Menjelaskan *Maple Programme*

```
> f := sin(x[1])^2*(1-cos(Pi*x[2]));

                              2
                     sin(x[1])  (1 - cos(Pi x[2]))

> whattype(f);

                               *

> op(1,f);

                                      2
                             sin(x[1])

> op(1,op(1,f));

                             sin(x[1])

> op(1,op(1,op(1,f)));

                               x[1]
```

F =$\sin(x[1])^2\ (1-\cos(Pi\ x\ [2]))$

Ahtatype berfungsi untuk melihat jenis operasi f yaitu kali.

Op ada operasi perkalian misalkan op(1,f) artinya operasi perkalian pada posisi pertama

Op(1,op(1,f)) artinya posisi pertama perkalian adalah $\sin(x[1])^2$ dan op lagi pada posisi tersebut adalan $\sin(x[1])$ dan seterusnya

if *expr* **then** *statseq*
[**elif** *expr* **then** *statseq*]*
[**else** *statseq*]
fi

```
if x < 0 then -1 elif x = 0 then 0 else 1 fi
```

[**for** *name*] [**from** *expr*] [**by** *expr*] [**to** *expr*] [**while** *expr*]
do *statseq* **od**

```
for i to 10 do print(i^2) od;
```

If one omits the **for**, **from**, **by** and **to** clauses we have a so called **while** loop.

```
i := 10^10+1;
while not isprime(i) do i := i + 2 od;
print(i);
```

Isprime berfungsi untuk menentukan apakah bilangan merupakan bilangan prima atau bukan.

Combining the **for** and **while** loops is sometimes nice, e.g. this example of searching for the first prime $> 10^{10}$ could be done as

```
for i from 10^10+1 by 2 while not isprime(i) do od;
print(i);
```

The second variation on the **for** loop is the so called **for-in** loop. It is really just a short hand for this loop which occurs very often

```
for i to nops(s) do f(op(i,s)) od;
```

```
> nops(op(2,p));          # It has 3 factors
                                  3
```

Nops berfungsi untu menentukan banyak unsur.

2.1 Sequences

A sequence is a sequence of expressions separated by commas. For example

```
> s := 1,4,9,16,25;
                        s := 1, 4, 9, 16, 25

> t := sin,cos,tan;
                         t := sin, cos, tan
```

A sequence of sequences simplifies into one sequence, that is, sequences are *associative*. For example

```
> s := 1,(4,9,16),25;
                        s := 1, 4, 9, 16, 25

> s,s;
                    1, 4, 9, 16, 25, 1, 4, 9, 16, 25
```

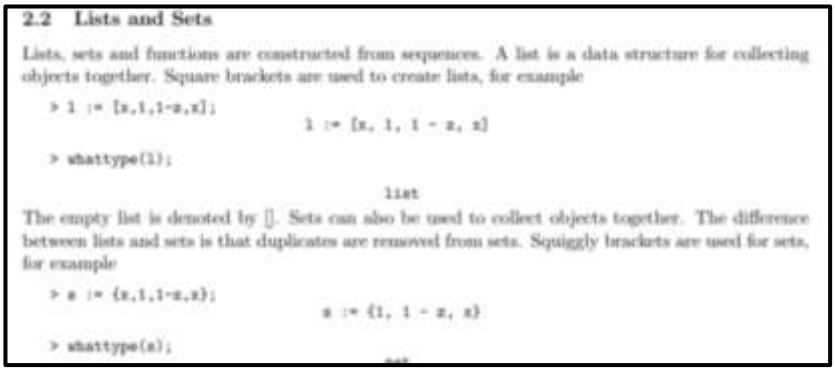
Ciri-ciri dari sequence tidak menggunakan kurung buka dapat berbentuk bilangan atau fungsi matematika. s:= **1,(4,9,16),25** semua bilangan merupakan anggota s secara satu persatutanpa menggunakan kurung. **S,s** artinya membuat anggota himpunan 2 kali banyak dan sama dengan semula.

```
> seq( i^2, i=1..5 );
                              1, 4, 9, 16, 25
> s := NULL; for i from 1 to 5 do s := s, i^2 od;
                                 s :=
                                 s := 1
                               s := 1, 4
                             s := 1, 4, 9
                           s := 1, 4, 9, 16
                         s := 1, 4, 9, 16, 25
```

S:= NULL artinya s sama dengan tidak ada, sedangkan perlu ada tanda **;** untuk mengakhiri statement. Seq(i^2,i=..5) berfungsi untuk fungssi sequence dan kunci pertama fungsi matematika, dan **variabel-variabel bebas** yang diberikan kepada sebuah fungsi matematika (**variabel terikat**).

```
> a := 3*x^3+y*x-11;
                                    3
                           a := 3 x  + y x - 11
> seq( coeff(a,x,i), i=0..degree(a,x) );
                               -11, y, 0, 3
> seq( D(f), f=[sin,cos,tan,exp,ln] );
                                      2
                  cos, - sin, 1 + tan , exp, a -> 1/a
```

a didefinisikan sebagai persamaan $3x^3 + yx - 11$, **coeff(a,x,i)** menentukan koefisien dari a dilihat variabel x, sedangkan fungsi **degree(a,x)** merupakan fungsi derajat dari persamaan a atau bisa kita lihat derajat tertinggi x. D(f) merupakan fungsi turunan dan f merupakan **list**.



2.2 Lists and Sets

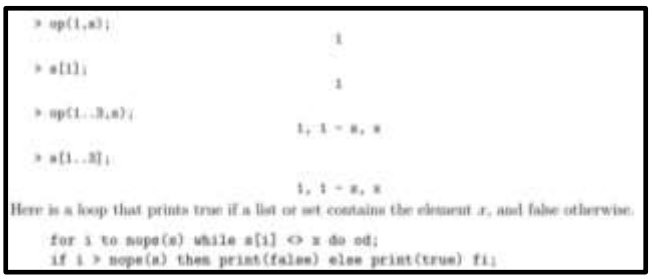
Lists, sets and functions are constructed from sequences. A list is a data structure for collecting objects together. Square brackets are used to create lists, for example

```
> l := [x,1,1-x,x];
                        l := [x, 1, 1 - x, x]

> whattype(l);
                              list
```

The empty list is denoted by []. Sets can also be used to collect objects together. The difference between lists and sets is that duplicates are removed from sets. Squiggly brackets are used for sets, for example

```
> s := {x,1,1-x,x};
                        s := {1, 1 - x, x}

> whattype(s);
```

List, anggota-anggotanya menggunakan tanda [] anggotanya boleh berulang, sedangkan **Sets** menggunakan {} tidak boleh berulang.



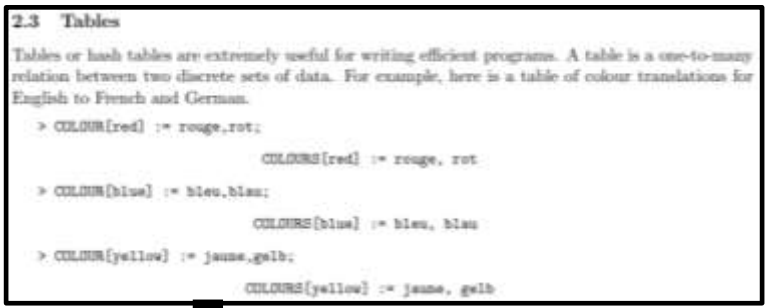
```
> op(1,s);
                                1

> s[1];
                                1

> op(1..3,s);
                             1, 1 - x, x

> s[1..3];
                             1, 1 - x, x
```

Here is a loop that prints true if a list or set contains the element x, and false otherwise.

```
for i to nops(s) while s[i] <> x do od;
if i > nops(s) then print(false) else print(true) fi;
```

```
> t := {u,x,z};
                              t := {x, z, u}

> s union t;
                             {x, z, y, u}
```

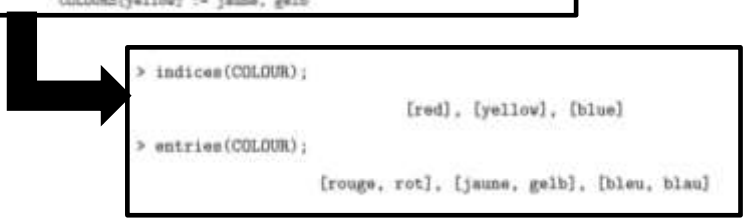
Union untuk menggabungkan anggota s dan t.



2.3 Tables

Tables or hash tables are extremely useful for writing efficient programs. A table is a one-to-many relation between two discrete sets of data. For example, here is a table of colour translations for English to French and German.

```
> COLOUR[red] := rouge,rot;
                    COLOURS[red] := rouge, rot

> COLOUR[blue] := bleu,blau;
                   COLOURS[blue] := bleu, blau

> COLOUR[yellow] := jaune,gelb;
                  COLOURS[yellow] := jaune, gelb
```



```
> indices(COLOUR);
                          [red], [yellow], [blue]

> entries(COLOUR);
                [rouge, rot], [jaune, gelb], [bleu, blau]
```

COLOUR nama tabel memberikan tabel yang terdiri dari kunci utama (**Key**) dan Angota-anggotanya (**Members**). **indices(COLOUR)** menentukan kunci pada tabel COLOR, **entries (COLOUR)** untuk melihat anggota-anggota masing-masing pada **Key**.

$$h := NULL; f := 0; g := nops(e); \textbf{for } i \textbf{ from } 1 \textbf{ to } nops(e) \textbf{ do } f := f + e[i] \textbf{ od}; h := evalf\left(\frac{f}{g}\right)$$

```
> COLOUR[red];
                                    rouge, rot
```

```
> assigned(COLOUR[blue]);
                                      true

> COLOUR[blue] := 'COLOUR[blue]';
                              COLOUR[blue] := COLOUR[blue]

> assigned(COLOUR[blue]);
                                     false

> print(COLOUR);
                               table([
                                   red = (rouge, rot)
                                   yellow = (jaune, gelb)
                               ])
```

Fungsi assigned untuk mengecek keberadaan, dan **fungsi print** melihat semua tabel.

3 Maple Procedures

3.1 Parameters, Local Variables, RETURN, ERROR

A Maple procedure has the following syntax

proc (*nameseq*)
[**local** *nameseq* ;]
[**global** *nameseq* ;]
[**options** *nameseq* ;]
statseq
end

where *nameseq* is a sequence of symbols separated by commas, and *statseq* is a sequence of statements separated by semicolons. Here is a simple procedure which, given x, y, computes $x^2 + y^2$.

```
proc(x,y) x^2 + y^2 end
```

Proc diatas secara sederhana adalah memasukan nilai x dan y dan perlu diperhatikan penutup proc adalah **end**.

```
MEMBER := proc(x,a) local v;
    for v in L do if v = x then RETURN(true) fi od;
    false
end;
```

MEMBER adalah nama fungsi dari prosedur (proc), x dan a adalah variabel masukan di dalam procesude, v merupakan variabel v secara privat (**Local**), v

adalah list L, sets(L) atau Table L, jika v = x maka hasilnya true kembali nilai dengan MEMBER.

```
> proc(x,n) s := 1; for i to n do s := s+x^i od; s end;

  proc(x,n) local s,i; s := 1; for i to n do  s := s+x^i od; s end
```

```
GCD := proc(a,b) local c,d,r;
    c := a;
    d := b;
    while d <> 0 do r := irem(c,d); c := d; d := r od;
    c
end;
```

GCD membuat KPK dengan variabel a dan b.

$GCD := \mathbf{proc}(a, b)$
$\quad \mathbf{local}\ c, d, r;$
$\quad c := a;$
$\quad d := b;$
$\quad \mathbf{while}\ d <> 0\ \mathbf{do}\ r := irem(c, d);\ c := d;\ d := r\ \mathbf{end\ do};$
$\quad c$
$\mathbf{end\ proc}$

> $GCD(21, 15)$

3

> $printlevel := 100$

$printlevel := 100$

> $GCD(21, 15)$

{--> enter GCD, args = 21, 15

$c := 21$
$d := 15$
$r := 6$
$c := 15$
$d := 6$
$r := 3$
$c := 6$
$d := 3$
$r := 0$
$c := 3$
$d := 0$
3

<-- exit GCD (now at top level) = 3}

3

Printlevel:=100 berfungsi untuk melihat proses pada local c, d, dan r.

```
> GCD := proc(a :: integer, b :: integer)
  if b = 0 then a else GCD(b, irem(a, b)) fi
  end;
```

```
GCD := proc(a::integer, b::integer)
    if b = 0 then a else GCD(b, irem(a, b)) end if
end proc
```

```
> GCD(15, 21)
{--> enter GCD, args = 15, 21
{--> enter GCD, args = 21, 15
{--> enter GCD, args = 15, 6
{--> enter GCD, args = 6, 3
{--> enter GCD, args = 3, 0
                                    3
<-- exit GCD (now in GCD) = 3}
                                    3
<-- exit GCD (now in GCD) = 3}
                                    3
<-- exit GCD (now in GCD) = 3}
                                    3
<-- exit GCD (now in GCD) = 3}
                                    3
<-- exit GCD (now at top level) = 3}
                                    3
```

3.3 Arrow Operators

For procedures which compute only a formula, there is an alternative syntax called the arrow syntax. This mimics the syntax for functions often used in algebra. For functions of one parameter the syntax is

symbol -> [**local** *nameseq*;] *expr* .

For 0 or more parameters, parameters are put in parentheses i.e.

(*nameseq*) -> [**local** *nameseq*;] *expr*

The example which computes $x^2 + y^2$ can be written more succinctly as

```
(x,y) -> x^2+y^2;
```

In Maple V Release 2, the syntax has been extended to allow the body of the procedure to be an **if** statement. So you can define piecewise functions. For example

```
x -> if x<0 then 0 elif x<1 then x elif x<2 then 2-x else 0 fi;
```

```
> f1 := proc(x) local g; g := x → x + 1; x·g(x) end;
                    f1 := proc(x) local g; g := x→x + 1; x*g(x) end proc
> f1(2)
```

```
{--> enter f1, args = 2
```

$$g := x \rightarrow x + 1$$

```
{--> enter g, args = 2
```

$$3$$

```
<-- exit g (now in f1) = 3}
```

$$6$$

```
<-- exit f1 (now at top level) = 6}
```

$$6$$

> $f1(3)$

```
{--> enter f1, args = 3
```

$$g := x \rightarrow x + 1$$

```
{--> enter g, args = 3
```

$$4$$

```
<-- exit g (now in f1) = 4}
```

$$12$$

```
<-- exit f1 (now at top level) = 12}
```

$$12$$

Types and Map

type function can be used to code a routine that does different things depending on the ty
e input. For example, the DIFF routine below differentiates expressions which are polynomi
e given variable x.

```
DIFF := proc(a:algebraic,x:name) local u,v;
    if type(a,numeric) then 0
    elif type(a,name) then if a = x then 1 else 0 fi
    elif type(a,'+') then map( DIFF, a, x )
    elif type(a,'*') then u := op(1,a); v := a/u; DIFF(u,x)*v + DIFF(v,x)*u
    elif type(a,anything^integer) then
        u := op(1,a); v := op(2,a); v*DIFF(u,x)*u^(v-1)
    else ERROR('don't know how to differentiate',a)
    fi
end;
```

a berfungsi sebagai aljabar, x sebagai name, numeric berupa angka, type menjelaskan tentang jenis

display(animatecurve(sin(x), x = 0 .. 4*Pi, color = red, thickness = 3, frames = 100), animatecurve(cos(x), x = 0 .. 4*Pi, color = blue, frames = 100));