Pràctiques Integrades
1er de Matemàtiques

Pràctica 7

curs 2002–03

7 Programació: lògica, iteracions i procediments

No sempre Maple tindrà les comandes que necessitem per a realitzar algun treball o càlcul concret, o bé ens agradaria iterar l'execució d'un paquet de comandes vàries vegades. En aquests casos necessitarem definir les nostres pròpies comandes (o funcions). Per això hem d'aprendre com crear procediments (comandes pròpies o blocs de programa) on podrem incloure funcions lògiques, iteracions i procediments.

Si ja heu programat en algun llenguatge informàtic fàcilment podreu reconèixer els operadors i funcions lògiques que apareixeran al llarg de la pràctica.

7.1 Lògica: `if-then-elif-else-end if`

La comanda bàsica que ens permetrà implementar raonaments lògics és la comanda `if` amb totes les seves variants. Farà que Maple executi diferents seqüències de comandes en funció del compliment d'unes condicions (que Maple avaluarà com certes o falses). El cas més simple correspon a `if...then...end if` que executarà una seqüència de comandes si es compleix una condició.

```
> restart;
> a:=3;
> if a=4 then b:=a+1 end if;
> if a=3 then b:=a end if;
```

Ara bé, aquest operador lògic `if` admet moltes variants que ens permetran combinar diferents condicions. Per exemple l'operador `else` que precedeix el codi que caldrà executar en el cas que no es compleixi la condició. També admet l'operador `elif` que ens permetrà considerar més d'una condició. En l'exemple següent podem veure el seu funcionament.

Exemple 7.1

Suposem que volem definir una funció `f(x)` que valgui -1 si $x < 0$, 0 si $x = 0$ i 1 si $x > 0$. Podríem fer-ho de la següent manera utilitzant `if`.

```
> f:=x-> if x<0 then -1 elif x=0 then 0 else 1 end if;
```

Comproveu que funciona executant:

```
> f(-.5);f(0);f(.5);
```

De fet, la funció que acabem de definir a l'exemple anterior ja existeix en Maple amb el nom de `signum()`.

```
> signum(-.5);signum(0);signum(.5);
```

El que hem fet és definir la funció `f` on a la seva definició hauríem de traduir `if` per "*si*", `elif` per "*altrament si*", `else` per "*altrament*" i `end if` per "*hem acabat*". Llavors podem pensar la comanda `if` com una comanda amb tres elements:

- El primer element que hem de tenir en compte és la sintaxi de la funció. Hem d'entrar els arguments `if ... then, elif ... then, else,` i `end if` sempre en aquest ordre. Els operadors `if..then..end if` són necessaris, la resta són opcionals.
- El segon és la introducció de condicions lògiques després dels corresponents `if`. Això són expressions com $x > 0$, $y \leq 3$, ... i que en Maple s'han d'introduir mitjançant els operadors relacionals següents:

`<`	més petit que,
`<=`	més petit o igual que,
`>`	més gran que,
`>=`	més gran o igual que,
`=`	igual,
`<>`	diferent.

  Quan Maple troba alguna d'aquestes expressions l'avalua com una "*expressió Booleana*", o sigui, com una expressió que pot ser "*certa*" o "*falsa*". Les avaluacions booleanes d'expressions també es poden realitzar directament mitjançant la funció `evalb( )`. Analizeu el resultat obtingut a l'executar la següent línia de comanda:

```
> evalb(7>5);evalb(3=4);evalb(3<>4);
```

  Podem fer construccions més complicades, combinant els operadors anteriors amb els operadors lògics `and`, `or`, `xor` i `not`. Podeu consultar l'ajuda per obtenir més detalls sobre els operadors lògics escrivint `?boolean;`.

- El tercer element que formarà part d'una comanda lògica són les comandes que s'han d'executar en cada un dels casos que es poden produir. En l'exemple tan sols havíem d'introduir -1, 0 i 1, però es poden introduir expressions més complexes, Per exemple, volem definir una funció `g` que prengui valors segons la següent definició:

```
> g:=x-> if x<0 then 1/(1+x^2) else cos(x) end if;
```

  Podem comprovar que funciona amb alguns exemples,

```
> g(-1.5);g(2.5);
```

  Tot i això, què passa en el següent exemple?

```
> g(Pi);
```

  El missatge d'error diu que no ha pogut avaluar la condició booleana que hem introduït. Això és a causa del fet que `Pi` no és exactament un número. De fet, obtenim exactament el mateix error quan avaluem `g` a una variable indeterminada x:

```
> g(x);
```

  Una manera de solucionar el problema és utilitzant la comanda `evalf()` i avaluar `g` en una aproximació numèrica de `Pi`.

```
> g(evalf(Pi));
```

Finalment intentem ara representar gràficament la funció `g` que hem definit:

```
> plot(g(x),x=-10..10);
```

Tornem a obtenir errors en les expressions booleanes. El problema és que quan substitueix x a la comanda `g` aquesta encara no és un número i per això retorna l'error. En aquest cas el que podem fer és introduir la "*notació d'operadors*", o sigui, no escriure la variable `x`. Recordeu que no era necessària a l'hora d'usar la comanda `plot` per representar gràficament una funció (no una expressió).

```
> plot(g,-10..10);
```

Però si volem utilitzar la funció `g` per a definir altres funcions, per exemple, si volem calcular la seva derivada, també tindrem problemes. Vegeu-ho en el següent exemple.

```
> diff(g(x),x);
```

La funció `if` és molt útil en segons quins contexts, però si el que volem és definir una funció a trossos, no és la més apropiada com acabem de veure. Per a fer això és més versàtil la funció `piecewise( )`. Podem redefinir la funció `g` utilitzant aquesta nova comanda:

```
> g:=x->piecewise(x<0, 1/(1+x^2),x>=0, cos(x));
```

Fixeu-vos en els paràmetres que necessita, és a dir, hem d'introduir entre els parèntesis les corresponents seqüències de "condició, funció" separades per comes. Si escrivim una última funció sense cap condició prèvia, aquesta és la que executarà en cas que no es compleixin cap de les condicions anteriors en la definició.

Ara podem treballar amb `g` com amb qualsevol altre funció, per exemple,

```
> g(Pi); g(a);
> plot(g(x),x=-5..5);
```

i calcular la seva derivada:

```
> diff(g(x),x);
```

Per tant per a definir funcions a trossos utilitzarem la comanda `piecewise`, mentre que reservarem `if-then-elif-else-end if` per a utilitzar-la en altres ocasions.

Exercici 7.1

Utilitzeu la comanda `piecewise` per a definir amb Maple la funció que avalua la següent funció i feu la gràfica a l'interval $(-4, 4)$.

$$f(t) = \begin{cases} \dfrac{(2+t)^3}{6} & \text{si } t \in [-2,-1) \\ \dfrac{-3t^3-6t^2+4}{6} & \text{si } t \in [-1,0) \\ \dfrac{3t^3-6t^2+4}{6} & \text{si } t \in [0,1) \\ \dfrac{-(t-2)^3}{6} & \text{si } t \in [1,2) \\ 0 & \text{si } t \notin [-2,2) \end{cases}$$

Segons com feu la definició haureu d'introduir rangs amb límit inferior i límit superior. Per a això hem d'introduir dues desigualtats, i això ho hem de fer amb l'ajuda de l'operador lògic `and`. (Encara que si s'ordenen adequadament les diferents condicions es pot escriure una comanda sense `and`).

7.2 Iteracions: `for`, `do`, `end do` i `while`

Una iteració consisteix en executar una secció de codi vàries vegades. Moltes comandes a Maple contenen iteracions: les comandes `sum`, `fsolve`, ... La comanda `sum` és semblant a la comanda `seq` que ja coneixeu amb la diferència que retorna la suma total de la seqüència de valors obtinguts. Com a exemple, calculem la suma dels enters entre 1 i 100 elevats al quadrat:

```
> sum(i^2,i=1..100);
```

En alguna part del procés Maple ha aplicat una iteració per a fer el càlcul. Tot i això algun cop necessitarem definir les nostres pròpies iteracions. Fet "a mà" hauríem fet:

Iteració suma:

Decidim primer el nom de la variable, i la inicialitzem a zero.

```
> resp:=0;
```

Seguidament cal introduir la iteració. Per a això utilitzem les comandes `for... from... by... to... while... do... end do`. El bloc de comandes a iterar s'ha d'introduir sempre entre `do` i `end do`.

```
> for i from 1 to 100 do
  resp:=resp + i^2;
  end do;
```

Per a evitar la sortida de tots els passos, podem canviar el punt i coma (`;`) per uns dos punts (`:`) i demanar el valor de la variable `resp` al final del procediment.

```
> resp:=0; for i from 1 to 100 do
> resp:=resp + i^2; end do:
> resp;
```

Exercici 7.2

Calculeu la suma de tots els nombre senars entre 1 i 99 utilitzant la comanda `for` (consulteu l'ajuda per veure la opció `by` dins de la comanda `for`).

Més iteracions:

A vegades no sabrem el nombre exacte d'iteracions que volem fer sinó que la informació que tindrem serà que hem d'iterar un bloc de comandes fins que alguna condició fixada es compleixi. Per exemple, què passa si pitgeu indefinidament la tecla "*cosinus*" de la vostra calculadora, començant pel valor 5? A l'exemple següent ho farem 20 vegades:

```
> x:=5.; for i from 1 to 20 do x:=cos(x);
> end do;
```

Podeu observar que sembla que la successió es va apropant a un valor determinat. De fet, podeu comprovar que aquest valor ha de complir l'equació $\cos(x) = x$. Tot i això, veureu que amb 20 cops no n'hi ha prou per a obtenir una successió que no variï (fixant 8 dígits correctes), per tant, el que volem fer és iterar el procediment fins que la diferència entre dos passos sigui inferior a un número fixat, per exemple, 10^{-8}.

Necessitem dues variables, una serà `x` i la segona serà `xold`, que en tot moment serà el valor de la iteració anterior (haurem de donar-li un valor inicial diferent del que donem a `x`).

```
> x:= 5.; xold:= 0.;
```

Llavors hem d'iterar mentre es compleixi que $|\texttt{x} - \texttt{xold}| > 10^{-8}$. La comanda la podríem entrar de la manera següent.

```
> while abs(x-xold)>1e-8 do
  tempx:=cos(x):
  xold:=x:
  x:=tempx:
  end do;
> xold; x;
```

En qualsevol iteració l'única part obligatòria és el `do..end do`. Les parts `for` i `while` són opcionals i serveixen per controlar el número d'iteracions que cal fer per obtenir el resultat desitjat.

Les diferents maneres de realitzar iteracions de blocs de comandes es poden combinar, és a dir, podem tenir un `while` dins d'un `for`.

Exercici 7.3

Llegiu el següent bloc de comandes i abans d'executar-lo escriviu el que vosaltres creieu que serà la sortida. La comanda `isprime( )` és una funció Booleana que retorna el valor `true` o `false` segons si el número és primer o no.

```
> for p from 1000 by -1 while not isprime(p) do
  end do;
  p;
```

Hi ha una comanda que permet aturar una iteració en qualsevol punt del bloc de comandes que iterem: `break`. Observeu el següent exemple,

```
> for p from 1000 by -1 do
  if isprime(p) then break end if;
  end do;
  p;
```

7.3 Procediments

Al definir una funció pot passar que necessitem incloure-hi vàries iteracions, condicions lògiques, altres funcions, ...En general potser necessitarem dues o més instruccions de Maple encadenades per a fer els càlculs. També és bastant comú que ens aquests casos utilitzem variables que només tenen sentit dins el càlcul de la funció que estem definint (mentre aquesta s'executi), per buidar-se en acabar l'execució, i no a la resta del full de treball. Això és el que anomenarem **variables locals**, mentre que les que són vàlides a tot el full de treball les anomenarem **variables globals**.

Per a definir funcions que inclouen variables locals i globals hem d'utilitzar la funció `proc`. El procediment o bloc de programa està definit per `proc ... end proc`. Al costat de `proc` hem de definir els paràmetres que el nostre procediment utilitzarà. Després de fixar els paràmetres, cal declarar les variables que utilitzarem, això es realitza amb la comanda `local` si són locals (es destruïran en acabar l'execució del programa) i/o amb la comanda `global` si són globals.

La millor manera d'entendre la sintaxi de definició d'un procediment és analitzant un exemple concret.

Exemple 7.2

Mireu l'estructura de la definició següent d'un procediment:

```
> f:=proc(x,y)
  local c,b,z;
  global d;
  c:=3.;b:=17.;
  z:=c*b*x*y*d;
  end proc;
```

Podem veure com s'executa aquesta definició provant:

```
> f(5,5);
> d:=2.; f(5,5);
> d:='d'; f(5,5);
> c; b; z;
```

Exemple 7.3

La funció següent avalua el cosinus d'un angle en graus:

```
> cosdeg:=proc(theta)
  local resultat,phi;
  phi:=theta*Pi/180;
  resultat:=cos(phi);
  end proc;
```

Proveu calculant el cosinus de 45 graus.

Si voleu veure tots els passos que segueix un procediment des dels paràemtres que li passem fins al resultat final, ho podeu fer amb la funció `trace`. Aquesta funció permet "*activar*" per pantalla els passos que hi ha dins de la funció `proc`.

Exemple 7.4

Per a activar la funció `cosdeg` hem de fer:

```
> trace(cosdeg);
```

llavors l'execució fa:

```
> cosdeg(45);
```

Si volem desactivar aquesta la visualització podem fer:

```
> untrace(cosdeg);
```

7.4 Exercicis

Els exercicis següents s'han pensat per a que practiqueu els diferents conceptes de "*programació*" que hem introduït, però també pretenen il·lustrar alguns conceptes matemàtics. No us oblideu d'interpretar els resultats que aneu obtenint.

Exercici 7.4

Considereu la funció:

$$\texttt{aprcos}(x,n) = \sum_{i=0}^{n} (-1)^i \frac{x^{2i}}{(2i)!}$$

on x és un paràmetre real i n és un enter positiu. Dibuixeu en un sol gràfic les funcions `cos(x)`, `aprcos(x,1)`, `aprcos(x,2)` i `aprcos(x,4)` per a `x` entre els valors -2π i 2π.

Exercici 7.5

Definiu una funció que depengui de tres paràmetres `aprox(funcio,x,iteracions)` i que retorni una llista amb els valors de la forma `[x + (1/i),f(x + (1/i) )]` per a `i=1..iteracions`.

Avalueu `aprox` a la funció $\dfrac{\sin(x)}{x}$, per a $x = 0$ amb 50 iteracions.

Modifiqueu la funció `aprox` per a que a més retorni un dibuix dels punts calculats.

Exercici 7.6

Definiu una funció `ordre` tal que donada una permutació σ en calculi el seu ordre determinant el primer nombre natural n tal que σ^n és la permutació identitat.

Calculeu l'ordre de les permutacions $\sigma_1 = (1,3,5)(2,4)$ i $\sigma_2 = (1,2,3,4,5)(6,7,8)$ utilitzant la funció `ordre` que acabeu de definir.

Exercici 7.7

Feu un procediment que donat n faci la llista de totes les permutacions de n elements, cada una en format de llista (és a dir, la permutació σ es representa per la llista `[`σ`(1), ...`σ`(n)]`).