

Scientific Computing Software. Maple

Lecture 2. Maple programming

**Data structures in Maple. User functions in Maple.
Conditional operator. Loop operator. Iterative commands.
Overview of programming operators. Creating procedures.**

Nasedkina A. A.



Data structures in Maple

Data structures:

- Expression sequence
- Set
- List
- String

Conversion of types and structures

Data structures

- Maple expressions can be composed of both simple objects and complex objects and other Maple expressions . Complex objects include *data structures* .

- Expression sequence
- Set
- List
- String

Other structures:

- Array
- Table
- Matrix , Vector

Data structures: expression sequence

- An **expression sequence** is a group of Maple expressions separated by commas.

$$\frac{0}{0}, \frac{0}{0}, \frac{0}{0}, \frac{0}{0}$$

- The expressions in the sequence may contain objects of different data types. The resulting object type is **exprseq**.

```
> s:=2,x,a+b,`a+b`,sin(x^2),x;
```

```
s := 2, x, a + b, a + b, sin(x^2), x
```

```
> whattype(s);
```

```
exprseq
```

- Accessing sequence elements

```
> s[-3];whattype(%);
```

```
a + b
```

```
symbol
```

```
> s[3];s[4];evalb(s[3]=s[4]);
```

```
a + b
```

```
a + b
```

```
false
```

1	2	3	4	5	6
$s := 2, x, a + b, a + b, \sin(x^2), x$					
-6	-5	-4	-3	-2	-1

Data structures: expression sequence

- Accessing multiple elements of a sequence

```
> s[2..4];
```

$x, a + b, a + b$

- Adding elements to a sequence and incrementing a sequence

```
> t:=s[2..4],8;
```

$t := x, 5 + b, a + b, 8$

```
> t:=t,s[1],10;
```

$t := x, 5 + b, a + b, 8, 2, 10$

- Empty sequence

```
> s:=NULL;
```

$s :=$

- Sequence generation operator \$

expr \$ name = initial .. final

```
> $1..5;
```

$1, 2, 3, 4, 5$

```
> s := i2 $ i = 2..6;
```

$s := 4, 9, 16, 25, 36$

Data structures: set

- A **set** is a group of Maple expressions written in *curly* braces separated by commas .

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

This object has all the features of a mathematical set:

- each element is stored in a single copy
- the specified order of elements is not stored

The expressions in a set can contain objects of different data types. The resulting object type of a "set" is called a **set**.

```
> m:={2,x,x,b,a,a};
```

```
m:= {2, x, a, b}
```

```
> whattype(m);
```

```
set
```

Data structures: set

- Accessing set elements

```
> m:={2,x,x,b,a,a};
```

$m := \{2, x, a, b\}$

```
> m[3];
```

a

```
> m[5];
```

Error, invalid subscript selector

- Empty set

```
> p:={};
```

$p := \{ \}$

- Set algebra operations

```
> S1:={a,b,c};
```

$S1 := \{a, b, c\}$

```
> S2:={b,c,d,e,d};
```

$S2 := \{b, c, d, e\}$

```
> SU:=S1 union S2;
```

$SU := \{a, b, c, d, e\}$

```
> SU := S1 U S2;
```

$SU := \{a, b, c, d, e\}$

Data structures: set

- Set algebra operations

> $S1 := \{a, b, c\};$

$S1 := \{a, b, c\}$

> $S2 := \{b, c, d, e, d\};$

$S2 := \{b, c, d, e\}$

> $SI := S1 \text{ intersect } S2;$

$SI := \{b, c\}$

> $SI := SI \cap S2;$

$SI := \{b, c\}$

> $Sm1 := SI \setminus S2;$

$Sm1 := \{a\}$

> $Sm2 := S2 \text{ minus } S1;$

$Sm2 := \{d, e\}$

- Adding elements to a set and incrementing a set

> $S := SI \cup \{1, x\}$

$S := \{1, a, b, c, x\}$

> $S := S \cup \{t\}$

$S := \{1, a, b, c, t, x\}$

Data structures: list

- A **List** is a group of Maple expressions written in *square* brackets separated by commas .

$[\text{ } , \text{ } , \text{ } , \text{ }]$

This object has features that are opposite to the set:

- all repeating elements are stored
- the specified order of elements is stored

The expressions in a list can contain objects of different data types. The resulting "set" object type is called **list** .

```
> L := [2, x, x, b, a, a] ;
```

```
L := [2, x, x, b, a, a]
```

```
> whattype(L) ;
```

```
list
```

- Empty list

```
> Lp := [] ;
```

```
Lp := [ ]
```

Data structures: list

- Accessing list elements

```
> L := [2, x, x, b, a, a];
```

```
L := [2, x, x, b, a, a]
```

```
> L[5];
```

```
a
```

```
> L[3..5];
```

```
[x, b, a]
```

- Adding items to a list and growing the list

```
> M := [op(L), c];
```

```
M := [2, x, x, b, a, a, c]
```

```
> M := [op(M), sin(x)];
```

```
M := [2, x, x, b, a, a, c, sin(x)]
```

- Sorting a list: **sort(L, f)** sorts a list **L**, where optional argument **f** specifies the sort order

```
> s := [2, 1, 3]
```

```
> sort(s);
```

default: sort in ascending order

```
[1, 2, 3]
```

```
> sort(s, `>`);
```

sort in descending order

```
[3, 2, 1]
```

Count and extract elements from lists and sets

nops (x) – returns the number of elements in the object **x**

op(i,e) – extracts the element at position **i** in object **e**

op(i .. j,e) – extracts elements located at positions **i** by **j** in object **e** (returns subsequence elements)

op(e) – retrieves all elements of object **e** (returns subsequence elements)

```
> s:={a,b,4,-1,f,c,f};nops(s);
```

```
s := {-1, 4, a, f, b, c}
```

```
6
```

```
> op(s);
```

```
-1, 4, c, a, b, f
```

```
> op(3,s);op(1..3,s);
```

```
a
```

```
-1, 4, a
```

Commands for counting and extracting elements

> `s:=[a,b,4,-1,f,c,f];nops(s);`

`s:=[a,b,4,-1,f,c,f]`

7

> `op(4,s);op(1..4,s);`

-1

`a,b,4,-1`

- **nops** and **op** commands can be used for other objects as well:

> `g:=x3+3x2+5x-6`

`g:=x3+3x2+5x-6`

> `nops(g)`

4

> `nops(op(1,g))`

2

> `op(1,op(2,g))`

3

Data structures : string

cat(a, b, c) – concatenation of strings and expressions

> *cat*("program", "ming")

"programming"

> *cat*(*a*, *b*); *cat*(*a*, 1 ..10)

ab

a1, a2, a3, a4, a5, a6, a7, a8, a9, a10

Useful commands from the **StringTools** package

Length(s) – returns the length of the string **s**

Split(s) – returns a list of strings that make up individual words in the string **s** (by default, spaces are analyzed). You can specify the separator type: **Split(s, sep)**

Stem(s) – returns a string containing the stem of the word **s** (English)

> *s* := "Impressive string":

> *with*(StringTools):

> *Length*(*s*) ;

17

> *m* := *Split*(*s*) ;

m := ["Impressive", "string"]

> *Stem*(*m*[1]) ;

"impress"

Using Data Structures in Maple Commands

The set and list data structures are widely used as arguments to various commands. Some commands accept either structure if the order of the elements is unimportant. If the order of the elements in the structure must be taken into account, the list structure must be used.

```
> solve({x+y=10,x-y=5},{x,y});
```

$$\left\{y = \frac{5}{2}, x = \frac{15}{2}\right\}$$

```
> solve({x+y=10,x-y=5},[x,y]);
```

$$\left[\left[x = \frac{15}{2}, y = \frac{5}{2}\right]\right]$$

```
> f:=x-10*y:eval(f,[x=1,y=2]);
```

-19

```
> eval(f,{x=1,y=2});
```

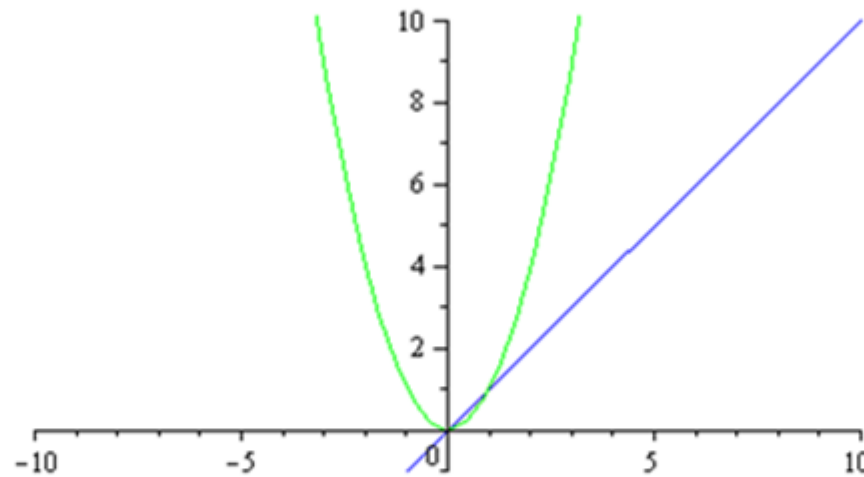
-19

Using Data Structures in Maple Commands :

Example for the plot command

You can always check the requirements for the command arguments and their types in the help information for a given command. Look at the example below for the plot command:

```
> plot([x,x^2],x=-10..10,  
      -1..10,scaling=CONSTRAINED,color=[blue,green]);
```



```
> plot({x,x^2},x=-10..10,-  
      1..10,scaling=CONSTRAINED,color={blue,green});  
Error, (in plot) invalid color specification: {blue, green}
```

Type conversion and structures: convert

- **convert** – converts an expression to another form, can be used to convert the type of an expression or data structure

convert (expr , form, optional arguments)

> *convert*([1, 2, 3], *vector*)

$\begin{bmatrix} 1 & 2 & 3 \end{bmatrix}$

> *convert*($x^2 + 2 \cdot x \cdot y + y^2$, *string*)

"x^2+2*x*y+y^2"

> *convert*(9, *binary*)

1001

> *convert*(1.23456, *rational*)

$\frac{3858}{3125}$

> *convert*($\frac{1}{8}$, *float*, 3)

0.125

> *convert*([1, 2, 3, 4], '+')

10

Programming operators

- Logical values in Maple
- Conditional operator
- Loop operator

Boolean algebra in Maple

- **Two-valued (binary) logic** : true and false (1 and 0) .
- **Three-valued (ternary) logic** : "true", "false" and "undefined" .
- Maple implements three - valued logic with logical values
 - **true**
 - **false**
 - **FAIL** (truth undefined)
- To write a logical expression in Maple you can use:
 - Boolean values **true** , **false** , **FAIL**
 - Relational operators **<**, **>**, **<=**, **>=**, **=**, **<>**
 - **Logical operators** **not** (NOT) , **and** (AND) , **or** (OR) , **xor** (exclusive OR) , **implies** (implication)
- The evaluation of a logical expression occurs automatically if it:
 - contains a logical operator
 - is in the **if condition** of the conditional operator
 - is in the **while condition** of the loop statement
- Otherwise use **evalb** command to evaluate boolean expression

Truth tables for logical operators not, and, or, xor

not	
a	not a
true	false
false	true
FAIL	FAIL

or				
a or b		b		
		true	false	FAIL
a	true	true	true	true
	false	true	false	FAIL
	FAIL	true	FAIL	FAIL

and				
a and b		b		
		true	false	FAIL
a	true	true	false	FAIL
	false	false	false	false
	FAIL	FAIL	false	FAIL

xor				
a xor b		b		
		true	false	FAIL
a	true	false	true	FAIL
	false	true	false	FAIL
	FAIL	FAIL	FAIL	FAIL

Conditional operator: types

- In Maple, you can implement the basic "branching" programming structure using various types *of conditional statements* .

Types of branching :

- bypass (if-then)
if ... then ... end if
- branching (if-then-else)
if ... then ... else ... end if
- multiple choice (if-then-else-if ...)
if ... then ... elif ... then ...<...> end if
- The *condition* must be a logical expression, and the *expressions* must be one or more Maple expressions . The then branch is executed if the condition evaluates to true. Otherwise (false or FAIL), the else branch (elif) is executed.

Conditional operator: simple forms

- **if condition then expressions end if;**
- **if condition then expressions1 else expressions2 end if;**

Commands for displaying messages on the screen:

- **print(x)** – prints the value of the variable **x**
- **print("xxx")** – prints the string "xxx"
- **printf (" bbb %a cc",x)** – prints a message to the screen, where the value of the variable **x** is substituted for %a in the string "bbb ... cc"

```
> a:=3; b:=5;
```

```
> if (a>b) then a else b end if;
```

```
5
```

```
> x := 3 > 2 : print(x);
```

```
2 < 3
```

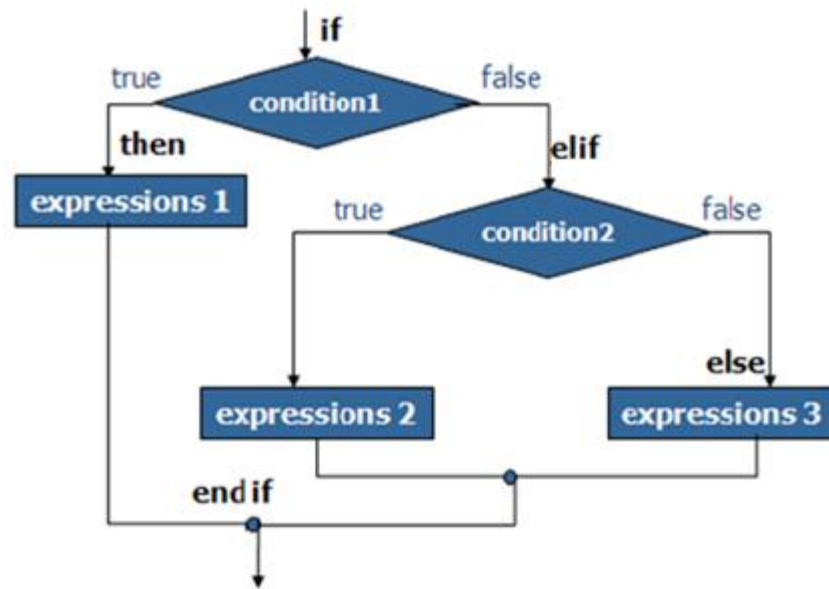
```
> if type(x, boolean)  
  then print ("x – логическое выражение");  
  printf ("%a - булево выражение", x) end if;
```

```
"x - логическое выражение"
```

```
2 < 3 – булево выражение
```

Conditional Operator: Complete Syntax

- **if** *condition1* **then** *expressions1* **elif** *condition2* **then** *expressions2* **elif** *condition3* **then** *expressions3* ... **else** *expressionsN* **end if**;



> $x := 7$;

> **if** $x < 2$ **then** $x := x^2$; *print*(x) **elif** $x = 2$ **then** $x := x^3$; *print*(x) **else** $x := x^4$; *print*(x) **end if**

$x := 2401$

2401

Loop operator: types

- Maple can implement the basic "repetition" programming structure using various types of preconditional *loop operators* . Postconditional loops (until) are not implemented in Maple 11.

Types of loop operators in Maple

- while

while ... do...end do

- loops with parameter (for/from, for/in)

for...from...to...end do

for...in...do...end do

- mixed loops (for/from while, for/in while)

for...from...while...do...end do

for...in...while...do...end do

- As a *condition* **while** A logical expression must be specified, with one or more Maple expressions as *expressions* . The expressions located between **the do** and **end do** keywords constitute the loop body.

while operator

- **while** *condition* **do** *expressions* **end do**;

As long as the condition test evaluates to true, the Maple expressions that make up the loop body are executed. As soon as the condition evaluates to false or FAIL, Maple exits the loop.

Example. Even numbers from 8 to 2 in descending order

```
[> j := 8;  
> while j ≥ 2 do if irem(j, 2) = 0 then print(j) end if; j := j - 1 end do;
```

8	>	1-я итерация
j := 7		
j := 6	—	2-я итерация
6	>	3-я итерация
j := 5		
j := 4	—	4-я итерация
4	>	5-я итерация
j := 3		
j := 2	—	6-итерация
2	>	7-я итерация
j := 1		

for/from operator

- **for** *counter* **from** *initial* **by** *increment* **to** *final* **do** *expressions* **end do**;

Short form: **for** *counter* **to** *final* **do** *expressions* **end do**;

Default values:

initial = 1, increment = 1, final = ∞

```
> for j from 4 to 8 do print(j) end do
```

4

5

6

7

8

Example. Even numbers from 8 to 2 in descending order

```
> for j from 8 to 2 by -1 do if irem(j, 2) = 0 then print(j) end if end do
```

8

6

4

2

```
> j;
```

1

for/in operator

- **for** *variable* **in** *expression* **do** *expressions* **end do**;

The loop parameter sequentially takes the values of the elements of a Maple expression (e.g., terms in a sum, factors in a product, characters in a string, elements of a list, a set, etc.)

```
> f := x^2 + 3 x + 1/x :
```

```
> for s in f do s; end do;
```

x^2

$3 x$

$\frac{1}{x}$

```
> s := "ЦИКЛ" :
```

```
> for i in s do print(i) end do
```

"Ц"

"И"

"К"

"Л"

for/in loop operator: examples

Example. Calculating the sum of perfect squares from 1 to 100.

> $SQR := \{i^2 \mid i = 1 \dots 10\};$

$SQR := \{1, 4, 9, 16, 25, 36, 49, 64, 81, 100\}$

> $s := 0;$

> **for** i **in** SQR **do** $s := s + i$ **end do**

$s := 1$

$s := 5$

$s := 14$

$s := 30$

$s := 55$

$s := 91$

$s := 140$

$s := 204$

$s := 285$

$s := 385$

> $convert(SQR, '+')$

385

Full syntax of for/from...while, for/in...while mixed loop operators

- **for** *counter* **from** *initial* **by** *increment* **to** *final* **while** *condition* **do** *expressions* **end do**;
 - **for** *variable* **in** *expression* **while** *condition* **do** *expressions* **end do**;
-
- **for / from** loop: if the final value of the loop parameter is specified (**to** *final*) and there is a condition check (**while** *condition*), then first the current value of the loop parameter is compared with the final value , after that the while condition is checked . If the while condition becomes false, Maple exits the loop.
 - **for / in** loop: if the loop statement includes a conditional test (**while** *condition*), then for each parameter value Maple checks the condition first, then the loop body is executed. The parameter sequentially takes on the value of the Maple expression elements until it encounters an element for which the condition becomes false. However, the loop may be terminated before all elements of the Maple expression are exhausted (see example below).

Mixed loops: examples

```
> for x from 1 by 2 to 7 while x < 6 do x end do
```

```
1
```

```
3
```

```
5
```

```
> x
```

```
7
```

Print all numerical values

```
> s := [2, a, b, 3.5, c]:
```

```
> for x in s while type(x, numeric) do x end do
```

```
2
```

```
> x
```

```
a
```

```
> s := [2, 3, 4, 3.5, c]:
```

```
> for x in s while type(x, numeric) do x : end do
```

```
2
```

```
3
```

```
4
```

```
3.5
```

```
> x
```

```
c
```

Find the sequence of the first 10 prime numbers

```
> COUNT:=0: PRIMES:=NULL:
```

```
> for n from 1 while COUNT<10 do
```

```
if isprime(n) then COUNT:=COUNT+1; PRIMES:=PRIMES,n end if  
end do;
```

```
> PRIMES;
```

```
2, 3, 5, 7, 11, 13, 17, 19, 23, 29
```

```
> COUNT;
```

Overview of programming operators

All programming operators in Maple 11.0

List of Maple statements

Description

- See ?topic for any of the following topics:

<u>!</u>	<u>#</u>	<u>assignment</u>	<u>break</u>	<u>by</u>
<u>catch</u>	<u>description</u>	<u>do</u>	<u>done</u>	<u>elif</u>
<u>else</u>	<u>empty</u>	<u>end do</u>	<u>end if</u>	<u>error</u>
<u>export</u>	<u>finally</u>	<u>for</u>	<u>from</u>	<u>function</u>
<u>if</u>	<u>in</u>	<u>local</u>	<u>module</u>	<u>next</u>
<u>option</u>	<u>proc</u>	<u>quit</u>	<u>read</u>	<u>restart</u>
<u>return</u>	<u>save</u>	<u>separator</u>	<u>stop</u>	<u>then</u>
<u>to</u>	<u>try</u>	<u>use</u>	<u>uses</u>	<u>while</u>

Example

- **break** – loop interrupt operator

```
L := [ 1, 2, "abc", "a", 7.0, ∞ ] : for x in L do if type(x, 'string') then  
    print(x); break end if end do  
    "abc"
```

Maple iterative commands

- **seq**
- **add** and **mul**
- **select, remove, selectremove**
- **map**
- **zip**

Iterative commands : seq

- Sequence generation operator

expr \$ name = initial .. final ;

- Creating a sequence

seq (expression, name = initial .. final, step); # analogous to a **for/from** loop

seq (expression, name in expression); # similar to the **for/in** loop

> \$ 2..5;

2, 3, 4, 5

> a[i] \$ i = 1..3;

a_1, a_2, a_3

> seq(i^2, i=1..5);

1, 4, 9, 16, 25

> seq(i, i="a".."f");

"a", "b", "c", "d", "e", "f"

> seq(x[i], i=1..5);

x_1, x_2, x_3, x_4, x_5

> seq(u, u in [Pi/4, Pi^2/2, 1/Pi]);

$\frac{\pi}{4}, \frac{\pi^2}{2}, \frac{1}{\pi}$

Iterative commands : add, mul

- Calculating the sum

add(expression, name = initial .. final); # similar to a **for/from** loop

add(expression, name in expression); # similar to the **for/in** loop

- Calculating the product

mul (expression, name = initial .. final); # similar to **for/from** loop

mul (expression, name in expression); # similar to **for/in** loop

```
> add( i^2, i=1..5 );
```

55

```
> add(u, u in [Pi/4, Pi/2, Pi/6]);
```

$\frac{11}{12} \pi$

```
> mul(i, i=1..5 );
```

120

```
> mul(u, u in [Pi/4, Pi^2/2, 1/Pi]);
```

$\frac{1}{8} \pi^2$

Iterative commands : select, remove

Extraction and deletion commands (analogous to the **for/in** loop)

- **select(proc,expression)** - extracts those operands from the expression that satisfy the Boolean function (procedure) *proc* .
- **remove(proc,expression)** - extracts those operands from the expression *expression* that do not satisfy the Boolean function (procedure) *proc* .
- **selectremove (proc,expression)** - first extracts those operands from the expression that satisfy the Boolean function (procedure) *proc* , and then those that do not satisfy it.
- Each command returns an object of the same type as the original expression.

Извлечем простые числа из списка

```
> integers := [$10..20];
```

```
integers := [10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
```

```
> select(isprime, integers);
```

```
[11, 13, 17, 19]
```

```
> remove(isprime, integers);
```

```
[10, 12, 14, 15, 16, 18, 20]
```

```
> selectremove(isprime, integers);
```

```
[11, 13, 17, 19], [10, 12, 14, 15, 16, 18, 20]
```

```
[> f := x + y + 2  
f := x + y + 2  
[> select(type, f, symbol)  
x + y]
```

Iterative commands: map

- Applying a command or procedure to each member of an expression
map(proc, expression); # similar to the **for/in** loop

```
> map(f, {a,b,c});
```

$\{f(a), f(b), f(c)\}$

```
> map(f, x + y*z);
```

$f(x) + f(yz)$

```
> map(f, y*z);
```

$f(y)f(z)$

```
> map(sqrt, a+b);
```

$\sqrt{a} + \sqrt{b}$

```
> f := x → sin(x + 3);
```

```
> map(f, y*z);
```

$\sin(y + 3) \sin(z + 3)$

Factorize all composite numbers

```
> integers := [$10..20]: comp:=remove(isprime,integers);  
map(ifactor,comp);
```

$comp := [10, 12, 14, 15, 16, 18, 20]$

$[(2)(5), (2)^2(3), (2)(7), (3)(5), (2)^4, (2)(3)^2, (2)^2(5)]$

Iterative commands: zip

- Applying a command or procedure component-wise to a pair of elements of two **zip** lists or vectors . The command combines two lists or vectors into one.

- **zip(proc,a,b)**

zip(proc,a,b,fill)

- The length of the resulting list or vector is min(m,n), where m and n are the lengths of the original lists.
- *the fill* parameter used to fill the elements of the resulting list in the case of lists of different lengths, while the resulting list will have a length of max(m, n)

```
> zip( `+`, [1, 2, 3], [4, 5, 6])  
[5, 7, 9]  
  
> zip( `+`, [1, 2, 3, 4, 5], [4, 5, 6], 10)  
[5, 7, 9, 14, 15]  
  
> zip( f, [a, b, c], [d, e])  
[f(a, d), f(b, e)]  
  
> zip( (x, y) → x + 100 y, [1, 2], [4, 5, 6])  
[401, 502]
```

User-defined procedures in Maple

- Functional operators
- Simple procedures
- General information about the Maple procedure
- Full procedure syntax
- Outputting procedure code to the screen
- Local and global variables
- Working with procedure arguments
- Recursive procedures
- Displaying error messages, exiting the procedure

Functional operator

- A **procedure** is a subroutine consisting of Maple commands and expressions . Essentially, a procedure is also a Maple command , but not a built-in one, but one created by the user. A simplified concept of a procedure is a functional operator.

General syntax :

- **f := var -> result**
- **f:=(var1,var2,...)->result**

Using the functional operator, you can define functions of one or more variables, as well as vector functions :

- $f := x \rightarrow x^2$ – function of one variable $R \rightarrow R$
- $f := (x, y) \rightarrow x^2 + y^2$ – function of two variables $R^2 \rightarrow R$
- $f := x \rightarrow (2 * x, 3 * x^4)$ – vector function of one variable $R \rightarrow R^2$
- $f := (x, y, z) \rightarrow (x * y, y * z)$ – vector function of three variables $R^3 \rightarrow R^2$

Functional operator: examples

Creating a functional operator from an expression and its variables

- **unapply (expression, x,y ...,); unapply (expression, list of variables);**

```
> g := (x,y) -> sin(x)*cos(y) + x*y;
```

$$g := (x, y) \rightarrow \sin(x) \cos(y) + xy$$

```
> g(Pi/2,Pi);
```

$$-1 + \frac{1}{2} \pi^2$$

```
> p:=x-> if x<0 then -x; else sqrt(x); end if: p(-2); p(2);
```

$$\sqrt{2}$$

```
> p := x^2 + sin(x) + 1; f := unapply(p,x); f(Pi/6);
```

$$p := x^2 + \sin(x) + 1$$

$$f := x \rightarrow x^2 + \sin(x) + 1$$

$$\frac{1}{36} \pi^2 + \frac{3}{2}$$

```
> q:=a-b;g:=unapply(q,a,b);
```

$$q := a - b$$

$$g := (a, b) \rightarrow a - b$$

```
> q :=x^2+y^3+1; g:=unapply(q,[x,y]); g(2,3);
```

$$q := x^2 + y^3 + 1$$

$$g := (x, y) \rightarrow x^2 + y^3 + 1$$

General information about the procedure, short procedure syntax

- A procedure is a custom Maple command .
- A procedure may have no arguments, one argument, or several arguments (separated by commas).
- The entire procedure description must be contained within a single execution group . To move to a new line, press **Shift + Enter**.

Short procedure syntax

```
• proc_name := proc ( parameterSequence )  
  statementSequence ;  
end proc ;
```

Words highlighted in blue may be missing.

- *proc_name* – procedure name
- **proc ... end proc** – keywords that indicate the beginning and end of a procedure
- *statementSequence* – a sequence of expressions implementing the procedure *body* . The value of the last expression (command) in this sequence or the value of the expressions specified after the **return** command is returned .

Define simple procedure without arguments

```
> ex := proc ( )  
   $\sqrt{2}$  ;  
  end proc ;  
  
ex := proc ( )  $\sqrt{2}$  end proc
```

Call and execute procedure

```
> ex ( ) ;
```

$\sqrt{2}$

Example of a simple procedure with a parameter

- The procedure returns the sequence of the first n prime numbers

```
> p:=proc(n)
  local i,COUNT, PRIMES;
  COUNT:=0: PRIMES:=NULL:
  for i from 1 while COUNT<n do
    if isprime(i) then COUNT:=COUNT+1; PRIMES:=PRIMES,i; end if
  end do;
  PRIMES;
end proc;
```

```
p:=proc(n)
  local i, COUNT, PRIMES;
  COUNT:=0;
  PRIMES:=NULL;
  for i while COUNT < n do if isprime(i) then COUNT:=COUNT+1; PRIMES:=PRIMES,i end if end do;
  PRIMES
```

end proc

```
> p(5)
```

2, 3, 5, 7, 11

```
> p(10)
```

2, 3, 5, 7, 11, 13, 17, 19, 23, 29

Full procedure syntax

- *proc_name* := **proc** (*parameterSequence* :: *type*) :: *returnType* ;
local *localSequence* ;
global *globalSequence* ;
option *optionSequence* ;
description *descriptionSequence* ;
uses *usesSequence* ;
statementSequence ;
end proc;

Words highlighted in blue may be missing.

- *parameterSequence* – a sequence of formal parameters (*arguments*) of a procedure. Each formal parameter can be assigned (*declared*) a specific data type using the double-colon operator :: followed by the data type name (*type*). When calling the procedure, if at least one parameter does not match its declared type, a system error message will be returned. Default values can also be assigned to parameters.
- *returnType* – an optional expected return type for the procedure. By default, if the return type does not match the expected type, no error occurs.

Full procedure syntax (continued)

- **local** is a special keyword used to describe a sequence of local variables (*localSequence*). *Local variables* are variables that are used only within a given procedure. Local variables can be assigned with returned type in the form `:: returnType` .
- **global** is a special keyword used to describe a sequence of global variables (*globalSequence*). *Global variables* are variables that are not local but are also used by a given procedure. Global variables are declared when values will be assigned to these variables within a procedure. The type of global variables cannot be specified within a procedure.
- **option** is a keyword used to describe the sequence of options for the *optionSequence* procedure . Options are represented by special words, such as **arrow** . (arrow), **builtin** (built-in procedure), **operator** (operator), **remember** (option for efficient operation of recursive procedures) , **Copyright ...** and some others.
- **description** is a keyword followed by *descriptionSequence* comments about the procedure's purpose and operation (one or more lines). Unlike comments specified with the # symbol, this information is displayed on the screen when printing the procedure code.
- **uses** – a keyword for describing a *usesSequence* of related names and modules that will be used in the procedure body. It can be used to include packages , for example: **uses StringTools ;**

Examples of procedures : required arguments, number of arguments when calling a procedure, declaring argument types

Пример процедуры с аргументами

> `p := proc(a, b) a + b; a - b : end proc:`

Возвращается значение последней команды в теле процедуры

> `p(1, 2);`

-1

> `p(2, 1);`

1

Избыточное количество аргументов при вызове процедуры – ошибки не происходит:

> `p(1, 2, 3);`

-1

Недостаточное количество аргументов при вызове процедуры – сообщение об ошибке:

> `p(2);`

Error, invalid input: p uses a 2nd argument, b, which is missing

Пример процедуры с декларированием типов аргументов

> `f := proc(a :: integer, b) a + b end proc:`

> `f(2, 3);`

5

> `f(2.5, 3);`

Error, invalid input: f expects its 1st argument, a, to be of type integer, but received 2.5

Examples of procedures : declaring argument types, multiple data types, default values

Пример процедуры с декларированием нескольких типов данных для аргументов

```
f := proc (a :: {integer, float}, b :: integer) ab end  
      proc:
```

```
> f(2, 3);
```

8

```
> f(2, 2.5);
```

Error, invalid input: f expects its 2nd argument, b, to be of type integer, but received 2.5

```
> f(2.5, 2);
```

6.25

Пример процедуры с декларированием типов аргументов и их значений по умолчанию

```
> f := proc (a :: integer := 10, b :: integer := 100.1) a + b end proc:
```

```
> f(3);
```

103.1

```
> f(3, 4);
```

7

```
> f(3.5, 4.5)
```

110.1

Examples of procedures : option, description, uses

*Пример процедуры с опциями функционального оператора (использование **option**)*

```
> f := proc(x) option operator, arrow; x^2-1 end proc;
```

Процедура задает функциональный оператор, ее запись эквивалента команде:

```
> f := x -> x^2-1;
```

*Пример процедуры с комментарием о ее назначении (использование **description**)*

```
> lc := proc( s, u, t, v )
```

```
    description "forms a linear combination of the  
arguments";
```

```
    s * u + t * v
```

```
end proc;
```

Вывод на экран комментариев к процедуре

```
> Describe(lc) ;
```

```
# forms a linear combination of the arguments lc( s, u, t, v )
```

*Пример процедуры с подключением пакета (использование **uses**)*

```
> LastWord:=proc(s::string)
```

```
uses StringTools;
```

```
Split(s);%[-1];
```

```
end proc;
```

```
> LastWord("Hello world!");
```

```
"world!"
```

```
> LastWord(a) ;
```

```
Error, invalid input: LastWord expects its 1st argument, s, to  
be of type string, but received a
```

Returning multiple values from a procedure

The procedure `PrimesAtInterval` searches for all prime numbers at the given interval and counts them. It returns the number of these values and the prime numbers as a list

```
> PrimesAtInterval := proc(a, b :: integer)
  local COUNT, PRIMES, n;
  COUNT := 0 : PRIMES := [ ];
  for n from a to b do
  if isprime(n) then COUNT := COUNT + 1; PRIMES := [op(PRIMES), n]
    end if
  end do:
  COUNT, PRIMES;
end proc:
```

```
> PrimesAtInterval(100, 200);
21, [101, 103, 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167,
173, 179, 181, 191, 193, 197, 199]
```

```
> (n, s) := PrimesAtInterval(100, 200) :
```

```
> n,
21
```

```
> s,
[101, 103, 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173,
179, 181, 191, 193, 197, 199]
```

Outputting procedure code to the screen

Displaying the user procedure code on the screen

- **print**(*proc_name*);
- **eval** (*proc_name*);

Пример пользовательской процедуры

```
> lc := proc( s, u, t, v )  
      description "forms a linear combination  
                  of the arguments";  
      s * u + t * v  
end proc;  
> eval(lc);
```

```
proc(s, u, t, v)  
  description  
    "form a linear combination of the arguments,"  
    s*u + t*v  
end proc
```

Outputting procedure code to the screen (continued)

Displaying procedure code from the Maple library on the screen
(except for built-in procedures, with the builtin option)

- **interface('verboseproc'=2): print(*proc_name*);**
- **interface('verboseproc'=2): eval (*proc_name*);**

Пример процедуры из библиотеки

```
> print(issqr) ;  
      proc(n) ... end proc  
  
> interface('verboseproc' = 2) :  
> print(issqr) ;  
      proc(n)  
      option  
      Copyright (c) 1990 by the University of  
      Waterloo. All rights reserved. ;  
      if type(n, integer) then  
          evalb(issqr(n)^2 = n)  
      elif type(n, numeric) then  
          false  
      else  
          'issqr(n)'  
      end if  
end proc
```

Пример встроенной процедуры

```
> interface('verboseproc' = 2) :  
> print(conjugate) ;  
      proc( )  
      option builtin = conjugate;  
  
end proc
```

Local and global variables: declaring local variables

Пример процедуры с локальными переменными

Процедура *maximum*, находит максимум из заданного списка целых чисел.

```
> maximum := proc (s::(list(integer)))
```

```
local max, i;
```

```
max := s[1];
```

```
for i to nops(s) do
```

```
    if s[i]>max then
```

```
        max := s[i]
```

```
    end if;
```

```
end do;
```

```
max;
```

```
end proc;
```

```
> maximum([4,1,8,-100]);
```

8

```
> maximum(4,1,8,-100);
```

Error, invalid input: maximum expects its 1st argument, s, to be of type list(integer), but received 4

```
> maximum([4,1,8,z]);
```

Error, invalid input: maximum expects its 1st argument, s, to be of type list(integer), but received [4, 1, 8, z]

Local and global variables: declaring local variables

warning for undeclared local variables max and i

>

```
maximum := proc (s :: list(integer))  
  max := s[1];  
  for i from 1 to nops(s) do  
    if s[i] > max then  
      max := s[i]  
    end if  
  end do;  
  max;  
end proc;
```

Warning, `max` is implicitly declared local to procedure
`maximum`

Warning, `i` is implicitly declared local to procedure `maximum`

The difference between local and global variables

Declare my_pi as local variable

```
> my_pi:=3.14:
> CircleArea1:=proc(r)
  local my_pi;
  my_pi:=evalf(Pi,10);
  my_pi*r^2;
end proc:
> CircleArea1(5);
      78.53981635

> my_pi; r:=5: my_pi*r^2;
      3.14
      78.50
```

Declare my_pi as global variable

```
> my_pi:=3.14:
> CircleArea2:=proc(r)
  global my_pi;
  my_pi:=evalf(Pi,10);
  my_pi*r^2;
end proc:
> CircleArea2(5);
      78.53981635

> my_pi; r:=5: my_pi*r^2;
      3.141592654
      78.53981635
```

Working with procedure arguments : the `_passed` and `_npassed` parameters for all passed arguments

- **`_passed`** – the sequence of all arguments passed to the procedure when it was called (obsolete: **`args`**), has the type *exprseq*
- **`_npassed`** – the number of all arguments passed to the procedure when it was called (deprecated: **`nargs`**)

```
> maximum := proc () local max, i;  
    max := _passed[1];  
    for i from 2 to _npassed do  
        if _passed[i] > max then  
            max := _passed[i]  
        end if  
    end do;  
    max;  
end proc;  
> maximum(2, 5, 77, -10, 100.2);
```

100.2

Working with procedure arguments : `_rest` and `_nrest` parameters for "extra" arguments

If, when calling a procedure, the number of arguments passed is greater than the number of required ones, then you can access the remaining “extra” arguments:

- **`_rest`** – a sequence of “extra” arguments passed to the procedure when it is called, has the *exprseq* type
- **`_nrest`** – the number of “extra” arguments passed to the procedure when it was called

```
> f := proc(a, b) local xarg, sumarg, x;  
  xarg := a, b, _rest; sumarg := 0;  
  for x in xarg do sumarg := sumarg + x; end do;  
  sumarg;  
end proc;
```

```
> f(3, 2)
```

5

```
> f(1, 2, 3, 4, 5)
```

15

```
> f(1)
```

Error, invalid input: f uses a 2nd argument, b, which is missing

Recursive Procedures: remember option

remember option reduces the running time of a recursive procedure

Пример. Вычисление n -го числа Фибоначчи. Числа Фибоначчи задаются формулой

$f_n = f_{n-1} + f_{n-2}$ для $n \geq 2$, $f_0=0$, $f_1=1$.

```
> Fib1 := proc(n)
  if n < 2 then n
  else Fib1(n-1) + Fib1(n-2)
  end if end proc;
```

```
> Fib2 := proc(n)
  option remember
  if n < 2 then n
  else Fib2(n-1) + Fib2(n-2)
  end if end proc;
```

```
> Fib1(30);
```

832040

```
> Fib2(30);
```

832040

Проверим с помощью команды `fibonacci(n)` из пакета `combinat`.

```
> with(combinat) : fibonacci(30);
```

832040

Вычислим время работы процедур:

```
> time(Fib1(30)); time(Fib2(30));
```

2.324

0.

Outputting a custom error message and terminating the procedure

- **error** “ *Message %1....String...%2....* ”,par1,par2,...

In the error message line, *par 1* is substituted for %1, *par 2* is substituted for % 2 , and so on.

```
sq :=proc(x :: numeric)
  if x < 0 then error "Неверный аргумент: %1", x;
  end if;
  sqrt(x);
end proc;
```

Вызов и выполнение процедуры.

> sq(2);

$\sqrt{2}$

> sq(2.5);

1.581138830

> sq(-2);

Error, (in sq) Неверный аргумент: -2
(выдается пользовательское сообщение об ошибке)

> sq(`b`);

Error, invalid input: sq expects its 1st argument, x, to be of type numeric, but received b

(выдается системное сообщение об ошибке при проверке типа аргумента)

When the **error command** is **executed** , all remaining commands in the procedure body are ignored.

Exiting a procedure anywhere in its body and returning the result

- **return** *exp1,expr2,...*

```
> sql := proc(x :: numeric)
  if x < 0 then return abs(x) end if;
  sqrt(x);
end proc;
```

```
> sql(4)
```

2

```
> sql(a)
```

Error, invalid input: sql expects its 1st argument, x, to be of type numeric, but received a

```
> sql(-4.5)
```

4.5

When the **return** command is executed, all remaining commands in the procedure body are ignored.