Parte xxx

BC: linguaggio aritmetico a precisione arbitraria

«

BC: esempi di programmazione

BC, ovvero *Basic calculator*, è un interprete di un linguaggio aritmetico, descritto nella sezione 22.14. Questo capitolo raccoglie solo

alcuni esempi di programmazione, in parte già descritti in altri capitoli. Per eseguire questi esempi basta usare il comando seguente, dove '`prova.b`' rappresenta il nome del file da eseguire:

```
$ bc prova.b[Invio]
```

Si vuole evitare l'uso di estensioni al linguaggio BC, per cui i programmi non vengono mostrati come script; inoltre manca la possibilità di controllare l'interazione con l'utilizzatore, quindi le funzioni devono essere richiamate manualmente e al termine si deve usare il comando '**`quit`**', oppure si conclude il flusso dello standard input con la combinazione [*Ctrl d*].

Negli esempi non si fa uso delle librerie standard, pertanto i nomi relativi possono essere riutilizzati.

Le espressioni vengono scritte in modo da evitare la visualizzazione non desiderata. Per esempio, invece di '**`i++`**', si preferisce usare la forma '**`i=(i+1)`**', quando possibile.

Bisogna ricordare che se non si assegna il risultato generato da una funzione, questo viene visualizzato. La variabile '**`t`**' è stata usata negli esempi per raccogliere questo risultato quando non desiderato.

Problemi elementari di programmazione

In questa sezione vengono mostrati alcuni algoritmi elementari portati in BC. Per la spiegazione degli algoritmi, se non sono già conosciuti, occorre leggere quanto riportato nel capitolo ??capitolo programmazione pseudo??.

Somma tra due numeri positivi

Il problema della somma tra due numeri positivi, attraverso l'incremento unitario, è descritto nella sezione 62.3.1.

```
/*
    somma.b
    Somma esclusivamente valori positivi.
*/

define s (x, y) {
    auto z, i
    z=x
    for (i=1; i<=y; i++) {
        z=(z+1)
    }
    return (z)
}

"Per calcolare la somma, si utilizzi la funzione s (x, y): "
```

In alternativa si può tradurre il ciclo '**`for`**' in un ciclo '**`while`**':

```
define s (x, y) {
    auto z, i
    z=x
    i=1
    while (i<=y) {
        z=(z+1)
        i=(i+1)
    }
    return (z)
}
```

Moltiplicazione di due numeri positivi attraverso la somma

«

Il problema della moltiplicazione tra due numeri positivi, attraverso la somma, è descritto nella sezione 62.3.2.

```
/*
    moltiplica.b
*/

define m (x, y) {
    auto z, i
    z=0
    for (i=1; i<=y; i++) {
        z=(z+x)
    }
    return (z)
}

"Per calcolare la moltiplicazione, si utilizzi la funzione m (x, y): "
```

In alternativa si può tradurre il ciclo '**`for`**' in un ciclo '**`while`**':

```
define m (x, y) {
    auto z, i
    z=0
    i=1
    while (i<=y) {
        z=(z+x)
        i=(i+1)
    }
    return (z)
}
```

Divisione intera tra due numeri positivi

Il problema della divisione tra due numeri positivi, attraverso la sottrazione, è descritto nella sezione 62.3.3.

```
/*
    dividi.b
    Divide esclusivamente valori positivi.
*/

define d (x, y) {
    auto z, i
    z=0
    i=x
    while (i>=y) {
        i=(i-y)
        z=(z+1)
    }
    return (z)
}

"Per calcolare la divisione intera, si utilizzi la funzione d (x, y): "
```

Elevamento a potenza

«

Il problema dell'elevamento a potenza tra due numeri positivi, attraverso la moltiplicazione, è descritto nella sezione 62.3.4.

```
/*
    exp.b
*/

define x (x, y) {
    auto z, i
    z=1
    for (i=1; i<=y; i++) {
        z=(z*x)
    }
    return (z)
}

"Per calcolare l'elevamento a potenza, si utilizzi la funzione x (x, y): "
```

In alternativa si può tradurre il ciclo '**`for`**' in un ciclo '**`while`**':

```
define x (x, y) {
    auto z, i
    z=1
    i=1
    while (i<=y) {
        z=(z*x)
        i=(i+1)
    }
    return (z)
}
```

È possibile usare anche un algoritmo ricorsivo:

```
define x (x, y) {
    if (x==0) {
        return (0)
    }
    if (y==0) {
        return (1)
    }
    return (x * x (x, y-1))
}
```

Radice quadrata

Il problema della radice quadrata è descritto nella sezione 62.3.5.

```
/*
    radice.b
*/

define r (x) {
    auto z, y
    z=0
    y=0
    while (1) {
        y=(z*z)
        if (y>x) {
            /* È stato superato il valore massimo. */
            z=(z-1)
            return (z)
        }
        z=(z+1)
    }
```

```
}

"Per calcolare la radice quadrata, si utilizzi la funzione r (x): "
```

Fattoriale

«

Il problema del fattoriale è descritto nella sezione 62.3.6.

```
/*
    fatt.b
*/

define f (x) {
    auto i
    i=(x-1)
    while (i>0) {
        x=(x*i)
        i=(i-1)
    }
    return (x)
}

"Per calcolare il fattoriale, si utilizzi la funzione f (x): "
```

In alternativa, l'algoritmo si può tradurre in modo ricorsivo:

```
define f (x) {
    if (x>1) {
        return (x * f (x-1))
    }
    return (1)
}
```

Massimo comune divisore

«

Il problema del massimo comune divisore, tra due numeri positivi, è descritto nella sezione 62.3.7.

```
/*
    mcd.b
*/
```

```
define m (x, y) {
    auto n
    while (x!=y) {
        n=0
        if (x>y) {
            x=x-y
            n=1
        }
        if (n==0) {
            y=(y-x)
        }
    }
    return (x)
}

"Per calcolare il massimo comune divisore, "
"si utilizzi la funzione m (x, y): "
```

Numero primo

Il problema della determinazione se un numero sia primo o meno, è descritto nella sezione 62.3.8.

```
/*
    primo.b
*/

define p(x) {
    auto i, j
    i=2
    while (i<x) {
        scale=0
        j=(x/i)
        j=x-(j*i)
        if (j==0) {
            return (0)
        }
        i=(i+1)
    }
    return (1)
}
```

```
"Per verificare se un numero sia primo, si utilizzi la funzione p (x, y); "
"1 indica un numero primo, 0 indica un numero che non è primo. "
```

Scansione di array

«

In questa sezione vengono mostrati alcuni algoritmi, legati alla scansione degli array, portati in BC.

Per usare questi programmi, mancando un sistema normale di interazione con l'utilizzatore, è necessario creare un array prima di utilizzare la funzione che svolge il lavoro di ricerca o di riordino. Per esempio, nel caso della funzione '**r()**' per la ricerca sequenziale:

```
$ bc ricercaseq.b [Invio]

  Ricerca sequenziale: r (<lista>, , <elemento>, <inizio>, <fine>)
a[0]=3 [Invio]

a[1]=10 [Invio]

a[2]=33 [Invio]

a[3]=56 [Invio]

r (a[], 33, 0, 3) [Invio]

  2
[Ctrl d]
```

Ricerca sequenziale

«

Il problema della ricerca sequenziale all'interno di un array, è descritto nella sezione 62.4.1.

```
/*
    ricercaseq.b
*/

/* r (<lista>, <elemento>, <inizio>, <fine>) */
define r (l[], x, a, z) {
    auto i
    for (i=a; i<=z; i++) {
        if (x==l[i]) {
            return (i)
        }
    }
    /* La corrispondenza non è stata trovata. */
    return (-1)
}

"Ricerca sequenziale: r (<lista>, , <elemento>, <inizio>, <fine>) "
```

Esiste anche una soluzione ricorsiva che viene mostrata nella funzione seguente:

```
define r (l[], x, a, z) {
    if (a>z) {
        return (-1)
    }
    if (x==l[a]) {
        return (a)
    }
    return (r (l[], x, a+1, z))
}
```

Ricerca binaria

«

Il problema della ricerca binaria all'interno di un array, è descritto nella sezione 62.4.2.

```
/*
    ricercabin.b
*/

/* r (<lista>, <elemento>, <inizio>, <fine>) */
define r (l[], x, a, z) {
    auto m
    /* Determina l'elemento centrale. */
    scale=0
    m = ((a+z)/2)
    if (m<a) {
        /* Non restano elementi da controllare: l'elemento cercato non c'è. */
        return (-1)
    }
    if (x<l[m]) {
        /* Si ripete la ricerca nella parte inferiore. */
        return (r (l[], x, a, m-1))
    }
    if (x>l[m]) {
        /* Si ripete la ricerca nella parte superiore. */
        return (r (l[], x, m+1, z))
    }
    /* $m rappresenta l'indice dell'elemento cercato. */
    return (m)
}

"Ricerca binaria: r (<lista>, <elemento>, <inizio>, <fine>) "
```

Algoritmi tradizionali

«

In questa sezione vengono mostrati alcuni algoritmi tradizionali portati in BC.

Per consentire la visualizzazione del contenuto di un array è necessario predisporre una funzione apposita, che viene presentata qui, senza ripeterla nei vari esempi proposti (per evitare di visualizzare uno zero aggiuntivo, conviene assegnare il valore restituito dalla funzione stessa).

```
/* v (<lista>, <inizio>, <fine>) */
define v (l[], a, z) {
    auto j
    for (j=a; j<=z; j++) {
        (l[j])
    }
    return
}
```

Bubblesort

Il problema del Bubblesort è stato descritto nella sezione 62.5.1. Viene mostrata prima una soluzione iterativa e successivamente la funzione '**`bsort()`**' in versione ricorsiva.

```
/*
    bsort.b
*/

/* l[] è l'array da riordinare. */

/* b (<inizio>, <fine>) */
define b (a, z) {
    auto s, j, k

    /* Inizia il ciclo di scansione dell'array. */
    for (j=a; j<z; j++) {
        /*
            Scansione interna dell'array per collocare nella posizione j
            l'elemento giusto.
        */
        for (k=(j+1); k<=z; k++) {
            if (l[k]<l[j]) {
                /* Scambia i valori */
                s=l[k]
                l[k]=l[j]
                l[j]=s
            }
        }
    }
    return
}
```

```
"Bubblesort: l[]; t = b (<inizio>, <fine>)    "
"L'array da riordinare è l[]. "
```

Segue la funzione '**bsort()**' in versione ricorsiva:

```
define b (a, z) {
    auto s, k
    if (a<z) {
        /*
            Scansione interna dell'array per collocare nella posizione a
            l'elemento giusto.
        */
        for (k=(a+1); k<=z; k++) {
            if (l[k]<l[a]) {
                /* Scambia i valori */
                s=l[k]
                l[k]=l[a]
                l[a]=s
            }
        }
        b (a+1, z)
    }
    return
}
```

Torre di Hanoi

«

Il problema della torre di Hanoi è descritto nella sezione 62.5.3.

```
/*
    hanoi.b
*/

/* h (<n-anelli>, <piolo-iniziale>, <piolo-finale>) */
define h (n, i, f) {
    auto t
    if (n>0) {
        t = h (n-1, i, 6-i-f)
        "Muovi l'anello " ; n
        "dal piolo " ; i
        "al piolo " ; f
        t = h (n-1, 6-i-f, f);
```

```
    }
    return
}

"Torre di Hanoi: t = h (<n-anelli>, <piolo-iniziale>, <piolo-finale>) "
```

Quicksort

«

L'algoritmo del Quicksort è stato descritto nella sezione 62.5.4.

```
/*
    qsort.b
*/

/* l[] è l'array da riordinare. */

/* p (<inizio>, <fine>) */
define p (a, z) {
    auto s, i, c
    /* Si assume che a sia inferiore a z. */
    i=(a+1)
    c=z

    /* Inizia il ciclo di scansione dell'array. */
    while (1) {
        while (1) {
            /* Sposta i a destra. */
            if (l[i]>l[a]) {
                break
            }
            if (i>=c) {
              break
            }
            i=(i+1)
        }
        while (1) {
            /* Sposta c a sinistra. */
            if (l[c]<=l[a]) {
                break
            }
            c=(c-1)
        }
        if (c<=i) {
```

```
            /* È avvenuto l'incontro tra i e c. */
            break
        }
        /* Vengono scambiati i valori. */
        s=l[c]
        l[c]=l[i]
        l[i]=s

        i=(i+1)
        c=(c-1)
    }

    /*
        A questo punto l[a..z] è stata ripartita e c è la collocazione
        di l[a].
    */
    s=l[c]
    l[c]=l[a]
    l[a]=s

    /*
        A questo punto l[c] è un elemento (un valore) nella
        posizione giusta.
    */
    return (c)
}

/* q (<inizio>, <fine>) */
define q (a, z) {
    auto c
    if (z>a) {
        c = p (a, z)
        q (a, c-1)
        q (c+1, z)
    }
    return
}

"Quicksort: l[]  t = q (<inizio>, <fine>)  "
"Prima riempire l'array l[], poi chiamare la funzione q()."
```

«

L’algoritmo ricorsivo delle permutazioni è descritto nella sezione 62.5.5.

```
/*
    permuta.b
*/

/* v (<lista>, <inizio>, <fine>) */
define v (l[], a, z) {
    auto j
    for (j=a; j<=z; j++) {
        (l[j])
    }
    return
}

/* p (<lista>, <inizio>, <fine>, <max_array>) */
define p (l[], a, z, d) {
    auto k
    auto t
    if ((z-a)>=1) {
        /*
            Inizia un ciclo di scambi tra l’ultimo elemento e uno degli
            altri contenuti nel segmento di array.
        */
        for (k=z; k>=a; k--) {
            /* Scambia i valori */
            s=l[k]
            l[k]=l[z]
            l[z]=s

            /*
                Esegue una chiamata ricorsiva per permutare un segmento
                più piccolo dell’array.
            */

            t = p (l[], a, z-1, d)

            /* Scambia i valori */
            s=l[k]
            l[k]=l[z]
            l[z]=s
```

```
        }
        return
    }
    /* Visualizza la situazione attuale dell'array. */
    " "
    t = v (l[],0,d)
    return
}

"Permutazioni: t = p (<lista>, <inizio>, <fine>, <max_array>)"
```