Programming in Maple.

The if command.

In Maple, the structure of the if command is as follows:

```
if  boolean1 then
    one or more commands seperated by colons or semi-colons
elif boolean2 then
    one or more commands seperated by colons or semi-colons
elif ...
  ...
else
   one or more commands seperated by colons or semi-colons
end if
```

This entire construction is considered as one Maple command (even though there can be many commands inside). To run the command, you have to place a colon or semi-colon after it, and hit return.

There can be as many "elif boolean then ..." as you want (there can also be 0). The output of the "if" command is the last evaluated expression. For example, if you give Maple the following if-command:

```
if x=0 then
     a:=3+5;
     b:=a+7
elif x=1 then
    5
else
   x^2
end if;
```

then, if x=0 the commands a:=3+5;b:=a+7 will be executed, the last expression was a+7 which is 3+5+7=15 so the output of this if-command will be 15 if x=0. If x=1 then the "command" 5 is the only thing that gets "executed", the output will be 5. In all other cases we reach "else" and the output is x^2.

Note that "fi" (that's "if" in reverse order) in Maple is short for "end if".
Remark for Maple 5 and older: In Maple 5 you can only end the "if" statement with "fi" and not with "end if". In Maple 6 and 7, you can choose between "fi" and "end if". So I'm typing "fi" now because that way it works in Maple 5, 6 and 7, whereas typing "end if" would work with Maple 6 and 7.

Example:

```
> x:=-3;
```

$$x := -3$$

```
> if x < 0 then -x else x fi;
```

$$3$$

```
> x:='x';
```

$$x := x$$

```
> if x < 0 then -x else x fi;
Error, cannot evaluate boolean
```

Here an error is produced because the boolean "x < 0" can not be simplified to true or false.

```
> if false then 0 else 1 fi;
```

$$1$$

The command evalb (evaluate boolean) will try to determine if the input is true or false. If it can, the output is true or false. If it can not determine that, then the output is just the input (or is equivalent to the input).

```
> evalb( 4 < 5);
```

$$true$$

```
> evalb(10! - 10^7 > 0);
```

$$false$$

```
> evalb( 17 ); # 17 can never become either "true" or "false" and
  thus an error:
Error, invalid boolean expression
> evalb( y^2+y<0 ); # right now we can not determine "true" or
  "false", but perhaps later in the future we can if some value
  would be assigned to y, so no error message, it just returns
  the input
```

$$y^2 + y < 0$$

```
> if y^2+y < 0 then 0 else 1 fi;
Error, cannot evaluate boolean
```

The following command is:
if "we can evaluate if y^2+y<0 or not" and "y^2+y is indeed < 0" then the output is 0 else 1.

```
> if evalb(y^2+y < 0) = true then 0 else 1 fi;
```

$$1$$

Note that the "then" part is not needed either.

```
> x:=-3;
```

$$x := -3$$

```
> if evalb(x < 0)=true then  x:=-x  fi;
```

$$x := 3$$

```
> x:='x';
```

$$x := x$$

```
> if evalb(x < 0)=true then  x:=-x  fi;
```

now evalb(x<0) was just x<0 and was not equal to true, so the if-command executed no other commands, so nothing happened.

When you have many elif's or if you have many Maple commands then for readability you might want to type the input over more than 1 line. There is one problem though, whenever you hit RETURN, Maple will think it will have to start computing. If you want to move to the next line but you have not yet finished typing the if command, then use shift-RETURN. So to enter an if command like the following, you need shift-returns to go to the next line. Only when the entire command is entered can you use RETURN.

```
> if 3^3 < 2^4 then # type shift return to get to the next line
     a := 2^4 - 3^3;
     b := a^2;
     a + b          # Note: a (semi)-colon is not needed here (*)
  elif 1 > 2 then
     print( "this can not possibly be right" );
     a := 11;
     b := 7+5       # Note: a (semi)-colon is not needed here
  else
     x^2            # Note: a (semi)-colon is not needed here
  fi;
```

$$x^2$$

(*) a (semi)-colon is not needed here because there is no command after a+b. However, it is not an error if you do use one anyway, you could write a+b; here.

The do command.

In Maple, the basic structure of the do command is as follows:

do
 commands seperated by (semi)-colons
end do

Note: "od" (which is "do" in reverse order) is the same as "end do". In Maple 6 and 7 you can use both "od" and "end do", but in Maple 5 you can only use "od", which is what I will use in this Maple worksheet.

The entire construction is considered as one Maple command (even though there can be many commands inside). To run it, end it with : or ; and hit return. In the following construction:

do
 command1;
 command2;
 command3
end do;

what will happen is: command1; command2; command3; command1; command2; command3;.....
So that's an infinite loop. Now in interactive computations that's inconvenient but sometimes acceptable because you can always click on "STOP" (in the graphical version of Maple) or press control-c (in the text version of Maple) to interrupt the computation. But infinite loops are of course terrible inside non-interactive programs.
You can get out of a do-od (do .. end do) loop by the "break" command. The break command will exit the loop. For example:

```
> i := 1;
  S := 0;
  do
     if i < 4 then
        i:=i+1
     elif i < 30 then
        i:=i+10
     else
        print( "i =", i);
        break
     fi;
     S := S + i^2
  od;
  T := S^2;
```

$$i := 1$$
$$S := 0$$
$$i := 2$$
$$S := 4$$
$$i := 3$$
$$S := 13$$
$$i := 4$$
$$S := 29$$
$$i := 14$$
$$S := 225$$
$$i := 24$$
$$S := 801$$
$$i := 34$$
$$S := 1957$$
$$\text{"i ="}, 34$$
$$T := 3829849$$

By the time we reach "else", when i is no longer <30, the break command is executed. When Maple sees "break", it will search for the next "od" or "end do" and continue after that. So the

commands between "break" and "od" will be skipped. As you can see, after we hit "break" the command S:=S+i^2 was skipped and we proceeded straight to T:=S^2;

The command "next" will also skip the rest of the loop, but will not break the loop. So the next command brings you right back to the beginning of the loop.

There are two other ways to use do-od without having an infinite loop. One is by using:

```
for ... do
  ...
od;
```

and the other way is:

```
while boolean do
  ...
od;
```

Note that for and while can also be combined. For example:

```
> S := {seq(seq(i^2+j^2,i=j+1..4),j=1..4)};
```

$$S := \{5, 10, 13, 17, 20, 25\}$$

```
> S:=sort([op(S)]);
```

$$S := [5, 10, 13, 17, 20, 25]$$

```
> T:=0;
  for i in S while i<>17 do
     print( "i =",i);
     T := T+i
  od;
```

$$T := 0$$
$$i := 5$$
$$\text{"i ="}, 5$$
$$T := 5$$
$$i := 10$$
$$\text{"i ="}, 10$$
$$T := 15$$
$$i := 13$$
$$\text{"i ="}, 13$$
$$T := 28$$
$$i := 17$$

The "for i in S" will cause i to go through S. But the "while i<>17" will cause the loop to stop whenever:

"we are back at the beginning of the loop, and the boolean i<>17 evaluates to false"

Half-way trough our list the boolean i<>17 becomes false and the loop stops.

See the help pages for "for" and "while" for other examples. Do this by typing:

```
> ?for
> ?while
```

Procedures

In Maple, a procedure looks like:

```
proc(a1,a2,...,an)
   local v1,v2,..,vk;
   global w1,w2,..,wl;
   options o1,o2,..os;
   body of the procedure
end proc
```

In Maple 5 you must end the procedure with "end" and not with "end proc". In Maple 6 and 7 you can use both "end" as well as "end proc"

Each of the three lines "local .. ", "global ...", "options ..." is optional.
local ...; specifies the local variables. A local variable is different from any variable outside of the procedure even if the name is the same. In most procedures, most or all of the variables used inside the procedure are local. This way one prevents messing up global variables (the variables outside of the procedure). Using global variables can cause some unpleasant surprises for the users, it causes that some variables change value when the procedure is called, which is in most cases not what a user expects.

Nevertheless, it is sometimes convenient to use global variables. To indicate to Maple that a variable is global, so that it does change variables that a user might be using, use global and specify the names of the variables.

It is allowed to use a variable inside a procedure without specifying if it is local or global. In that case Maple will guess if it is local or global, and print its guess on the screen. This is convenient for short procedures.

The body of the procedure consists of one or more commands, that are seperated by a colon or semicolon. It makes no difference if you use colon or semicolon. Note that the purpose of these (semi)-colons is to seperate commands. Therefore it is not necessary to use a (semi)-colon after the last command. However, if you use one anyway even though it was not necessary, then Maple will not complain. However, if you don’t use a (semi)-colon where one is needed, then you will get an error message.

In the line "options ..." some options can be specified. The most commonly used option is "options remember;" of which we will see examples soon.

The input of the procedure (in Maple that is called the "arguments") are specified after proc between ( ). Note that n could be 0, one could have procedures with no arguments. Even if n=0 so you do something like proc() ... then you can still give arguments to that procedure, and those arguments are visibible inside the procedure as follows:

nargs = number of arguments
args[i] = i'th argument.

In most situations, if you want to use the procedure in a convenient way, you will have to give the procedure a name. In Maple, this is done by an assignment:

Defining a procedure is done by one single assignment.

And since an assignment is a Maple command, you have to end this with a colon (nothing printed to the screen) or semi-colon (Maple will print the procedure). So if we want to define a procedure and give it a name, we have to do something that looks like:

```
name_of_the_procedure :=   proc(a1,a2,...,an)
    local v1,v2,..,vk;
    global w1,w2,..,wl;
    options o1,o2,..os;
    first_command;
    second_command;
    ...
    last_command
  end proc:
```

The output if the procedure is always the same as the output of the last command that was executed inside the procedure. For example:

```
> F := proc(x)
    if x=0 then
       infinity
    else
       1/x
    fi
  end;
```

$$F := \mathbf{proc}(x)\ \mathbf{if}\ x = 0\ \mathbf{then}\ \infty\ \mathbf{else}\ 1 / x\ \mathbf{fi}\ \mathbf{end}$$

```
> seq(F(i), i=-2..4);
```

$$\frac{-1}{2}, -1, \infty, 1, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}$$

If x was 0 then the last "command" was infinity, and otherwise the last command was 1/x.
A procedure like this would not make much sense:

```
> F := proc(x)
    if x=0 then
       infinity
    else
```

```
      1/x
    fi;
    x^2
  end;
```

$$F := \textbf{proc}(x)\ \textbf{if}\ x = 0\ \textbf{then}\ \infty\ \textbf{else}\ 1 / x\ \textbf{fi};\ x\text{^}2\ \textbf{end}$$

because the if-command is executed but that has no impact on the output, the last evaluated expression is always x^2 anyways.

The "return" command ends a procedure right away. You can give an argument to return. If you do, that will be the output. If you don't give an argument to return, then the output of the procedure is NULL, which looks like:

```
> NULL; # this prints nothing to the screen
```

Note:
In Maple 5 the syntax is:
RETURN(output)
or
RETURN()
if you want to return "nothing" i.e. return NULL.
This also works in Maple 6 and 7, however, in 6 and 7 you can use:
return output
or just:
return

Example:

```
> F := proc(x)
    if x=0 then RETURN(infinity) fi;
    # from now on we may assume that x<>0 so the following will
  not give an error:
    1/x
  end;
>
```

$$F := \textbf{proc}(x)\ \textbf{if}\ x = 0\ \textbf{then}\ \text{RETURN}(\infty)\ \textbf{fi};\ 1 / x\ \textbf{end}$$

```
> seq(F(i), i=-1..2);
```

$$-1, \infty, 1, \frac{1}{2}$$

Now for some more interesting examples.
Example "subsets"
Input: a set S.
Output: a set T such that every subset of S is an element of T.

Note 1: If S has n elements, then there are precisely 2^n subsets of S. So T has 2^n elements.
Note 2: If S = {} is the empty set, then S has 1 subset, namely {}. So then T must have {} as element. So T={{}}.

Note 3: S is always a subset of S, so S must be an element of T.
Note 4: If e is an element of S, and s is a subset of S then there are two possibilities:

a) e is not in s
b) e is in s.

To write an algorithm for determining all subsets of S, it is sufficient to know only "Note 2" and "Note 4". Notes 1 and 3 were completely unnecessary information. Using just note 2 and 4 gives us the following algorithm:

```
> subsets := proc(S)
     local e,U,T,i; # Note: if you replace "local" by "global"
  then
                    # the procedure will give wrong answers.
     if S = {} then
         T := {{}}   # because of Note 2.
     else
         # At this point we may assume that S has an element.
         # This implies that the following command will not
         # lead to an error:
         e := S[1];
         # Now e is some element of S. The rest of S is:
         U := S minus {e};
         # We can now calculate all subsets of U by recursion.
         # Recursion means: call this procedure from itself. This
         # is OK as long as U is "smaller" in some sense as S,
         # because then the recursion must end at some point.
         T := subsets(U);
         # Notice that the elements of T are precisely those
         # subsets of S that do not have the element e.
         # Now we will use Note 4:
         T := T union {seq(i union {e}, i=T)}
     fi;
     # to check that our procedure is correct we can now do
     # the following:
     if nops(T) <> 2^nops(S) then
        # the following command will interrupt the computation
        # and will print an error message to the screen:
        ERROR( "our procedure is completely wrong" )
     fi;
     # Now that the output T is sufficiently tested, lets return
  it:
     T
  end;
```

subsets := **proc**(*S*)

local *e*, *U*, *T*, *i*;

if $S = \{\ \}$ **then** $T := \{\{\ \}\}$

else

$e := S[1]$;

$U := S$ minus $\{e\}$;

$T :=$ subsets(U);

$T := T$ union $\{$ seq(i union $\{e\}, i = T$) $\}$

fi;

if nops(T) ≠ 2^nops(S) **then** ERROR("our procedure is completely wrong") **fi**;

T

end

```
> S:={x,y,2};
```

$$S := \{2, x, y\}$$

```
> subsets(S);
```

$$\{\{\ \}, \{2, x, y\}, \{2\}, \{x, y\}, \{x\}, \{y\}, \{2, x\}, \{2, y\}\}$$

Same procedure but now shorter and without the test:

```
> subsets := proc(S)
     local e,U,T,i;
     if S = {} then
         {{}}
     else
         e := S[1];
         U := S minus {e};
         T := procname(U);
         T union {seq(i union {e}, i=T)}
     fi
  end;
```

subsets := **proc**(S)

local e, U, T, i;

if $S = \{\ \}$ **then** $\{\{\ \}\}$

else $e := S[1]$; $U := S$ minus $\{e\}$; $T :=$ procname(U); T union $\{$ seq(i union $\{e\}, i = T$) $\}$

fi

end

Note: the name "procname" refers to "name of this procedure". It is convenient to call "procname" instead of "subsets" if you later decide to use a different name for this procedure (then you don't have to make any changes inside the procedure).

```
> {};
```

$$\{\ \}$$

```
> subsets(%);
```

$$\{\{\ \}\}$$

```
> subsets(%);
```

$$\{\{\ \},\{\{\ \}\}\}$$

```
> subsets(%);
```

$$\{\{\ \},\{\{\ \}\},\{\{\{\ \}\}\},\{\{\ \},\{\{\ \}\}\}\}$$

Another example:

Input: an integer n.
Output: the set of all lists, such that:
1) the elements of lists are positive integers
2) the sum of the elements of each list is n.

Remarks:
n < 0 --> output is {}
because no such lists exists, so the solution set is the empty set {}
n = 0 --> output is { [] }
because only the empty list has sum 0, so the empty list is the only element of the solution set.
n > 0. If n>0 and if L is a list in the output, then L[1] must be one of the following numbers: 1,2..,n

These three remarks are sufficient to write a complete algorithm for this problem!

```
> sum_is_n := proc(n)
     local i,j;
     if n<0 then
        {}
     elif n=0 then
        { [] }
     else
        # Note: if i is a number and j is a list j=[a,b,c,..]
        # then    [i,op(j)] = [i,a,b,c,...]
        {seq( seq([i,op(j)], j = procname(n-i)), i=1..n)}
     fi
  end;
```

sum_is_n := **proc**(*n*)

local *i*, *j*;

if $n < 0$ **then** { }

elif $n = 0$ **then** {[ ]}

else $\{\text{seq}(\text{seq}([i, \text{op}(j)], j = \text{procname}(n-i)), i = 1 .. n)\}$

fi

end

```
> sum_is_n(1);
```

$$\{[1]\}$$

```
> sum_is_n(2);
```

$$\{[1, 1], [2]\}$$

```
> sum_is_n(3);
```

$$\{[1, 1, 1], [1, 2], [2, 1], [3]\}$$

```
> sum_is_n(4);
```

$$\{[1, 1, 1, 1], [1, 1, 2], [1, 2, 1], [1, 3], [2, 1, 1], [2, 2], [3, 1], [4]\}$$

Another example:
Fibonnaci:

F(0)=F(1)=1
n>1 -> F(n) = F(n-1) + F(n-2)

```
> F := proc(n)
    if n=0 or n=1 then
       1
    elif n>1 then
       F(n-1) + F(n-2)
    else
       ERROR( "input should be a non-negative integer" )
    fi
  end;
```

F := **proc**(*n*)

if $n = 0$ **or** $n = 1$ **then** 1

elif $1 < n$ **then** $F(n-1) + F(n-2)$

else ERROR("input should be a non-negative integer")

fi

end

```
> F(5);
```

$$8$$

```
> F(24); # This takes some time.
```

$$75025$$

This procedure is slow because many things are re-computed over and over again. To see this, issue the following command:

```
> trace(F);  # this will cause Maple to display the computation
```

$$F$$

```
> F(4);
{--> enter F, args = 4
{--> enter F, args = 3
{--> enter F, args = 2
{--> enter F, args = 1
```

$$1$$

```
<-- exit F (now in F) = 1}
{--> enter F, args = 0
```

$$1$$

```
<-- exit F (now in F) = 1}
                                        2
<-- exit F (now in F) = 2}
{--> enter F, args = 1
                                        1
<-- exit F (now in F) = 1}
                                        3
<-- exit F (now in F) = 3}
{--> enter F, args = 2
{--> enter F, args = 1
                                        1
<-- exit F (now in F) = 1}
{--> enter F, args = 0
                                        1
<-- exit F (now in F) = 1}
                                        2
<-- exit F (now in F) = 2}
                                        5
<-- exit F (now at top level) = 5}
                                        5
```

As you can see (especially if you type F(6) or F(7) or so), the procedure F is entered several times with the same input.
If you type:
F(30)
then F(10) will be computed thousands of times. And F(5) will be called even more often. You can prevent Maple from recomputing the same result by putting "options remember" in the procedure.
With "options remember", whenever an input is encountered that has already been calculated before, Maple will not run the procedure but simply give the same output as it had before.

So if you type F(30) with the following F, then F(10) will not be computed thousands of times, but only once, which should of course be much faster.

```
> F := proc(n)
    options remember;
    if n=0 or n=1 then
       1
    elif n>1 then
       F(n-1) + F(n-2)
    else
       ERROR( "input should be a non-negative integer" )
    fi
  end;
```

F := **proc**(*n*)

option *remember*;

```
    if n = 0 or n = 1 then 1
    elif 1 < n then F(n − 1) + F(n − 2)
    else ERROR("input should be a non-negative integer")
    fi
end
```

```
> F(40); # takes almost no time:
```

165580141

With the previous procedure F, computing F(40) would have taken "forever" (not really forever, it would finish in a finite amount of time but it would take longer than we are willing to wait, which in practise is the same thing as forever).