FONDAMENTI DI INFORMATICA II

ESERCITAZIONE n° 1: Linguaggio C

1. Macro

ESERCIZIO n° 1
Definire una macro `ODD(i)` che restituisca `TRUE` o `FALSE` a seconda che il valore intero passato sia rispettivamente dispari o pari.

Soluzione
Poiché un intero è dispari se ha il bit meno significativo a 1, ed è pari in caso contrario, una possibile definizione può essere la seguente:

```
#define ODD(i)   ((i) & 1)
```

ESERCIZIO n° 2
Definire una macro `CPL2(i)` che restituisca il complemento a 2 del valore passatole come argomento.

Soluzione
Poiché il complemento a 2 si ottiene sommando 1 al complemento a 1 del numero dato, si ha semplicemente:

```
#define CPL2(a)     (1 + (~a))
```

ESERCIZIO n° 3
Definire una macro `AND(a,b)` che restituisca il risultato dell'operazione di AND logico fra i due parametri `a` e `b`.

Soluzione
Ricordando che nell'operazione di AND il risultato è certamente zero se uno dei due operandi è falso, mentre è uguale all'altro operando in caso contrario, si può scrivere:

```
#define AND(a,b)    ((a) ? (b) : 0)
```

ESERCIZIO n° 4
Definire una macro `DEMORGAN_AND(a,b)` che restituisca il risultato dell'operazione di AND logico fra i due parametri effettuando il calcolo sulla base delle identità del Teorema di De Morgan.

Soluzione
Poiché è noto che:

$$A \wedge B = \neg ((\neg A) \vee (\neg B))$$

si ha immediatamente:

```
#define DEMORGAN_AND( a, b ) ( ~( (~a) | (~b) ) )
```

ESERCIZIO n° 5

Definire una macro `GENERIC_SWAP( NAME, ELEM_TYPE )` che definisca una funzione di nome `NAME` che effettui lo scambio (swap) fra due elementi di tipo `ELEM_TYPE`: `NAME` deve ricevere come parametri i puntatori ai due elementi da scambiare. In altri termini:

`GENERIC_SWAP(int_swap, int)`	dovrebbe definire una funzione `int_swap` capace di scambiare due interi
`GENERIC_SWAP(float_swap, float)`	dovrebbe definire una funzione `float_swap` capace di scambiare due valori di tipo float

Soluzione

Poiché questa macro è decisamente più complessa di quelle fin qui viste, è necessario fare una precisazione. Come regola generale, le macro devono essere scritte su un'unica riga: qualora sia indispensabile andare a capo, è necessario, per indicare che il testo continua nella riga successiva, porre al termine della riga stessa un backslash '\'. Chiarito ciò, una macro che effettua quanto richiesto può essere la seguente:

```
#define GENERIC_SWAP( NAME, ELEM_TYPE ) \
    void NAME( a, b ) \
    ELEM_TYPE  *a,  *b; \
    { ELEM_TYPE  t; \
      t = *a;  *a = *b; *b = t; }
```

Nel primo dei due casi indicati nel testo dell'esercizio (`GENERIC_SWAP(int_swap, int)`) il risultato sarebbe quindi una funzione `int_swap` così definita:

```
void int_swap ( a, b )
int *a,  *b;
{ int  t;
  t = *a;  *a = *b; *b = t; }
```

2. Ricorsione

ESERCIZIO n° 6
Scrivere una funzione che, a ogni invocazione, restituisca il successivo numero primo.

Soluzione
Poiché il tempo di vita di una variabile locale di una funzione (variabile *automatica*) è quello della funzione stessa, a ogni nuova invocazione delle funzione, tutte le variabili locali sono concettualmente vuote: quelle per le quali è specificato un valore iniziale sono inizializzate a tale valore, le altre hanno un contenuto indefinito. Sebbene questo comportamento sia fortemente desiderabile e opportuno nella stragrande maggioranza dei casi, perché garantisce l'indipendenza di una invocazione da tutte quelle passate e future, esistono casi (come quello in esame) in cui, per il particolare compito svolto da una funzione, è utile che una o più variabili (a tutti gli effetti, *variabili di stato*) mantengano il loro valore fra un'invocazione e l'altra della funzione stessa. Questa necessità insorge particolarmente quando sono richieste *funzioni di generazione* di sequenze di valori che non devono ripetersi e in cui, eventualmente, ogni valore può dipendere dai precedenti.
In questi casi, è utile la possibilità, offerta dal C, di *dichiarare* `static` *delle variabili interne a una funzione*. Il significato di questa dichiarazione, del tutto diverso da quello di un'analoga dichiarazione applicata a un oggetto esterno, è che *una variabile locale dichiarata* `static` *mantiene il suo valore fra una chiamata e l'altra* della stessa funzione, il che è esattamente quanto ci serve per tenere traccia dell'ultimo numero primo generato.

Costruiremo quindi una funzione `nextPrime`, senza parametri, che a sua volta si avvarrà di una funzione `isPrime` per controllare che un dato valore sia primo. Quest'ultimo controllo, esclusi i casi banali dei numeri 1 e 2, verrà attuato col metodo di Eratostene, verificando che il numero (dispari) dato non sia divisibile per alcun numero minore della sua radice quadrata. Il programma potrà allora presentarsi come segue:

```
#include <stdio.h>
#include <math.h>    /* Include la libreria matematica (sqrt) */

#define NO     0
#define YES    1

int isPrime(int n)     /* Ritorna 0 o 1 secondo se n è primo    */
{
     int i, max=sqrt( (double)n );

     if (n>0 && n<4) return YES;
     else if (!(n%2)) return NO;

     for(i=3; i<=max; i+=2)
      if (!(n%i)) return NO;

     return YES;
}

int nextPrime(void)
```

```
{
     static n=0;    /* variabile statica con inizializzazione */

     if (n>=0 && n<=2) return ++n;
      else {
                do
                 n+=2;
                while(!isPrime(n));
                return n;
           }
}

void main()
{
     int p,max;

     do {
       printf("\nGenerare i numeri primi fino a:\t");
       scanf("%d",&max);
     } while(max<1);

     do {
      printf("\nNumero primo:\t%d", p=nextPrime() );
     } while(p<max);
}
```

L'unico elemento realmente innovativo è costituito dalla funzione `nextPrime`, che contiene al suo interno una dichiarazione di variabile statica il cui valore viene mantenuto fra una chiamata e l'altra. Una variabile statica è sempre inizializzata automaticamente dal sistema a zero, a meno che non sia espressamente indicato un diverso valore. Tale inizializzazione avviene una sola volta, concettualmente *prima* dell'inizio dell'esecuzione del programma. Perciò la variabile `n` viene azzerata (da codice generato ad hoc dal compilatore) ogni volta che il programma viene *rieseguito*, e *non a ogni invocazione* della funzione `nextPrime`, come invece avverrebbe per una normale variabile locale.

ESERCIZIO n° 7

Scrivere una funzione *ricorsiva* che sommi i primi *N* numeri naturali.

Soluzione

In C, una funzione può, se ciò è utile e necessario, richiamare se stessa, direttamente o indirettamente (cioè attraverso altre funzioni): può, cioè, essere *ricorsiva*. Dal punto di vista concettuale, non c'è alcuna differenza fra chiamare un'altra funzione o la stessa funzione: in entrambi i casi, infatti, vi è un *cliente* (main o altra funzione) che si rivolge a un *servitore* per l'espletamento di un particolare compito. Come caso particolare, una funzione può utilizzare se stessa come suo servitore, qualora ciò sia utile. Conseguentemente, anche il linguaggio tratta i due casi esattamente nello stesso modo. Tecnicamente, ciò è possibile perché ogni attivazione di una nuova funzione produce immediatamente la creazione (sul momento) del relativo *ambiente*, ossia di una zona con tutti i parametri e tutte le variabili locali della funzione stessa: ergo, ogni nuova

attivazione di funzione (anche della stessa funzione) è indipendente da tutte le attivazioni precedenti.

Per affrontare un problema con approccio ricorsivo, bisogna innanzitutto modificare il proprio modo di pensare, cambiando metodo rispetto al "classico" approccio iterativo. Infatti, mentre quest'ultimo richiede di cogliere l'essenza del problema ed essere in grado di esprimere il passo generico che porta alla soluzione, l'approccio ricorsivo si limita a tentare di *abbassare il grado di difficoltà* del problema, esprimendone la soluzione in termini di alcuni passi elementari *e inoltre* della soluzione dello stesso problema, in un caso più semplice.
Nel caso in esame, anziché vedere il problema come accumulazione progressiva di valori con relativi risultati parziali, per evidenziare un *passo elementare di decomposizione* si può osservare che se *N*=0 la somma è banalmente zero, mentre se *N*>0 la somma vale *N* più la somma dei valori fino a *N*-1. Quest'ultima somma non è altro che la soluzione del medesimo problema (sommare un certo insieme di numeri), applicato però al caso (più semplice) di *N*-1 valori anziché a *N* valori.

Questo porta a scrivere direttamente il codice della soluzione che si presenta come segue.

```
#include <stdio.h>

int sum_to(unsigned int n)
{
     if (n==0) return 0;
      else return n+sum_to(n-1);
}

main()
{
     int n;

     printf("\nIntrodurre N:\t");
     scanf("%d",&n);

     printf("\nSomma fino a %d:\t%d\n", n, sum_to(n));
}
```

La funzione `sum_to` opera in modo concettualmente assai semplice: se *n* è zero, il risultato della somma è zero; altrimenti, esso è pari alla somma di tutti gli *n-1* numeri precedenti (il cui calcolo è delegato alla stessa `sum_to`), più *n* stesso. Ad esempio, per *n*=4 si ha:

```
sum_to(4)  = 4 + sum_to(3) =
           = 4 + ( 3 + sum_to(2) ) =
           = 4 + ( 3 + ( 2 + sum_to(1) ) ) =
           = 4 + ( 3 + ( 2 + (1 + sum_to(0) ) ) ) =
           = 4 + ( 3 + ( 2 + (1 + 0 ) ) ) =                    /* § */
           = 4 + ( 3 + ( 2 + 1 ) ) =
           = 4 + ( 3 + 3 ) =
           = 4 + 6 =
           = 10.
```

Concretamente, ogni “espansione” di `sum_to` corrisponde alla creazione di un ambiente per una nuova attivazione di questa funzione. Quindi, in corrispondenza del punto §, vi sono cinque attivazioni di `sum_to` contemporaneamente attive, ognuna in attesa che la successiva produca il valore necessario per concludere la rispettiva operazione. Da § in poi, avendo l’ultima chiamata di `sum_to` fornito direttamente un risultato (0), le varie invocazioni iniziano a chiudersi, e inizia a formarsi il risultato finale. Questa successione di chiamate ricorsive può essere seguita col debugger del Turbo C, tramite l'opzione *call stack* del menù *debug*.

ESERCIZIO n° 8

Scrivere una funzione ricorsiva che stampi una parola rovesciata..

Soluzione

Per stampare a rovescio (cioè dall'ultima lettera alla prima) una parola, si può partire dall'idea di stampare innnanzitutto l'ultimo carattere, e poi stampare rovesciati (se esistono) tutti gli altri caratteri. Con questa semplice osservazione, si è espresso il problema di stampare rovesciata una parola lunga *n* tramite un'operazione elementare (la stampa del singolo carattere) e dello stesso problema in un caso più semplice (stampare rovesciati i precedenti *n*-1 caratteri).
La funzione richiesta può quindi agire esattamente in questo modo. Più precisamente, se la stringa passatale è lunga zero non fa nulla, se è lunga uno la stampa e termina, mentre se è lunga più di uno stampa l'ultimo carattere e richiama se stessa sulla sottostringa iniziale (privata, cioè, dell'ultimo carattere). Una possibile codifica può essere la seguente:

```
#include <stdio.h>
#include <string.h>
#define MAXLEN 30

void print_rev(char word[])
{
     int len=strlen(word);
     char w[MAXLEN];

     if (len<1) return;
     else if (len==1) putchar(word[0]);
      else {
               strncpy(w, word, len-1);
               w[len-1]='\0';
               putchar(word[len-1]);
               print_rev(w);
          }
     return;
}

main()
{
     int n;
     char parola[MAXLEN];

     printf("\nIntrodurre una parola:\t");
     scanf("%s",&parola);
```

```
    print_rev(parola);
}
```

ESERCIZIO n° 9

Realizzare una funzione che risolva ricorsivamente il problema della *Torre di Hanoi*.

La Torre di Hanoi

Si tratta di un antico gioco, in cui sono date tre torri (sinistra, centrale, e destra) e un certo numero di dischi forati. I dischi hanno diametro diverso gli uni dagli altri, e inizialmente sono infilati uno sull'altro (dal basso in alto) dal più grande al più piccolo sulla torre di sinistra. Scopo del gioco è portarli tutti sulla torre di destra, usando quella centrale come appoggio (torre di transito), e rispettando due regole:

a) si può muovere un solo disco alla volta;

b) un disco più grande non può mai stare sopra a un disco più piccolo.

Soluzione

Una soluzione generale, non ricorsiva, è molto difficile da dare, soprattutto quando il numero dei dischi supera poche unità (già 4 non è banale). La soluzione ricorsiva, invece, è immediata: per spostare *n* dischi dalla torre di sinistra alla torre di destra basta supporre di saper spostare gli *n-1* dischi superiori in quella centrale, muovere quello rimasto dalla torre di sinistra a quella di destra, e infine rispostare gli *n-1* superiori da quella centrale a quella di destra.

Ciò equivale a dire che il problema di grado *n* può essere ricondotto a una mossa elementare e a due problemi di grado *n-1*, che possono essere a loro volta risolti procedendo nello stesso modo. Ovviamente, a forza di decomporre il problema si arriva al punto di spostare un solo disco, che è una mossa elementare fattibile direttamente. Questa strategia, ancorché semplice, non è comoda da gestire a mano: viceversa è adattissima a un calcolatore. Il programma che la implementa è riportato qui di seguito.

```
#include <stdio.h>

typedef enum { SINISTRA, CENTRALE, DESTRA } torre;

void hanoi(int d, torre start, torre end, torre transit);

main()
{
 int d;

 printf("\nTORRE DI HANOI\n\n");
 printf("\nQuanti dischi?\t");
 scanf("%d", &d);
 printf("\n\nSOLUZIONE:\n");
 hanoi(d, SINISTRA, DESTRA, CENTRALE);
 return 0;
}

void hanoi(int d, torre iniziale, torre finale, torre ausiliaria)
{
 if (d==1)
```

```
 {
  printf("Muovi un disco dalla torre %d alla torre %d\n",
       iniziale, finale);
  return;
 }
 else
 {
  hanoi(d-1,iniziale, ausiliaria, finale);
  printf("Muovi un disco dalla torre %d alla torre %d\n",
       iniziale, finale);
  hanoi(d-1, ausiliaria, finale, iniziale);
  return;
 }
}
```

Osserviamo che per modellare la tre torri, si è introdotto, tramite la direttiva `typedef`, un nuovo tipo `torre`, che da questo momento è liberamente usabile al pari dei tipi predefiniti del C (int, char, etc). Quindi, sarà possibile, in particolare, definire e usare variabili di tipo `torre`.[1] Concretamente, essendo questo nuovo tipo definito come un'enumerazione, i tre identificatori `SINISTRA`, `CENTRALE` e `DESTRA` sono associati ai tre valori interi 0, 1 e 2: perciò l'output del programma si esprime in termini di "torre 0, torre 1, torre 2". Naturalmente sarebbe semplice prevedere forme di output più chiare e leggibili (ad esempio facendo stampare "sinistra" in luogo di "torre 0", etc.): questo miglioramento è lasciato al lettore. Piuttosto, interessa sottolineare che la funzione `hanoi` realizza esattamente la logica sopra riportata: infatti, nel caso di un solo disco lo muove direttamente, altrimenti prima ne sposta *n-1*, poi sposta quello rimasto, indi risposta nuovamente i precedenti *n-1*.
Le tre torri iniziale, finale e ausiliaria naturalmente variano da una chiamata all'altra, perché i vari sottoproblemi devono in generale spostare dei dischi fra due generiche torri (p. es. dalla torre centrale a quella di destra), per la qual operazione sfruttano come transito la terza torre (ad es. quella di sinistra).

Qualche Esercizio in più sulla Ricorsione

1) Stampare un vettore di interi in maniera ricorsiva.

2) Cercare un elemento in un vettore di interi in modo ricorsivo, nei due casi di vettore ordinato o non ordinato.

3) Cercare il massimo di un vettore di interi in maniera ricorsiva.

4) Scrivere un programma ricorsivo, costituito dal solo `main()`, che scriva un numero N fissato di volte la stringa "Scrivere 17 volte una stringa".

[1] Maggiori dettagli sulla direttiva typedef si possono trovare nel commento all'esercizio n° 20.

5) Calcolare il fattoriale di un numero N, inserito run-time da tastiera, sia in maniera iterativa che ricorsiva.

6) Scrivere un programma ricorsivo che sia in grado di calcolare il termine n-esimo di una successione di Fibonacci:
$f(0) = 0$ $f(1) = 1$ $f(x) = f(x+1) * [f(x) -7]$

7) Eseguire il *Merge-Sort* di un vettore di interi:

- dividere gli N elementi dell'array in due sottosequenze di N/2 elementi;
- ordinare ciascuna delle due sottosequenze in maniera ricorsiva;
- fondere le due sottosequenze per ottenere una sequenza ordinata.

3. Stringhe e Vettori (parte 1)

ESERCIZIO n° 10
Realizzare una funzione che copi una stringa in un'altra tramite puntatori.

Soluzione
Il problema della copiatura di una stringa in un'altra è interessante perché evidenzia i limiti della "analogia" fra puntatori e vettori. Per questo, nel seguito sono presentate e commentate tre versioni: una errata, le altre due entrambe corrette ma con diversa semantica.

Una prima idea può essere quella di assegnare direttamente un vettore di caratteri a un'altro, come nell'esempio che segue:

```
#include <stdio.h>

main()
{
 char s1[80],s2[80];

 printf("\nStringa 1:\t");
 scanf("%s",s1);

 s2=s1;                  /* ATTENZIONE:  ERRATO!!      */

 printf("\nStringa 1:\t%s",s1);
 printf("\nStringa 2:\t%s",s2);

 printf("\nModifica stringa 1:\t");
 scanf("%s",s1);
 printf("\nStringa 1:\t%s",s1);
 printf("\nStringa 2:\t%s",s2);
}
```

Tuttavia, assegnare direttamente una stringa a un'altra (qui, s1 a s2) *non significa copiarla,* ma soltanto (tentare di) assegnare all'una l'indirizzo dell'altra (qui, l'indirizzo di s1 all'indirizzo di s2). Ciò è però errato, non essendo ovviamente possibile cambiare l'indirizzo a cui si trova un vettore. Una tale operazione, *ancorché limitata ai soli puntatori*, è invece lecita se alla sinistra dell'assegnamento compare un *puntatore a char*, come nell'esempio che segue.

```
#include <stdio.h>

main()
{
 char s1[80], *s2;

 printf("\nStringa 1:\t");
 scanf("%s",s1);
```

```
 s2=s1;              /* LECITO, ma NON DUPLICA LA STRINGA! */

 printf("\nStringa 1:\t%s",s1);
 printf("\nStringa 2:\t%s",s2);

 printf("\nModifica stringa 1:\t");
 scanf("%s",s1);
 printf("\nStringa 1:\t%s",s1);
 printf("\nStringa 2:\t%s",s2);
}
```

Ora l'assegnamento è sintatticamente lecito, tuttavia esso *produce solo una copia di puntatori*, non delle variabili da essi puntate! Perciò, s1 e s2 referenziano ora entrambi *la stessa stringa*, e dunque qualunque modifica fatta sull'una si ritrova immediatamente sull'altra. Per copiare davvero una stringa in un'altra è invece necessaria una copia fisica, uno per volta, di tutti gli elementi, che si può attuare ad esempio tramite un ciclo:

```
#include <stdio.h>

void strcpy(char *s, char *t)
{
     while((*s=*t)!='\0') {s++; t++;}
}

main()
{
 char s1[80], s2[80];    /* stavolta, s2 deve avere lo spazio */

 printf("\nStringa 1:\t");
 scanf("%s",s1);

 strcpy(s2,s1);

 printf("\nStringa 1:\t%s",s1);
 printf("\nStringa 2:\t%s",s2);

 printf("\nModifica stringa 1:\t");
 scanf("%s",s1);
 printf("\nStringa 1:\t%s",s1);
 printf("\nStringa 2:\t%s",s2);
}
```

Osserviamo che la funzione strcpy, scritta in questo modo, è ridondante: un programmatore esperto l'avrebbe probabilmente sintetizzata così:

```
void strcpy(char *s, char *t)
{
     while(*s++=*t++);
}
```

Infatti, gli aggiornamenti dei due puntatori possono essere fatti efficacemente con due post-incrementi, e il confronto col carattere '\0' è, al solito, sovrabbondante.

Osserviamo anche che, per quanto sia stretta la correlazione fra i due, *vettori e puntatori non sono due concetti identici*. Ciò è confermato dal seguente esempio:

```
char msg[] = "testo del messaggio";   /* msg è un VETTORE   */
char *msg  = "testo del messaggio";   /* msg è un PUNTATORE */
```

Nel primo caso, `msg` è il nome di un vettore, la cui lunghezza è fissata staticamente: perciò, i singoli caratteri entro il "testo del messaggio" possono cambiare, ma `msg` si riferisce sempre alla stessa area di memoria. Non gli si può quindi assegnare, come nel primo degli esercizi precedenti, un altro indirizzo, perché il nome msg è univocamente associato a questa zona di memoria, e non ad altre.
Nel secondo caso, invece, essendo `msg` un puntatore, può essere fatto puntare altrove, ma il contenuto della stringa è una costante che non dovrebbe essere modificata. Tentando di farlo, il risultato è, in generale, indefinito[2].

ESERCIZIO n° 11

Scrivere una funzione C in grado di riconoscere identificatori, ovvero sequenze di caratteri che rispettino la seguente grammatica:

```
<identificatore> ::=<lettera> | <lettera> <caratteri>
<caratteri> ::=     <lettera> <caratteri> | <cifra> <caratteri>
<lettera> ::=       _ | a | b | c | ..... | z | A | B | C | ..... | Z
<cifra> ::=         0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Soluzione

Tradotto dal formalismo, si tratta evidentemente di riconoscere sequenze di caratteri che cominciano con una lettera o un underscore, e proseguono con lettere, undescore o cifre. Quindi, ci è richiesta una funziona che riceva in ingresso una stringa e, sostanzialmente, risponda SI o NO (oppure TRUE o FALSE, o 1 o 0, eccetera) a seconda che essa rispetti tale sintassi e costituisca, perciò, un identificatore, o meno. Da notare che è opportuno, per generalità, prevedere l'eventualità che esistano spazi iniziali, che ovviamente devono essere saltati. Una funzione che fa questo può essere la seguente.

```
#define NULL 0

char *identifier( char *s )
{
     while ( isspace(*s) ) s++;
     if ( !(isalpha(*s)) && *s != '_' ) return NULL;
      else s++ ;
     while ( isalpha(*s) || isdigit(*s) || *s=='_' )  s++ ;
     if ( isspace(*s) || *s=='\0' ) return s;
      else return NULL;
};
```

[2] La ragione di questo è abbastanza evidente: si pensi a cosa potrebbe accadere se più stringhe condividono, nella rappresentazione dei dati organizzata dal sistema. delle sottostringhe...

Questa funzione non soltanto verifica la sintassi degli identificatori: restituisce anche un puntatore al primo carattere nella stringa che non corrisponde alla sintassi (ad esempio uno spazio che separa l'identificatore stesso dalla parola successiva). Qualora la verifica dia esito negativo, per convenzione il valore ritornato è NULL.

ESERCIZIO n° 12

Scrivere una funzione C in grado di riconoscere espressioni matematiche che rispettino la seguente grammatica (che costituisce una versione semplificata di quella illustrata nella lezione n. 34):

```
<espressione> ::=   <termine> |
                    <termine> + <termine> |
                    <termine> - <termine>
<termine> ::=       <fattore> |
                    <fattore> * <fattore> |
                    <fattore> / <fattore>
<fattore> ::=       <identificatore> | <numero> | ( <espressione> )
```

Soluzione

In primo luogo, osserviamo che occorrerà disporre di idonee funzioni atte a riconoscere identificatori e numeri, che sono le entità di più basso livello trattate dalla grammatica. Rimandando per gli identificatori all'esercizio precedente, tralasceremo in questa sede di presentare il codice della funzione che riconosce numeri, in quanto facilmente realizzabile sulla scorta degli esercizi precedenti; ci concentreremo invece sulla parte, più innovativa, di riconoscimento delle espressioni.
Sebbene l'operazione di riconoscimento in sé sia non banale, a causa soprattutto della definizione ricorsiva dei fattori in termini di espressioni, la possibilità offerta dal C di specificare funzioni ricorsive consente di ottenere lo scopo con un codice soprendentemente breve e compatto, come mostra l'esempio sotto riportato.

```
#define NULL 0

char *term(char *);       /* dichiarazioni delle funzioni successive  */
char *factor(char *);
char *identifier(char *);       /* vedere esercizio precedente        */
char *number(char *);           /* da fare a cura del lettore         */

char *expression (char *s)
{
     char *t;
     printf("\nEspressione...");
     t = term(s);
     if ( t == NULL ) return NULL;
      else {
          s = t;
          while ( isspace(*s) ) s++ ;
          if (*s != '-' && s* != '+' ) return s;
                                              /* è un puro termine */
           else s++;
          while ( isspace(*s) ) s++ ;
          t = term(s);
          if ( t == NULL ) return NULL;
```

```
            else return t;
      }
}

char *term (char *s)
{
     char *f;
     printf("...termine...");
     f = factor(s);
     if ( f == NULL ) return NULL;
      else {
          s = f;
          while ( isspace(*s) ) s++ ;
          if (*s != '*' && *s != '/' ) return s;
                                          /* è un puro fattore */
           else s++;
          while ( isspace(*s) ) s++ ;
          f = factor(s);
          if ( f == NULL ) return NULL;
           else return f;
      }
}

char *factor (char *s)
{
     char *t;
     printf("...fattore...");
     t = identifier(s);
     if ( t != NULL ) return t;
      else  t = number(s);
     if ( t != NULL ) return t;
      else  {                             /* espress. fra parentesi */
          printf("...(espressione)...");
          while ( isspace(*s) ) s++ ;
          if (*s != '(' ) return NULL;
           else s++;
          while ( isspace(*s) ) s++ ;
          t = expression(s);              /* chiamata ricorsiva !! */
          if ( t == NULL ) return NULL;
           else s=t;
          while ( isspace(*s) ) s++ ;
          if (*s != ')' ) return NULL;
           else return ++s;
      }
}

     /* ------ un esempio di main ------- */

main()
{
 char s[250];
 char *t;
 printf("\nIntrodurre l'espressione:\t");
 scanf("%s", &s);
 t = expression(s);
 if ( t==NULL ) printf("\nEspressione errata\n");
  else
```

```
    printf("\nEspressione OK\nParte residua non riconosciuta:%s\n",t);
}
```

Come si può vedere, la funzione che analizza i termini è del tutto analoga a quella che analizza le espressioni, in ossequio al fatto che la struttura sintattica delle due entità è assolutamente analoga: in effetti, tutta la differenza sta nel nome dell'entità analizzata e negli operatori (additivi in un caso, moltiplicativi nell'altro) accettati e riconosciuti. Per questo, descriveremo solo la prima.
Nella funzione expression, in primo luogo si verifica la presenza di un termine: se esso non esiste, la funzione termina e fallisce restituendo NULL. Altrimenti, si cerca il primo carattere non-spazio e si controlla se si tratta di un operatore di somma (+ o -): se così non è si termina, restituendo il puntatore a tale elemento non identificato. Se invece si riscontra la presenza di "+" o "-", si parte, tolti gli eventuali spazi, alla ricerca di un nuovo termine. Di nuovo, se esso non viene trovato la funzione fallisce restituendo NULL, diversamente ritorna il puntatore (fornito dalla funzione term) al primo carattere non identificato.
La funzione factor è sostanzialmente simile, anche se formalmente diversa. In questo caso, si cerca in primo luogo un identificatore: se lo si trova si termina restituendo, al solito, il puntatore (fornito da identifier) al primo elemento non identificato; altrimenti, e qui è la sola reale differenza, si cambia strada (ovvero si tenta di applicare una diversa regola di produzione), controllando se non si tratti per caso di un numero. Di nuovo, se così è, si restituisce il puntatore al primo carattere non identificato, altrimenti si passa a tentare la terza e ultima via: quella corrispondente a un fattore realizzato racchiudendo una espressione fra parentesi.
Tolti gli spazi, si procede allora alla ricerca di una parentesi tonda aperta (fallendo se non la si trova), quindi si richiama ricorsivamente la funzione expression originaria per l'analisi del contenuto delle parentesi (fallendo se da essa risulta che non si tratta di una espressione corretta), infine si cerca la parentesi chiusa, fallendo se manca.
Da notare che, per come è definita la grammatica, vi sono alcune differenze rispetto al modo usuale di scrivere le espressioni: in particolare, ogni operatore di un dato livello deve agire esclusivamente su due entità ben definite, eventualmente delimitate da parentesi. Ne segue che

A + 3 6 - 8 * 5 4 / 3 - k2 sono legali,

mentre

a + 3 + 5 4 * 5 / 8 5 - k + 8 sono errate;

la versione corretta sarebbe:

a + (3+5) 4 * (5/8) (5-k) + 8

Questa apparente complicazione è alla base della semplicità con cui è stato possibile scrivere il codice: la grammatica completa della lezione 34 sarebbe stata decisamente più complessa, essenzialmente a causa del fatto che gli operatori devono risultare associativi a sinistra (ovvero, 7 - 4 - 6 deve essere letto come (7-4)-6, e non come 7-(4-6)), il che richiede un'analisi della stringa più sofisticata, con capacità di "guardare avanti". Non approfondiamo oltre questo aspetto ora: chi desidera può comunque provare a riflettere su come differenze grammaticali anche piccole possano comportare notevoli complicazioni nell'analizzatore corrispondente.