse apparirebbero rispettivamente come
3 4 + e 7 9 /.

Per valutare un'espressione in notazione infissa complessa, il compilatore la convertirà prima nella notazione polacca inversa e quindi la valuterà in questa sua nuova versione. Ognuno di questi algoritmi richiede soltanto un singolo passaggio da sinistra a destra sull'espressione. Ogni algoritmo utilizza una pila per supportare la sua attività, anche se in ognuno di essi la pila è usata per uno scopo diverso.

In questo esercizio scriverete una versione C dell'algoritmo di conversione da notazione infissa a polacca inversa. Nel prossimo esercizio, scriverete una versione C dell'algoritmo di valutazione di un'espressione in notazione polacca inversa.

Scrivete un programma che converta un'ordinaria espressione aritmetica in notazione infissa (supponete che ne sia stata immessa una valida), formata da interi di una sola cifra come

(6 + 2) * 5 - 8 / 4

in un'espressione in notazione polacca inversa. La versione polacca inversa della precedente espressione in notazione infissa è

6 2 + 5 * 8 4 / -

Il programma dovrà leggere l'espressione e immagazzinarla nel vettore di caratteri **infix** e, in seguito, utilizzare la versione modificata delle funzioni per la gestione di una pila implementate in questo capitolo, per aiutare a creare l'espressione in notazione polacca inversa nel vettore di caratteri **postfix**. L'algoritmo per creare un'espressione in notazione polacca inversa è il seguente:

- 1) Inserite nella pila una parentesi aperta '**(**'.
- 2) Accodate alla fine di **infix** una parentesi chiusa '**)**'.
- 3) Fintanto che la pila non è vuota, leggete **infix** da sinistra a destra ed eseguite le seguenti operazioni:

Nel caso in cui il carattere corrente di **infix** sia un numero, copiatelo nell'elemento successivo di **postfix**.

Nel caso in cui il carattere corrente di **infix** sia una parentesi aperta, inseritela nella pila.

Nel caso in cui il carattere corrente di **infix** sia un operatore,

Estraete gli operatori (se ce ne sono) dalla testa della pila, fintanto che abbiano una priorità maggiore o uguale a quella dell'operatore corrente, e inseriteli in **postfix**.

Inserite nella pila il carattere corrente in **infix**.

Nel caso in cui il carattere corrente di **infix** sia una parentesi chiusa,

Estraete gli operatori dalla testa della pila e inseriteli in **postfix**, finché non ci sarà una parentesi aperta sulla cima della pila.

Estraete (ed eliminate) la parentesi aperta dalla pila.

In un'espressione dovranno essere consentite le seguenti operazioni aritmetiche:

+	addizione
-	sottrazione
*	moltiplicazione
/	divisione
^	elevamento a potenza
%	modulo

La pila dovrà essere gestita con le seguenti dichiarazioni:

```
struct stackNode {
    char data;
    struct stackNode *nextPtr;
};
```

```

typedef struct stackNode STACKNODE;
typedef STACKNODE *STACKNODEPTR;

Il programma dovrà consistere di una funzione main e di altre otto funzioni, con le seguenti intestazioni:
void convertToPostfix(char infix[], char postfix[])
Convertirà la notazione infissa in quella polacca inversa.

int isOperator(char c)
Determinerà se c è un operatore.

int precedence(char operator1, char operator2)
Determinerà se la priorità di operator1 è minore, uguale o maggiore di quella di operator2.
La funzione restituirà rispettivamente -1, 0 o 1.

void push(STACKNODEPTR *topPtr, char value)
Inserirà un valore in cima alla pila.

char pop(STACKNODEPTR *topPtr)
Estrarrà un valore dalla cima della pila.

char stackTop(STACKNODEPTR topPtr)
Restituirà il valore contenuto in cima alla pila, senza estrarlo dalla stessa.

int isEmpty(STACKNODEPTR topPtr)
Determinerà se la pila è vuota.

void printStack(STACKNODEPTR topPtr)
Visualizzerà la pila.

```

12.13 Scrivete un programma che valuti un'espressione in notazione polacca inversa (supponete che sia valida) come

6 2 + 5 * 8 4 / -

Il programma dovrà leggere e immagazzinare in un vettore un'espressione in notazione polacca inversa formata da numeri e operatori. Usando la versione modificata delle funzioni per la gestione di una pila implementate in precedenza in questo capitolo, il programma dovrà analizzare l'espressione e valutarla. L'algoritmo è il seguente:

- 1) Accodate il carattere **NULL** ('\0') alla fine dell'espressione in notazione polacca inversa. Nel momento in cui il carattere **NULL** sarà stato incontrato, non saranno necessarie ulteriori operazioni.
- 2) Fintanto che '\0' non sia stato incontrato, leggete l'espressione da sinistra a destra. Nel caso in cui il carattere corrente sia un numero, inserite il suo valore intero in testa alla pila. Ricordate che il valore intero di un carattere numerico è quello assunto nell'insieme dei caratteri del computer meno il valore di '0'. Altrimenti, qualora il carattere corrente sia un operatore, estraete dalla cima della pila i primi due elementi e immagazzinatevi nelle variabili **x** e **y**. Calcolate **y operatore x**. Inserite il risultato del calcolo in testa alla pila.
- 3) Estraete il valore contenuto in cima alla pila, qualora nell'espressione abbiate incontrato il carattere **NULL**. Quel valore sarà proprio il risultato dell'espressione in notazione polacca inversa.

Nota: nel precedente punto 2), quando l'operatore sarà '*/*', la testa della pila conterrà 2 e il suo elemento successivo sarà 8, allora dovrete estrarre il 2 in *x*, il valore 8 in *y*, calcolare $8 / 2$ e reinserirlo nella pila il risultato, 4. Questa nota si applicherà anche all'operatore '*-*'. Ecco un elenco delle operazioni aritmetiche che saranno consentite in un'espressione:

+	addizione
-	sottrazione
*	moltiplicazione
/	divisione
^	elevamento a potenza
%	modulo

La pila dovrà essere gestita con le seguenti dichiarazioni:

```
struct stackNode {
    int data;
    struct stackNode *nextPtr;
};

typedef struct stackNode STACKNODE;
typedef STACKNODE *STACKNODEPTR;
```

Il programma dovrà essere formato dal **main** e da altre sei funzioni con le seguenti intestazioni:

int evaluatePostfixExpression(char *expr)

Valuterà l'espressione in notazione polacca inversa.

int calculate(int op1, int op2, char operator)

Calcolerà il valore dell'espressione **op1 operator op2**.

void push(STACKNODEPTR *topPtr, int value)

Inserirà un valore in cima alla pila.

int pop(STACKNODEPTR *topPtr)

Estrarrà un valore dalla cima della pila.

int isEmpty(STACKNODEPTR topPtr)

Determinerà se la pila è vuota.

void printStack(STACKNODEPTR topPtr)

Visualizzerà la pila.

12.14 Modificate il programma per la valutazione delle espressioni in notazione polacca inversa dell'Esercizio 12.13, in modo che possa elaborare degli operandi interi maggiori di nove.

12.15 (Simulazione di un supermercato) Scrivete un programma che simuli una fila alla cassa di un supermercato. La fila è una coda. I clienti arriveranno casualmente in intervalli interi compresi tra 1 e 4 minuti. Anche i clienti saranno serviti in modo casuale ad intervalli interi compresi tra 1 e 4 minuti. Ovviamente, i ritmi dovranno essere bilanciati. La coda crescerà all'infinito qualora il ritmo degli arrivi sia superiore a quello del servizio. Nonostante i ritmi bilanciati, il caso potrebbe causare delle lunghe code. Eseguite la simulazione del supermercato per una giornata lavorativa di 12 ore (720 minuti), usando il seguente algoritmo:

- 1) Scegliete un intero casuale compreso tra 1 e 4, per determinare il minuto in cui arriverà il primo cliente.

- 2) Nel momento in cui arriva il primo cliente:
 Determinate il tempo di servizio del cliente (un intero casuale compreso tra 1 e 4);
 Cominciate a servire il cliente;
 Stabilite il tempo di arrivo del prossimo cliente (un intero casuale compreso tra 1 e 4 aggiunto al tempo corrente).
- 3) Per ogni minuto del giorno:
 Qualora arrivi il cliente successivo:
 Mettete il cliente nella coda;
 Stabilite il tempo di arrivo del prossimo cliente;
 Nel caso in cui sia stato completato il servizio per l'ultimo cliente:
 Comunicatelo
 Togliete dalla coda il prossimo cliente da servire
 Determinate il tempo di completamento del servizio per il cliente (ovverosia, un intero casuale da 1 a 4 aggiunto al tempo corrente).

Eseguite ora la vostra simulazione e ponetevi le seguenti questioni:

- a) Qual è stato il numero massimo di clienti in coda durante tutta la simulazione?
- b) Qual è stata l'attesa più lunga tra quelle sperimentate dai clienti?
- c) Che cosa succederebbe se l'intervallo di arrivo, compreso tra 1 e 4 minuti, fosse cambiato in uno compreso tra 1 e 3 minuti?

12.16 Modificate il programma della Figura 12.19 in modo da consentire l'inserimento di valori duplicati nell'albero binario.

12.17 Scrivete un programma basato su quello della Figura 12.19 che prenda in input una linea di testo, separi la frase in parole, le inserisca in un albero di ricerca binaria e visualizzi la visita simmetrica, anticipata e differita dell'albero.

Suggerimento: leggete la linea di testo e immagazzinate la in un vettore. Usate **strtok** per dividere in parole il testo. Nel momento in cui avrete isolato una parola dovete creare un nuovo nodo nell'albero, assegnare il puntatore restituito da **strtok** al membro **string** del nuovo nodo e inserirlo nell'albero.

12.18 In questo capitolo, abbiamo visto che l'eliminazione dei duplicati è semplice, qualora si stia creando un albero di ricerca binaria. Descrivete in che modo eseguireste l'eliminazione dei duplicati usando soltanto un vettore a una dimensione. Confrontate le prestazioni dell'operazione di eliminazione dei duplicati basata sul vettore con quella basata sull'albero di ricerca binaria.

12.19 Scrivete una funzione **depth** che riceva un albero binario e determini di quanti livelli sia composto.

12.20 (Visualizzazione ricorsiva di una lista in ordine inverso) Scrivete una funzione **printListBackwards** che visualizzi in modo ricorsivo e in ordine inverso gli elementi di una lista. Usate la vostra funzione in un programma di prova che crei una lista ordinata di interi e la visualizzi in ordine inverso.

12.21 (Ricerca ricorsiva in una lista) Scrivete una funzione **searchList** che cerchi in modo ricorsivo un dato valore all'interno di una lista concatenata. La funzione dovrà restituire un puntatore al valore, qualora l'abbia ritrovato, o **NULL** in caso contrario. Usate la vostra funzione in un programma di prova che crei una lista di interi. Il programma dovrà richiedere all'utente il valore da individuare nella lista.

12.22 (Rimozione da un albero binario) In questo esercizio, affronteremo la rimozione degli elementi dagli alberi di ricerca binaria. L'algoritmo di rimozione non è così semplice come quello di inserimento. Durante l'eliminazione di un elemento, ci si potrebbe imbattere in tre casi: l'elemento è contenuto in

un nodo foglia (ovverosia, non ha figli), oppure in un nodo che ha un figlio, o in un nodo che ha due figli.

Qualora l'elemento da eliminare sia contenuto in una foglia, questo sarà rimosso e il puntatore del nodo padre sarà impostato a **NULL**.

Qualora l'elemento da eliminare sia contenuto in un nodo che abbia un figlio, il puntatore del padre sarà impostato in modo da fare riferimento al nodo figlio, mentre quello che contiene i dati sarà eliminato. Ciò significa che, all'interno dell'albero, il nodo figlio assumerà il posto di quello rimosso.

L'ultimo caso è il più difficile. Qualora si elmini uno che abbia due figli, un altro nodo dell'albero dovrà assumere il suo posto. Tuttavia, il puntatore del nodo padre non potrà semplicemente essere impostato in modo da fare riferimento a uno dei figli di quello da eliminare. Ciò perché, nella maggior parte dei casi, l'albero di ricerca binaria risultante non si manterebbe fedele alla sua caratteristica principale: *I valori di ogni sottoalbero sinistro dell'albero sono inferiori a quello del nodo padre, mentre quelli di ogni sottoalbero destro dell'albero sono maggiori di quello del nodo padre.*

Quale nodo dovrà essere usato come *nodo sostitutivo* per mantenere questa caratteristica? Potrà essere il nodo contenente il valore più grande dell'albero che sia inferiore a quello del nodo eliminato; oppure potrà essere quello contenente il valore più piccolo dell'albero che sia maggiore di quello del nodo eliminato. Consideriamo il nodo con il valore inferiore. In un albero di ricerca binaria, il valore più grande inferiore a quello di un padre sarà sistemato nel sottoalbero sinistro del nodo padre, e sarà sicuramente contenuto nel nodo all'estremità destra del sottoalbero. Troveremo il suddetto nodo scendendo nel sottoalbero sinistro del padre e andando sempre a destra, fintanto che il puntatore al figlio destro del nodo corrente non sarà **NULL**. A questo punto, staremo puntando al nodo di sostituzione che potrà essere una foglia, o un nodo con un solo figlio alla sua sinistra. Qualora il nodo di sostituzione sia una foglia, i passi per eseguire l'eliminazione saranno i seguenti:

- 1) Salvate il puntatore al nodo da eliminare in una variabile temporanea di tipo puntatore: questo puntatore sarà usato per liberare la memoria allocata dinamicamente.
- 2) Impostate il puntatore del padre del nodo da eliminare in modo che faccia riferimento al nodo di sostituzione.
- 3) Impostate a **NULL** il puntatore al sottoalbero destro nel padre del nodo di sostituzione.
- 4) Impostate il puntatore al sottoalbero destro (sinistro) del nodo di sostituzione in modo che faccia riferimento al sottoalbero destro (sinistro) del nodo da eliminare.
- 5) Eliminate il nodo puntato dalla variabile temporanea di tipo puntatore.

Per eseguire l'eliminazione nel caso di un nodo di sostituzione che abbia un figlio sinistro, si effettueranno delle operazioni simili a quelle effettuate nel caso di un nodo di sostituzione senza figli, ma l'algoritmo dovrà spostare anche il figlio nella posizione attualmente occupata dal nodo di sostituzione all'interno dell'albero. Qualora quello di sostituzione sia un nodo con un figlio a sinistra, i passi per eseguire l'eliminazione saranno i seguenti:

- 1) Salvate il puntatore al nodo da eliminare in una variabile temporanea di tipo puntatore.
- 2) Impostate il puntatore del padre del nodo da eliminare in modo che faccia riferimento al nodo di sostituzione.
- 3) Impostate il puntatore al sottoalbero destro contenuto nel padre del nodo di sostituzione, in modo che faccia riferimento al figlio sinistro del nodo di sostituzione.
- 4) Impostate il puntatore al sottoalbero destro (sinistro) del nodo di sostituzione in modo che faccia riferimento al sottoalbero destro (sinistro) del nodo da eliminare.
- 5) Eliminate il nodo puntato dalla variabile temporanea di tipo puntatore.

Scrivete la funzione **deleteNode** che accetti come suoi argomenti un puntatore al nodo radice dell'albero e il valore da eliminare. La funzione dovrà individuare all'interno dell'albero il nodo contenente il valore da eliminare, e usare gli algoritmi discussi in questo esercizio per rimuovere il nodo.

Qualora il valore non sia stato ritrovato nell'albero, la funzione dovrà visualizzare un messaggio che indichi se il valore sia stato eliminato o no. Modificate il programma della Figura 12.19, in modo che usi questa funzione. Dopo aver eliminato un elemento, richiamate le funzioni di visita **inOrder**, **preOrder** e **postOrder** per confermare che l'operazione di rimozione sia stata eseguita correttamente.

12.23 (Ricerca in un albero binario) Scrivete la funzione **binaryTreeSearch** che tenti di individuare un dato valore all'interno di un albero di ricerca binaria. La funzione dovrà accettare come argomenti un puntatore al nodo radice dell'albero e una chiave di ricerca da individuare. Qualora il nodo contenente la chiave di ricerca sia stato individuato, la funzione dovrà restituire un puntatore a quel nodo; in caso contrario, la funzione dovrà restituire un puntatore **NULL**.

12.24 (Visita per livelli di un albero binario) Il programma della Figura 12.19 ha mostrato tre metodi ricorsivi per visitare un albero binario: la visita simmetrica, quella anticipata e quella differita. Questo esercizio presenterà la visita per livelli di un albero binario, nella quale i valori dei nodi saranno visualizzati livello dopo livello, incominciando da quello del nodo radice. I nodi di ogni livello saranno visualizzati da sinistra a destra. La visita per livelli non è un algoritmo ricorsivo. Essa usa la struttura di dati coda per controllare l'output dei nodi. L'algoritmo sarà il seguente.

- 1) Inserite il nodo radice nella coda.
- 2) Fintanto che siano rimasti dei nodi nella coda,

Prendete il nodo successivo della coda

Visualizzate il valore del nodo

Qualora il puntatore al figlio sinistro del nodo non sia **NULL**

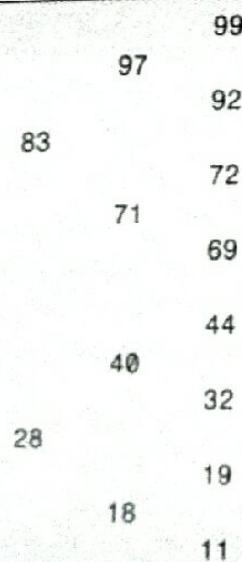
Inserite il figlio sinistro nella coda

Qualora il puntatore al figlio destro del nodo non sia **NULL**

Inserite il figlio destro nella coda

Scrivete la funzione **levelOrder** che esegua una visita per livelli di un albero binario. La funzione dovrà accettare come argomento un puntatore al nodo radice dell'albero. Modificate il programma della Figura 12.19, in modo che utilizzi questa funzione. Confrontate l'output di questa funzione con quelli degli altri algoritmi di visita per verificare che funzioni correttamente. (Nota: in questo programma, avrete anche bisogno di modificare e incorporare le funzioni per l'elaborazione delle code mostrate nella Figura 12.13.)

12.25 (Visualizzare gli alberi) Scrivete una funzione ricorsiva **outputTree** che visualizzi sullo schermo un albero binario. La funzione dovrà visualizzare l'albero una riga per volta, con la cima (la radice) posta alla sinistra dello schermo e la base (le foglie) sistemata a destra dello stesso. Ogni riga sarà visualizzata verticalmente. Per esempio, l'albero binario mostrato nella Figura 12.22 sarà visualizzato come segue:



Osservate che la foglia più a destra compare in cima all'output, nella colonna più a destra, mentre il nodo radice compare a sinistra dell'output. Ogni colonna dell'output incomincia cinque spazi più a destra di quella precedente. La funzione `outputTree` dovrà ricevere come argomenti un puntatore al nodo radice dell'albero e l'intero `totalSpaces`. Questa variabile indicherà il numero di spazi che visualizzato a sinistra dello schermo. Per visualizzare l'albero, la funzione userà una versione modificata della visita simmetrica: essa incomincerà con il nodo più a destra dell'albero e proseguirà tornando indietro a sinistra. L'algoritmo sarà dunque il seguente:

Fintanto che il puntatore al nodo corrente non sia `NULL`

Chiamate in modo ricorsivo `outputTree` con il sottoalbero destro del nodo corrente e `totalSpaces + 5`

Usate una struttura `for` per contare da 1 a `totalSpaces` e visualizzare gli spazi
Visualizzate il valore del nodo corrente

Impostate il puntatore al nodo corrente in modo che faccia riferimento al suo sottoalbero sinistro

Incrementate `totalSpaces` di 5.

Sezione speciale: costruite il vostro compilatore

Negli Esercizi 7.18 e 7.19, abbiamo introdotto il Linguaggio Macchina Simpletron (LMS) e creato il simulatore di computer Simpletron per eseguire i programmi scritti in LMS. In questa sezione, costruiremo un compilatore che convertirà in LMS dei programmi scritti in un linguaggio di programmazione di alto livello. Questa sezione "legherà" insieme l'intero processo di programmazione. Scrivereemo dei programmi nel nuovo linguaggio di alto livello, li compileremo con il compilatore che avremo costruito e li eseguiremo con il simulatore che abbiamo costruito nell'Esercizio 7.19.

12.26 (Il Linguaggio Semplice) Prima di cominciare a costruire il compilatore, discuteremo un semplice ma potente linguaggio di alto livello simile alle prime versioni del popolare linguaggio BASIC. Chiameremo questo linguaggio Semplice. Ogni costrutto eseguibile di Semplice sarà formato da un numero di riga e da un'istruzione di Semplice. I numeri di riga dovranno comparire in ordine crescente. Ogni istruzione incomincerà con uno dei seguenti comandi di Semplice: `rem`, `input`, `let`, `print`, `goto`, `if/goto o end` (consultate la Figura 12.23). Eccettuato `end`, tutti i comandi potranno essere usati ripetutamente. Semplice valuterà soltanto delle espressioni intere usando gli operatori `+`, `-`, `*` e `/`. Questi operatori manterranno la stessa priorità che hanno in C. Potranno comunque essere utilizzate delle parentesi per cambiare l'ordine di valutazione di un'espressione.

Il nostro compilatore di Semplice riconoscerà soltanto le lettere minuscole. Tutti i caratteri inclusi in un file di Semplice dovranno essere in minuscolo (quelli maiuscoli provocheranno un errore di sintassi, sempre che non compaiano in un'istruzione `rem`, nel qual caso saranno ignorati). I *nomi di variabile* saranno formati da una singola lettera. Il Semplice non consentirà nomi di variabile descrittivi, perciò queste dovranno essere descritte nei commenti in modo da indicare il loro uso all'interno del programma. Il Semplice userà soltanto delle variabili intere e non sarà necessario dichiararle: basterà menzionare all'interno del programma un nome di variabile, perché questa sia automaticamente dichiarata e inizializzata a zero. La sintassi di Semplice non consentirà la manipolazione delle stringhe (lettura, scrittura, confronto, ecc.). Il compilatore genererà un errore di sintassi, qualora incontri all'interno di un programma in linguaggio Semplice una stringa preceduta da un comando diverso da `rem`. Il nostro compilatore presumerà che i programmi in linguaggio Semplice siano stati immessi correttamente. L'Esercizio 12.29 chiederà al lettore di modificare il compilatore così che possa eseguire la verifica della sintassi.

Semplice userà l'istruzione condizionale `if/goto` e quella incondizionata `goto` per alterare il flusso di controllo durante l'esecuzione del programma. Qualora l'istruzione condizionata `if/goto` sia vera, il controllo sarà trasferito a una riga specifica del programma. In una istruzione `if/goto` saranno validi

i seguenti operatori relazionali e di uguaglianza: `<`, `>`, `<=`, `>=`, `==` o `!=`. La priorità di questi operatori sarà la stessa che hanno in C.

Comando	Istruzione di esempio	Descrizione
<code>rem</code>	<code>50 rem questo è un commento</code>	Qualsiasi testo successivo al comando <code>rem</code> sarà inserito soltanto a scopo documentativo e sarà ignorato dal compilatore.
<code>input</code>	<code>30 input x</code>	Visualizzerà un punto interrogativo per chiedere all'utente di immettere un intero. Leggerà l'intero dalla tastiera e lo immagazzinerà in <code>x</code> .
<code>let</code>	<code>80 let u = 4 * (j - 56)</code>	Assegnerà a <code>u</code> il valore <code>4 * (j - 56)</code> . Notate che, a destra del segno di uguale, potrebbe comparire un'espressione arbitrariamente complessa.
<code>print</code>	<code>10 print w</code>	Visualizzerà il valore di <code>w</code> .
<code>goto</code>	<code>70 goto 45</code>	Trasferirà il controllo del programma alla riga <code>45</code> .
<code>if/goto</code>	<code>35 if i == z goto 80</code>	Confronterà <code>i</code> e <code>z</code> per verificarne l'uguaglianza e trasferirà il controllo del programma alla riga <code>80</code> , qualora la condizione risulti vera; in caso contrario, continuerà l'esecuzione con l'istruzione successiva.
<code>end</code>	<code>99 end</code>	Terminerà l'esecuzione del programma.

Figura 12.23 I comandi di Semplice.

Prendiamo ora in considerazione alcuni programmi scritti in linguaggio Semplice che dimostreranno le caratteristiche di quest'ultimo. Il primo programma (Figura 12.24) leggerà due interi dalla tastiera, immagazzinerà i loro valori nelle variabili `a` e `b` e calcolerà e visualizzerà la loro somma (immagazzinata nella variabile `c`).

```

10 rem determinare e visualizzare la somma di due interi
15 rem
20 rem prende in input i due interi
30 input a
40 input b
45 rem
50 rem somma gli interi e immagazzina il risultato in c
60 let c = a + b
65 rem
70 rem visualizza il risultato
80 print c
90 rem termina l'esecuzione del programma
99 end

```

Figura 12.24 Determinare la somma di due interi.

Il programma della Figura 12.25 determinerà e visualizzerà il maggiore tra due numeri interi. Gli interi saranno letti dalla tastiera e immagazzinati in `s` e `t`. L'istruzione `if/goto` verificherà la condizione `s >= t`. Qualora la condizione sia vera, il controllo sarà trasferito alla riga `90` e `s` sarà visualizzato; altrimenti, verrà visualizzato `t` e il controllo sarà trasferito all'istruzione `end` della riga `99`, dove il programma terminerà la propria esecuzione.

```

10 rem determinare il maggiore di due interi
20 input s
30 input t
32 rem
35 rem verifica se s >= t
40 if s >= t goto 90
45 rem
50 rem t è maggiore di s, perciò visualizza t
60 print t
70 goto 99
75 rem
80 rem s è maggiore o uguale a t, perciò visualizza s
90 print s
99 end

```

Figura 12.25 Trovare il maggiore di due interi.

Il Semplice non fornisce una struttura di iterazione (come **for**, **while** o **do/while** del C). Tuttavia, può simulare ogni struttura di iterazione del C, usando le istruzioni **if/goto** e **goto**. La Figura 12.26 userà un ciclo controllato da un valore sentinella per calcolare i quadrati di vari interi. Ogni intero sarà preso in input dalla tastiera e immagazzinato nella variabile **j**. Qualora il valore immesso corrisponda a quello di guardia, **-9999**, il controllo sarà trasferito alla riga **99** dove l'esecuzione del programma terminerà. In caso contrario, il quadrato di **j** sarà assegnato a **k**, questa sarà visualizzata sullo schermo e il controllo sarà passato alla riga **20**, dove sarà preso in input l'intero successivo.

Usando come vostra guida i programmi di esempio delle Figure 12.24, 12.25 e 12.26, scrivete un programma in linguaggio Semplice che esegua ognuna delle seguenti attività:

- Prendete in input tre interi, determinatene la media e visualizzate il risultato.
- Usate un ciclo controllato da un valore sentinella per prendere in input 10 interi e calcolare e visualizzare la loro somma.
- Usate un ciclo controllato da un contatore per prendere in input 7 interi, alcuni positivi e altri negativi, e calcolare e visualizzare la loro media.
- Prendete in input una serie di interi e determinatene e visualizzatene il maggiore. Il primo intero in input indicherà quanti numeri dovranno essere elaborati.
- Prendete in input 10 interi e visualizzatene il minore.
- Calcolate e visualizzate la somma degli interi pari compresi tra 2 e 30.
- Calcolate e visualizzate il prodotto degli interi dispari compresi tra 1 e 9.

12.27 (Costruire un compilatore; Prerequisiti: completate gli Esercizi 7.18, 7.19, 12.12, 12.13 e 12.26) Ora che il linguaggio Semplice è stato presentato (Esercizio 12.26), discuteremo di come costruire il nostro compilatore di Semplice. In primo luogo, considereremo il processo attraverso il quale un programma in linguaggio Semplice sarà convertito in LMS ed eseguito dal simulatore Simpletron (consultate la Figura 12.27). Un file contenente un programma in linguaggio Semplice sarà letto dal compilatore e convertito in codice LMS. Questo sarà salvato in un file su disco, nel quale le istruzioni LMS compariranno una per riga. Il file LMS sarà quindi caricato nel simulatore Simpletron e i risultati saranno inviati a un file su disco e allo schermo. Osservate che il programma Simpletron sviluppato nell'Esercizio 7.19 prendeva il suo input dalla tastiera. Esso dovrà quindi essere modificato per leggere l'input da un file, così che possa eseguire i programmi prodotti dal nostro compilatore.

Per convertirlo in LMS, il compilatore effettuerà due *passaggi* sul programma in linguaggio Semplice. Il primo passaggio costruirà la *tabella dei simboli* (la tabella dei simboli sarà discussa più tardi in dettaglio) nella quale saranno immagazzinati, con i loro tipi di dato e le rispettive posizioni all'interno del codice LMS finale, ogni *numero di riga*, *nome di variabile* e *costante* del programma in linguaggio

Semplice. Il primo passo produrrà anche le istruzioni LMS corrispondenti a ogni espressione di Semplice. Come vedremo, il primo passaggio produrrà un codice LMS che conterrà alcune istruzioni incomplete, qualora il programma in linguaggio Semplice contenga delle istruzioni che trasferiscano il controllo a una riga successiva del codice. Il secondo passaggio del compilatore individuerà e completerà le istruzioni incomplete e salverà il programma LMS in un file.

```

10 rem  calcolare i quadrati di vari interi
20 input j
23 rem
25 rem  controlla il valore sentinella
30 if j == -9999 goto 99
33 rem
35 rem  calcola il quadrato di j e assegna il risultato a k
40 let k = j * j
50 print k
53 rem
55 rem  itera per prendere il prossimo j
60 goto 20
99 end

```

Figura 12.26 Calcolare il quadrato di vari interi.

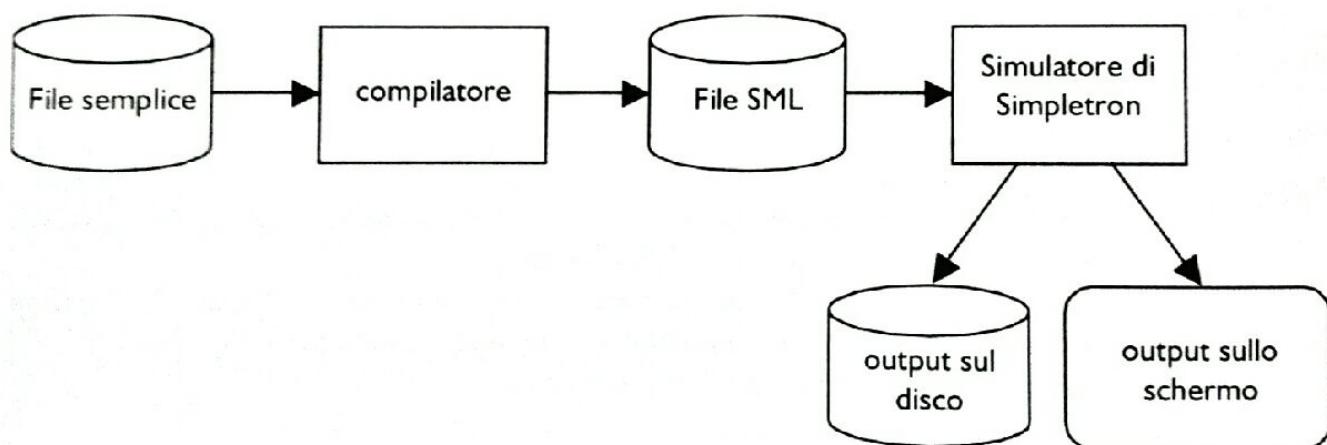


Figura 12.27 Scrivere, compilare ed eseguire un programma in linguaggio Semplice.

Primo passaggio

Il compilatore inizierà leggendo nella memoria un'istruzione del programma in linguaggio Semplice. Per l'elaborazione e la compilazione, la riga dovrà essere separata nei suoi singoli *token* (ovverosia, nei "pezzi" di un'istruzione) e, per facilitare questo compito, potrà essere usata la funzione **strtok** della libreria standard. Ricordate che ogni istruzione incomincia con un numero di riga seguito da un comando. Man mano che il compilatore scomporrà un'istruzione nei suoi token, questi saranno sistemati nella tabella dei simboli, qualora siano dei numeri di riga, delle variabili o delle costanti. Un numero di riga sarà sistemato nella tabella dei simboli soltanto qualora sia il primo token di un'istruzione. La **symbolTable** è un vettore di strutture **tableEntry** che rappresenterà ogni simbolo del programma e, di conseguenza, potrà essere particolarmente corposa per certi programmi. Per ora, create la **symbolTable** come un vettore di 100 elementi. Potrete incrementare o ridurre le sue dimensioni, una volta che il programma sarà funzionante.

La definizione della struttura **tableEntry** è la seguente:

```
struct tableEntry {
    int symbol;
    char type;      /* 'C', 'L', o 'V' */
    int location;   /* da 00 a 99 */
}
```

Ogni struttura **tableEntry** contiene tre membri. Il membro **symbol** è un intero che conterrà la rappresentazione ASCII di una variabile (ricordate che i nomi di variabile sono caratteri singoli), un numero di riga o una costante. Il membro **type** corrisponderà a un carattere che indicherà il tipo del simbolo: 'C' per una costante, 'L' per un numero di riga o 'V' per una variabile. Il membro **location** conterrà la posizione (da 00 a 99) puntata dal simbolo nella memoria del Simpletron. Questa è un vettore di 100 elementi interi in cui saranno immagazzinati i dati e le istruzioni LMS. Per un numero di riga, la posizione rappresenterà l'elemento nel vettore della memoria del Simpletron in cui incominceranno le istruzioni LMS corrispondenti a quella in linguaggio Semplice. Per una variabile o una costante, la posizione rappresenterà l'elemento nel vettore della memoria del Simpletron in cui sarà immagazzinata la variabile, o la costante. Queste saranno allocate cominciando dalla fine della memoria del Simpletron. La prima variabile o costante sarà immagazzinata nella posizione 99, la successiva nella 98, ecc.

La tabella dei simboli gioca una parte integrante nella conversione in LMS dei programmi scritti in linguaggio Semplice. Nel Capitolo 7 abbiamo appreso che un'istruzione LMS è un intero di quattro cifre, formato da due parti: il *codice dell'operazione* e l'*operando*. Il codice dell'operazione è determinato dai comandi del linguaggio Semplice. Per esempio, il comando di Semplice **input** corrisponde al codice dell'operazione LMS 10 (read), mentre il comando di Semplice **print** corrisponde al codice dell'operazione LMS 11 (write). L'operando è una posizione di memoria contenente i dati su cui il codice dell'operazione dovrà eseguire il proprio compito (il codice dell'operazione 10, per esempio, leggerà un valore dalla tastiera e lo immagazzinerà nella posizione di memoria specificata dall'operando). Per ogni simbolo, il compilatore effettuerà una ricerca nella **symbolTable**, per determinarne la posizione all'interno della memoria del Simpletron, così che la stessa possa essere utilizzata per completare le istruzioni LMS.

La compilazione di ogni istruzione di Semplice è basata sui suoi comandi. Per esempio, una volta che il numero di riga di un **rem** sarà stato inserito nella tabella dei simboli, la parte rimanente dell'istruzione sarà ignorata dal compilatore, poiché un commento ha solo scopi documentativi. Le istruzioni **input**, **print**, **goto** ed **end** corrisponderanno a quelle LMS *read*, *write*, *branch* (a una posizione specificata) e *halt*. Le istruzioni contenenti questi comandi di Semplice saranno convertite direttamente in LMS (notate che un **goto** potrebbe contenere un riferimento irrisolto, qualora il numero di riga punti a un'istruzione successiva nel file del programma in linguaggio Semplice; ovverosia il cosiddetto riferimento in avanti).

Un **goto** con un riferimento irrisolto sarà compilato in un'istruzione LMS che dovrà essere *contrassegnata*, per indicare che il secondo passaggio del compilatore dovrà completarla. Le segnalazioni (o flag) saranno immagazzinate nei 100 elementi del vettore **flags** di tipo **int**, che saranno inizializzati con il valore -1. Nel caso in cui la posizione di memoria puntata da un numero di riga del programma in linguaggio Semplice non sia ancora nota (ovverosia, non sia stata ritrovata nella tabella dei simboli), allora quel numero di riga dovrà essere immagazzinato nell'elemento del vettore **flags** avente lo stesso indice dell'istruzione incompleta. L'operando dell'istruzione incompleta sarà impostato temporaneamente a 00. Per esempio, un'istruzione di salto incondizionato (che abbia un riferimento in avanti) sarà lasciata come +4000 fino al secondo passaggio del compilatore (che descriveremo tra breve).

La compilazione delle istruzioni **if/goto** e **let** è più complicata di quella delle altre: infatti, queste sono le uniche che producono più di un'istruzione LMS. Per un'istruzione **if/goto**, il compilatore produrrà un codice che verificherà la condizione e, se sarà necessario, salterà a un'altra riga. Il risultato del salto potrà anche essere un riferimento irrisolto. Ognuno degli operatori relazionali e di uguaglianza

za potrà essere simulato usando le istruzioni LMS *branch zero* e *branch negative*, o eventualmente una combinazione di entrambe.

Per un'istruzione **let**, il compilatore produrrà un codice che valuterà un'espressione aritmetica arbitrariamente complessa formata da variabili intere e/o costanti. Le espressioni dovranno separare ogni operando e operatore con degli spazi. Gli Esercizi 12.12 e 12.13 hanno presentato l'algoritmo di conversione dalla notazione infissa a quella polacca inversa, e l'algoritmo di valutazione usato dai compilatori per valutare le espressioni in notazione polacca inversa. Prima di procedere con il vostro compilatore, dovrete completare ognuno dei suddetti esercizi. Quando un compilatore incontrerà un'espressione, la convertirà dalla notazione infissa a quella polacca inversa e quindi la valuterà.

Ma in che modo il compilatore produrrà il codice per valutare un'espressione che contenga delle variabili? L'algoritmo di valutazione delle espressioni in notazione polacca inversa prevede una variante, che consentirà al nostro compilatore di generare le istruzioni LMS, invece di valutare effettivamente le espressioni. Per attivare la suddetta variante nel compilatore, l'algoritmo di valutazione delle espressioni in notazione polacca inversa dovrà essere modificato in modo che possa ricercare (ed eventualmente inserire) nella **symbolTable** ogni simbolo che incontrerà. Per ognuno di quei simboli, l'algoritmo dovrà quindi determinare e inserire nella pila la corrispondente posizione di memoria, invece del simbolo. Nel momento in cui verrà incontrato un operatore all'interno dell'espressione in notazione polacca inversa, saranno estratte le due posizioni di memoria in cima alla pila e sarà prodotto il codice in linguaggio macchina necessario per eseguire l'operazione, usando come operandi le posizioni di memoria. Il risultato di ogni componente dell'espressione sarà immagazzinato in una posizione di memoria temporanea e inserito nuovamente nella pila, perché possa proseguire la valutazione dell'espressione in notazione polacca inversa. Nel momento in cui la valutazione sarà stata completata, la posizione di memoria contenente il risultato sarà l'unica posizione rimasta sulla pila. Questa sarà dunque estratta e saranno generate le istruzioni LMS per assegnare il risultato alla variabile a sinistra dell'istruzione **let**.

Secondo passaggio

Il secondo passaggio del compilatore eseguirà due attività: risolvere ogni riferimento irrisolto e salvare il codice LMS in un file. La risoluzione dei riferimenti avverrà nel modo seguente:

- 1) Ricercate nel vettore **flags** un riferimento irrisolto (ovverosia, un elemento con un valore diverso da -1).
- 2) Individuate nel vettore **symbolTable** la struttura contenente il simbolo immagazzinato in **flags** (assicuratevi che il tipo del simbolo sia 'L', per i numeri di riga).
- 3) Inserite la posizione di memoria indicata dal membro della struttura **location** nell'istruzione con il riferimento irrisolto. Ricordatevi che un'istruzione contenente un riferimento irrisolto ha l'operando **00**.
- 4) Ripetete i passi 1, 2 e 3 finché non sarà stata raggiunta la fine del vettore **flags**.

Una volta che il processo di risoluzione sarà stato completato, l'intero vettore del codice LMS sarà salvato in un file su disco, sistemandone un'istruzione LMS per riga. Questo file potrà essere letto dal Simpletron per l'esecuzione, una volta che avrete modificato il simulatore in modo che possa leggere il suo input da un file.

Un esempio completo

L'esempio seguente illustrerà una conversione completa in LMS di un programma scritto in linguaggio Semplice, così come sarà eseguita dal compilatore di Semplice. Considerate un programma scritto in linguaggio Semplice, che prenda in input un intero e sommi i valori compresi nell'intervallo da 1 all'intero specificato. Nella Figura 12.28 sono mostrati il programma e le istruzioni LMS generate dal primo passaggio. Nella Figura 12.29 è mostrata la tabella dei simboli costruita dal primo passaggio.

La maggior parte delle istruzioni di Semplice potrà essere convertita direttamente in singole istruzioni LMS. In questo programma, le eccezioni saranno rappresentate dai commenti, dal-gio macchina. Tuttavia, i numeri di riga di un commento saranno ugualmente sistemati nella tabella dei simboli, nell'eventualità che siano stati puntati da un **goto** o da un **if/goto**. La riga **20** specifica che il controllo del programma passerà alla riga **60** qualora la condizione **y == x** sia vera. Dato che la riga **60** comparirà solo più tardi nel programma, il primo passaggio del compilatore non lo ha ancora sistemato nella tabella dei simboli (i numeri di riga saranno sistemati nella tabella dei simboli, solo qualora compaiano come primo token di un'istruzione). Di conseguenza, a questo punto, non è possibile determinare l'operando dell'istruzione LMS **branch zero**, nella posizione **03** del vettore che contiene le istruzioni LMS. Il compilatore sistemerà dunque il **60** nella posizione **03** del vettore **flags** per indicare che il secondo passaggio dovrà completare questa istruzione.

Programma in Semplice	Posizione e istruzione LMS	Descrizione
5 rem aggiunge 1 a x	<i>nessuna</i>	rem ignorata
10 input x	00 +1099	legge x nella posizione 99
15 rem verifica se y == x	<i>nessuna</i>	rem ignorata
20 if y == x goto 60	01 +2098 02 +3199 03 +4200	carica y (98) nell'accumulatore sottrae x (99) dall'accumulatore <i>branch zero</i> a una posizione irrisolta
25 rem incrementa y	<i>nessuna</i>	rem ignorata
30 let y = y + 1	04 +2098 05 +3097 06 +2196 07 +2096 08 +2198	carica y nell'accumulatore aggiunge 1 (97) all'accumulatore lo immagazzina nella posizione temporanea 96 carica dalla posizione temporanea 96 immagazzina l'accumulatore in y
35 rem aggiunge y al totale	<i>nessuna</i>	rem ignorata
40 let t = t + y	09 +2095 10 +3098 11 +2194	carica t (95) nell'accumulatore aggiunge y all'accumulatore immagazzina nella posizione temporanea
94	12 +2094 13 +2195	carica dalla posizione temporanea 94 immagazzina l'accumulatore in t
45 rem ciclo y	<i>nessuna</i>	rem ignorata
50 goto 20	14 +4001	salta alla posizione 01
55 rem visualizza il risultato	<i>nessuna</i>	rem ignorata
60 print t	15 +1195	visualizza t sullo schermo
99 end	16 +4300	termina l'esecuzione

Figura 12.28 Le istruzioni LMS generate dopo il primo passaggio del compilatore.

Dovremo mantenere traccia della posizione della prossima istruzione nel vettore LMS, perché non c'è una corrispondenza uno a uno tra le istruzioni di Semplice e quelle di LMS. Per esempio, l'istruzione **if/goto** della riga **20** sarà compilata in tre istruzioni LMS. Di conseguenza, ogni volta che sarà generata un'istruzione dovremo incrementare il *contatore di istruzioni*, in

modo che punti alla posizione successiva del vettore LMS. Osservate che la dimensione della memoria del Simpletron potrebbe rappresentare un problema per i programmi in linguaggio Semplice formati da molte istruzioni, variabili e costanti. Infatti, è probabile che il compilatore esaurisca la memoria. Per controllare questa eventualità, il vostro programma dovrà contenere un *contatore di dati*, in modo da conservare la posizione del vettore LMS nella quale sarà immagazzinata la prossima variabile o costante. Il vettore LMS sarà pieno quando il valore del contatore di istruzioni sarà superiore a quello del contatore di dati. In questo caso, il processo di compilazione dovrà essere interrotto e il compilatore dovrà visualizzare un messaggio di errore, per indicare che avrà esaurito la memoria durante la compilazione.

Simbolo	Tipo	Posizione
5	L	00
10	L	00
'x'	V	99
15	L	01
20	L	01
'y'	V	98
25	L	04
30	L	04
1	C	97
35	L	09
40	L	09
't'	V	95
45	L	14
50	.L	14
55	L	15
60	L	15
99	L	16

5	L	00
10	L	00
'x'	V	99
15	L	01
20	L	01
'y'	V	98
25	L	04
30	L	04
1	C	97
35	L	09
40	L	09
't'	V	95
45	L	14
50	.L	14
55	L	15
60	L	15
99	L	16

Figura 12.29 Tabella dei simboli per il programma della Figura 12.28

Un esame passo per passo del processo di compilazione

Osserviamo ora il processo di compilazione per il programma in linguaggio Semplice mostrato nella Figura 12.28. Il compilatore legge in memoria la prima riga del programma

5 rem aggiunge 1 a x

Il primo simbolo dell'istruzione, il numero di riga, sarà determinato usando **strtok** (consultate il Capitolo 8 per una discussione sulle funzioni C di manipolazione delle stringhe). Il simbolo restituito da **strtok** sarà convertito in un intero usando **atoi**, perciò nella tabella dei simboli potrà essere ricercato il token **5** che sarà inserito nella suddetta tabella, qualora non sia stato ritrovato. Dato che siamo all'inizio del programma e che questa è la prima riga, nella tabella non c'è ancora nessun simbolo. Di conseguenza, il **5** vi sarà inserito come tipo **L** (numero di riga) e sarà assegnato alla prima posizione (**00**) del vettore LMS. Nonostante si tratti di un commento, allocheremo ugualmente uno spazio per il numero di riga all'interno della tabella dei simboli, nell'eventualità che sia stato puntato da un **goto** o un **if/goto**. Per la **rem** non sarà generata nessuna istruzione LMS e quindi il relativo contatore non sarà incrementato.

In seguito, sarà analizzata l'istruzione

10 input x

Il numero di riga **10** sarà sistemato nella tabella dei simboli, come tipo **L**, e sarà assegnato alla prima posizione del vettore LMS (**00**, poiché il programma comincia con un commento e quindi il contatore di istruzioni varrà ancora **00**). Il comando **input** indica che il prossimo simbolo sarà una variabile, poiché in un'istruzione **input** può comparire solo una variabile. Il compilatore dovrà semplicemente determinare la posizione di **x** nel vettore LMS, poiché **input** corrisponde direttamente a un codice di operazione LMS. Il simbolo **x** non sarà ancora presente nella tabella dei simboli e quindi vi dovrà essere inserita la sua rappresentazione **ASCII**, con il tipo **V**, e gli sarà assegnata la posizione **99** del vettore LMS, perché l'immagazzinamento dei dati incomincia dalla posizione **99** e prosegue dall'alto verso il basso. A questo punto potrà essere generato il codice LMS per questa istruzione. Il codice di operazione **10** (ovverosia quello LMS per la lettura) sarà moltiplicato per **100** e la posizione di **x**, determinata dalla tabella dei simboli, vi sarà aggiunta per completare l'istruzione. L'istruzione sarà quindi immagazzinata nella posizione **00** del vettore LMS. Il contatore di istruzioni sarà incrementato di 1 poiché sarà stata generata una sola istruzione LMS.

In seguito sarà analizzata l'istruzione

15 rem verifica se y == x

Il numero di riga **15** sarà ricercato e non trovato nella tabella dei simboli, dove sarà inserito con il tipo **L**, e sarà assegnato alla posizione successiva del vettore, **01**. Ricordate che l'istruzione **rem** non genera codice, perciò il contatore di istruzioni non sarà incrementato.

In seguito sarà analizzata l'istruzione

20 if y == x goto 60

Il numero di riga **20** sarà inserito nella tabella dei simboli e gli sarà assegnato il tipo **L** e la posizione successiva del vettore LMS: **01**. Il comando **if** indica che dovrà essere valutata una condizione. La variabile **y** non sarà ancora presente nella tabella dei simboli, perciò vi sarà inserita e gli sarà assegnato il tipo **V** e la posizione **98** di LMS. In seguito, saranno generate le istruzioni LMS che valuteranno la condizione. Dato che in LMS non c'è nessuna diretta corrispondenza per il costrutto **if/goto**, questo dovrà essere simulato eseguendo un calcolo con **x** e **y** e trasferendo il controllo del programma in base al risultato. Il risultato della sottrazione di **x** da **y** sarà zero, qualora **y** sia uguale a **x**, e perciò potrà essere usata l'istruzione **branch zero** con il risultato del calcolo per simulare l'istruzione **if/goto**. Il primo passo richiederà che **y** sia caricata nell'accumulatore dalla posizione **99** di LMS, generando quindi l'istruzione **01 +2098**. In seguito, **x** sarà sottratta dall'accumulatore, generando l'istruzione **02 +3199**.

A questo punto, il valore contenuto nell'accumulatore potrà essere zero, positivo o negativo. Dato che l'operatore è **==**, dovremo usare l'istruzione LMS **branch zero**. In primo luogo, cercheremo la posizione di destinazione per il salto (**60**, in questo caso) nella tabella dei simboli, ma non la troveremo. Di conseguenza, il **60** dovrà essere immesso nella posizione **03** del vettore **flags** e dovrà essere generata l'istruzione **03 +4200**. Infatti, non potremo aggiungere la destinazione del salto, perché non avremo ancora assegnato una posizione nel vettore LMS alla riga **60**. Il contatore di istruzioni sarà incrementato a **04**.

Il compilatore procederà con l'istruzione

25 rem incrementa y

Il numero di riga **25** sarà inserito nella tabella dei simboli con il tipo **L** e con la posizione **04** di LMS. Il contatore di istruzioni non sarà incrementato.

Nel momento in cui l'istruzione

30 let y = y +1

sarà analizzata, il numero di riga **30** sarà inserito nella tabella dei simboli con il tipo **L** e con la posizione **04** di LMS. Il comando **let** indica che la riga è un'istruzione di assegnamento e, di conseguenza, tutti i simboli della riga dovranno essere inseriti nella tabella dei simboli, qualora non vi siano già stati inclusi. L'intero **1** sarà aggiunto alla tabella dei simboli con il tipo **C** e con la posizione **97** di LMS. La parte destra dell'assegnamento sarà quindi convertita dalla notazione infissa a quella polacca inversa. In

seguito, sarà valutata l'espressione in notazione polacca inversa (**y** **1** **+**). Il token **y** sarà individuato nella tabella dei simboli e la sua posizione di memoria sarà sistemata in cima alla pila. Anche il token **1** sarà individuato nella tabella dei simboli e la sua posizione di memoria sarà sistemata in cima alla pila. Una volta che avrà incontrato l'operatore **+**, la funzione di valutazione delle espressioni in notazione polacca inversa estrarrà due valori consecutivi dalla pila, assegnandoli rispettivamente agli operandi di destra e di sinistra dell'operatore e generando le istruzioni LMS

04 +2098 (*load y*)
05 +3097 (*add 1*)

Il risultato dell'espressione sarà immagazzinato in una posizione temporanea della memoria (**96**), con l'istruzione

06 +2196 (*store temporaneo*)

e la posizione temporanea sarà sistemata sulla pila. Una volta che l'espressione sarà stata valutata, il risultato dovrà essere immagazzinato in **y** (ovverosia, nella variabile a sinistra di **=**). Di conseguenza, la posizione temporanea sarà caricata nell'accumulatore e questo sarà immagazzinato in **y**, con le istruzioni

07 +2096 (*load temporaneo*)
08 +2198 (*store y*)

Vi sarete accorti immediatamente che le istruzioni LMS sembrano essere ridondanti. Discuteremo di questo tra breve.

Nel momento in cui l'istruzione

35 rem aggiunge y al totale

sarà analizzata, il numero di riga **35** sarà inserito nella tabella dei simboli con il tipo **L** e con la posizione **09**.

L'istruzione

40 let t = t + y

è simile alla riga **30**. La variabile **t** sarà dunque inserita nella tabella dei simboli con il tipo **V** e con la posizione **95** di LMS. Le istruzioni seguono la stessa logica e formato della riga **30**, perciò saranno generate le istruzioni **09 +2095**, **10 +3098**, **11 +2194**, **12 +2094**, e **13 +2195**. Osservate che il risultato di **t + y** sarà assegnato alla posizione temporanea **94**, prima di essere assegnato a **t** (**95**). Ancora una volta, avrete notato che le istruzioni inserite nelle posizioni di memoria **11** e **12** sembrano essere ridondanti. Di nuovo, discuteremo di ciò tra breve.

L'istruzione

45 rem ciclo y

è un commento, perciò la riga **45** sarà aggiunta alla tabella dei simboli con il tipo **L** e con la posizione **14** di LMS.

L'istruzione

50 goto 20

trasferisce il controllo alla riga **20**. In LMS l'equivalente di **goto** è l'istruzione di *salto incondizionato* (**40**) che trasferisce il controllo a una specifica posizione di LMS. Il compilatore cercherà dunque nella tabella dei simboli la riga **20** e rileverà che corrisponderà alla posizione **01** di LMS. Il codice dell'operazione (**40**) sarà moltiplicato per 100 e aggiunto alla posizione **01** così da generare l'istruzione **14 +4001**.

L'istruzione

55 rem visualizza il risultato

è un commento e, di conseguenza, la riga **55** sarà inserita nella tabella dei simboli con il tipo **L** e con la posizione **15**.

L'istruzione

```
60 print t
```

è un'istruzione di output. Il numero di riga **60** sarà inserito nella tabella dei simboli con il tipo **L** e con la posizione **15** di LMS. In LMS l'equivalente di **print** è il codice di operazione **11 (write)**. La posizione di **t** sarà rilevata dalla tabella dei simboli e sarà aggiunta al prodotto tra 100 e il codice dell'operazione.

L'istruzione

```
99 end
```

è la riga finale del programma. Il numero di riga **99** sarà immagazzinato nella tabella dei simboli con il tipo **L** e con la posizione **16** di LMS. Il comando **end** genererà l'istruzione **+4300** (**43** è *halt* in LMS), che sarà scritta come istruzione finale nel vettore della memoria LMS.

Questo completa il primo passaggio del compilatore. Considereremo ora il secondo passaggio. Cercheremo nel vettore **flags** i valori diversi da -1. La posizione **03** contiene **60**, perciò il compilatore saprà che l'istruzione **03** non è completa. Esso integrerà quindi l'istruzione ricercando il **60** nella tabella dei simboli, determinandone la sua posizione e aggiungendola a quella incompleta. In questo caso, la ricerca determinerà che la riga **60** corrisponde alla posizione **15** di LMS e, di conseguenza, sarà generata l'istruzione completa **03 +4215** in sostituzione di **03 +4200**. A questo punto il programma scritto in Semplice sarà stato compilato con successo.

Per costruire il compilatore, dovete eseguire ognuna delle seguenti attività:

- Modificate il simulatore di Simpletron che avete scritto nell'Esercizio 7.19, in modo che prenda il suo input da un file specificato dall'utente (consultate il Capitolo 11). Il compilatore dovrà anche salvare i suoi risultati in un file su disco, seguendo lo stesso formato dell'output sullo schermo.
- Modificate l'algoritmo di conversione dalla notazione infissa a quella polacca inversa dell'Esercizio 12.12, in modo che elabori operandi interi con più cifre e operandi corrispondenti a nomi di variabile formati da una sola lettera. Consiglio: per individuare tutte le costanti e le variabili di un'espressione, potrà essere usata la funzione **strtok** della libreria standard, mentre le costanti di stringa potranno essere convertite in interi, usando la funzione **atoi** della libreria standard. Nota: la rappresentazione dei dati dell'espressione in notazione polacca inversa dovrà essere alterata in modo da supportare i nomi di variabile e le costanti intere.
- Modificate l'algoritmo di valutazione delle espressioni in notazione polacca inversa, in modo che elabori operandi interi con più cifre e operandi corrispondenti a nomi di variabile. L'algoritmo dovrà anche implementare la variante discussa in precedenza, in modo che siano prodotte le istruzioni LMS, invece di valutare direttamente l'espressione. Consiglio: per individuare tutte le costanti e le variabili di un'espressione, potrà essere usata la funzione **strtok** della libreria standard, mentre le costanti di stringa potranno essere convertite in interi usando la funzione **atoi** della libreria standard. Nota: la rappresentazione dei dati dell'espressione in notazione polacca inversa dovrà essere alterata in modo da supportare i nomi di variabile e le costanti intere.
- Costruite il compilatore. Incorporate le parti (b) e (c) che valutino le espressioni delle istruzioni **let**. Il vostro programma dovrà contenere due funzioni che eseguano il primo e il secondo passo del compilatore. Entrambe le funzioni potranno richiamarne altre per svolgere i loro compiti.

12.28 (Ottimizzare il compilatore di Semplice) Quando un programma sarà compilato e convertito in LMS, sarà generato un insieme di istruzioni. Certe combinazioni di istruzioni si ripeteranno spesso, di solito in triplette dette produzioni. Queste saranno normalmente formate da tre istruzioni come **load**, **add** e **store**. Per esempio, la Figura 12.30 mostra cinque delle istruzioni LMS che sono state generate durante la compilazione del programma nella Figura 12.28. Le prime tre istruzioni corri-

spondono alla produzione che aggiunge 1 a y. Osservate che le istruzioni **06** e **07** immagazzinano il valore dell'accumulatore nella posizione temporanea **96** e poi lo ricaricano nuovamente nell'accumulatore, in modo che l'istruzione **08** possa immagazzinarlo nella posizione **98**. Spesso una produzione è seguita da un'istruzione di caricamento dalla stessa posizione che è stata appena usata per l'immagazzinamento. Questo codice potrà dunque essere ottimizzato, eliminando l'istruzione di immagazzinamento e quella susseguente di caricamento che operano sulla stessa posizione di memoria. L'ottimizzazione consentirà al Simpletron di eseguire più velocemente il programma, poiché in questa versione ci saranno meno istruzioni. La Figura 12.31 mostra il codice LMS ottimizzato per il programma della Figura 12.28. Osservate che nel codice ottimizzato ci sono quattro istruzioni in meno: un risparmio di memoria del 25%.

```

04 +2098 (load)
05 +3097 (add)
06 +2196 (store)

07 +2096 (load)
08 +2198 (store)

```

Figura 12.30 Codice non ottimizzato tratto dal programma in Figura 12.28.

Modificate il compilatore in modo da offrire un'opzione di ottimizzazione del codice generato in Linguaggio Macchina Simpletron. Confrontate manualmente il codice non ottimizzato e quello ottimizzato e calcolate la percentuale di riduzione.

Programma in Semplice	Posizione e istruzione in LMS	Descrizione
5 rem aggiunge 1 a x	nessuna	rem ignorata
10 input x	00 +1099	legge x nella posizione 99
15 rem verifica se y == x	nessuna	rem ignorata
20 if y == x goto 60	01 +2098 02 +3199 03 +4211	carica y (98) nell'accumulatore sottrae x (99) dall'accumulatore se zero, salta alla posizione 11
25 rem incrementa y	nessuna	rem ignorata
30 let y = y + 1	04 +2098 05 +3097 06 +2198	carica y nell'accumulatore aggiunge 1 (97) all'accumulatore immagazzina l'accumulatore in y (98)
35 rem aggiunge y al totale	nessuna	rem ignorata
40 let t = t + y	07 +2096 08 +3098 09 +2196	carica t dalla locazione (96) aggiunge y (98) all'accumulatore immagazzina l'accumulatore in t (96)
45 rem ciclo y	nessuna	rem ignorata
50 goto 20	10 +4001	salta alla posizione 01
55 rem visualizza il risultato	nessuna	rem ignorata
60 print t	11 +1196	visualizza t (96) sullo schermo
99 end	12 +4300	termina l'esecuzione

Figura 12.31 Il codice ottimizzato per il programma della Figura 12.28.

12.29 (Modifiche al compilatore di Semplice) Apportate le seguenti modifiche al compilatore di Semplice. Alcune di esse potrebbero anche richiedere delle variazioni al Simulatore Simpletron scritto nell'Esercizio 7.19.

- Consentite l'uso dell'operatore modulo (%) nelle istruzioni **let**. Il Linguaggio Macchina Simpletron dovrà essere modificato per includere l'istruzione modulo.
- Consentite l'elevamento a potenza nelle istruzioni **let**, usando il simbolo ^ come operatore di elevamento a potenza. Il Linguaggio Macchina Simpletron dovrà essere modificato per includere l'istruzione di elevamento a potenza.
- Consentite al compilatore di riconoscere le lettere maiuscole e minuscole nelle istruzioni di Semplice (per esempio, 'A' è equivalente ad 'a'). Non saranno necessarie modifiche al Simulatore Simpletron.
- Consentite all'istruzione **input** di leggere valori per variabili multiple come **input x, y**. Non saranno necessarie modifiche al Simulatore Simpletron.
- Consentite al compilatore di visualizzare valori multipli in una singola istruzione **print** come **print a, b, c**. Non saranno necessarie modifiche al Simulatore Simpletron.
- Aggiungete al compilatore la capacità di verificare la sintassi, in modo che siano visualizzati dei messaggi, qualora fossero riscontrati degli errori di sintassi all'interno di un programma in linguaggio Semplice. Non saranno necessarie modifiche al Simulatore Simpletron.
- Implementate i vettori di interi. Non saranno necessarie modifiche al Simulatore Simpletron.
- Implementate le subroutine specificate dai comandi di Semplice **gosub** e **return**. Il comando **gosub** passerà il controllo del programma a una subroutine, mentre il comando **return** lo restituirà all'istruzione successiva alla **gosub**. Questo meccanismo è simile alla chiamata di funzione del C. Una stessa subroutine potrà essere richiamata da molte **gosub** distribuite in tutto il programma. Non saranno richieste modifiche al Simulatore Simpletron.
- Implementate le strutture di iterazione secondo il formato

```
for x = 2 to 10 step 2
    istruzioni in linguaggio Semplice
next
```

L'istruzione **for** itera da **2** a **10** con un incremento di **2**. La riga **next** indica la fine del corpo del **for**. Non saranno richieste modifiche al Simulatore Simpletron.

- Implementate le strutture di iterazione secondo il formato

```
for x = 2 to 10
    istruzioni in linguaggio Semplice
next
```

L'istruzione **for** itera da **2** a **10** con un incremento predefinito di **1**. Non saranno richieste modifiche al Simulatore Simpletron.

- Consentite al compilatore di elaborare l'input e l'output delle stringhe. Ciò richiederà la modifica del Simulatore Simpletron per consentirgli di elaborare e immagazzinare i valori di tipo stringa. Consiglio: ogni parola del Simpletron potrà essere divisa in due gruppi che contengano un intero di due cifre. Ogni intero di due cifre rappresenterà il valore decimale ASCII equivalente a un carattere. Aggiungete un'istruzione in linguaggio macchina che possa visualizzare una stringa cominciando da una certa posizione della memoria del Simpletron. La prima metà della parola in quella posizione corrisponderà al numero di caratteri presenti nella stringa (ovverosia, alla sua lunghezza). Ogni successiva mezza parola conterrà un carattere ASCII espresso con due cifre decimali. L'istruzione in linguaggio macchina controllerà la lunghezza e visualizzerà la stringa, traducendo ogni numero di due cifre nel carattere corrispondente.

- m) Consentite al compilatore di elaborare i valori in virgola mobile, oltre a quelli interi. Anche il Simulatore Simpletron dovrà essere modificato per consentirgli di elaborare i valori in virgola mobile.

12.30 (Un interprete di Semplice) Un interprete è un programma che legge un'istruzione di un codice scritto in un linguaggio di alto livello, determina l'operazione da eseguire per quell'istruzione e la esegue immediatamente. Dunque il programma non sarà convertito prima in linguaggio macchina. Gli interpreti lavorano lentamente, perché ogni istruzione incontrata nel programma dovrà prima essere decifrata. Nel caso in cui le istruzioni siano contenute in un ciclo, queste saranno decifrate ogni volta che verranno incontrate all'interno del ciclo. Le prime versioni del linguaggio di programmazione BASIC furono implementate proprio come interpreti.

Scrivete un interprete per il linguaggio Semplice discusso nell'Esercizio 12.26. Il programma dovrà usare il convertitore da notazione infissa a polacca inversa, che avete sviluppato nell'Esercizio 12.12, e la funzione di valutazione di espressioni in notazione polacca inversa, che avete sviluppato nell'Esercizio 12.13, per valutare le espressioni di un'istruzione **let**. In questo programma, dovranno essere adottate le stesse restrizioni imposte dall'Esercizio 12.26 al linguaggio Semplice. Verificate il funzionamento dell'interprete con il programma in linguaggio Semplice scritto nell'Esercizio 12.26. Confrontate i risultati ottenuti dall'esecuzione interpretata di questo programma, con quelli ottenuti mediante la sua compilazione ed esecuzione con il simulatore Simpletron costruito nell'Esercizio 7.19.