

Пакеты научных вычислений

Лекция 6

Программирование в Maple

Булева алгебра в Maple. Условный оператор.
Оператор цикла. Итеративные команды. Обзор операторов программирования. Создание процедур. Создание модулей, пакетов, библиотек. Работа с файлами

Наседкина А. А.



Булева алгебра в Maple

- Логические константы и выражения
- Логические операторы и их таблицы истинности

Реализация булевой алгебры в Maple

- Двухзначная (бинарная, двоичная) логика оперирует с двумя логическими значениями: истина и ложь (1 и 0).
- Трехзначная (троичная) логика оперирует с тремя логическими значениями: «истина», «ложь» и «не определено».
- В Maple реализована трехзначная логика с логическими значениями
 - **true** (истина)
 - **false** (ложь)
 - **FAIL** (истинность не определена)
- **Логическим**, или **булевым, выражением** в Maple называется любое выражение, которое имеет логическое значение. Для записи логического выражения в Maple можно использовать:
 - Логические значения **true, false, FAIL**
 - Операторы отношения **<, >, <=, >=, =, <>**
 - **Логические операторы** **not** (НЕ), **and** (И), **or** (ИЛИ), **xor** (исключающее ИЛИ), **implies** (импликация)

Вычисление логических выражений

- Для вычисления логических выражений служит функция **evalb**.

```
> evalb(1 > 2);  
false  
[  
> evalb(not false)  
true  
]
```

- Логическое выражение является объектом **булевского типа** данных **boolean**. Этот тип данных включает в себя следующие подтипы: ``=`, `<>`, `<`, `<=`, `and`, `or`, `xor`, `implies`, `not`.`

```
> whattype(1 = 1);  
`=`,  
[  
> type(1 = 1, boolean);  
true  
]
```

- Вычисление логического выражения происходит автоматически, если оно:
 - содержит логический оператор
 - находится в условии **if** условного оператора
 - находится в условии **while** оператора цикла

Таблицы истинности логических операторов **not** и **and**

Оператор not (НЕ)	
a	not a
true	false
false	true
FAIL	FAIL

Оператор and (И)				
a and b		b		
		true	false	FAIL
a	true	true	false	FAIL
	false	false	false	false
	FAIL	FAIL	false	FAIL



Таблицы истинности логических операторов **or** и **xor**

Оператор or (ИЛИ)				
a or b		b		
		true	false	FAIL
a	true	true	true	true
	false	true	false	FAIL
	FAIL	true	FAIL	FAIL



Оператор xor (исключающее ИЛИ)				
a xor b		b		
		true	false	FAIL
a	true	false	true	FAIL
	false	true	false	FAIL
	FAIL	FAIL	FAIL	FAIL

Операторы программирования

- Условный оператор
- Оператор цикла

Условный оператор: виды

- В Maple можно реализовать базовую структуру программирования «ветвление» с помощью различных видов *условных операторов*.

Виды ветвлений:

- обход (если-то)
if...then...end if
- разветвление (если-то-иначе)
if...then...else...end if
- множественный выбор (если-то-иначе-если...)
if...then...elif...then...<...> end if
- В качестве *условия* должно быть задано логическое выражение, в качестве *выражений* – одно или несколько выражений Maple. Ветка **then** выполняется, если результат проверки условия – истина (**true**). В противном случае (**false** или **FAIL**) выполняется ветка **else** (**elif**).

Условный оператор: простые формы

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

Команды для вывода сообщений на экран:

- **print(x)** –печатает на экран значение переменной *x*
- **print("xxx")** –печатает на экран строку "xxx"
- **printf("bbb %a cc",x)** –печатает на экран сообщение, где в строку "bbb ... cc" вместо %a подставляется значение переменной *x*

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

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

5

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

2 < 3

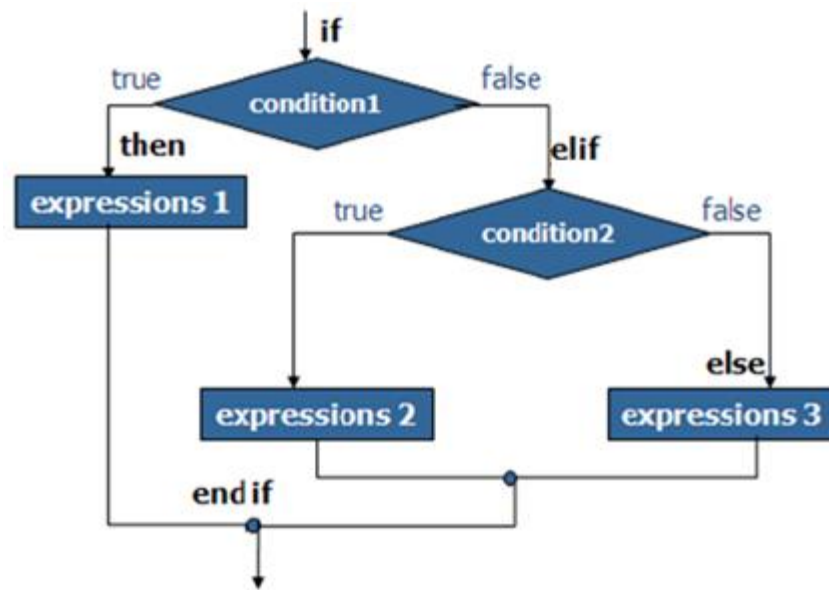
```
> if type(x, boolean)
  then print ("x – логическое выражение");
  printf ("%a - булево выражение", x) end if;
```

"x - логическое выражение"

2 < 3 – булево выражение

Условный оператор: полный синтаксис

- **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

Оператор цикла: виды

- В Maple можно реализовать базовую структуру программирования «повторение» с помощью различных видов *операторов цикла* с предусловием. Цикл с постусловием в Maple не реализован

Виды операторов цикла в Maple

- цикл-пока (while)
while...do...end do
- циклы с параметром (for/from, for/in)
for...from...do...end do
for...in...do...end do
- смешанные циклы (for/from while, for/in while)
for...from...while...do...end do
for...in...while...do...end do
- В качестве *условия while* должно быть задано логическое выражение, в качестве *выражений* – одно или несколько выражений Maple.
Выражения, расположенные между ключевыми словами **do** и **end do** составляют тело цикла.

Оператор цикла while

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

Пока результат проверки условия – истина (true), выполняются выражения Maple, составляющие тело цикла. Как только результат проверки условия становится ложным (false) или FAIL, то происходит выход из структуры цикла.

Пример. Четные числа от 8 до 2 в убывающем порядке

```
[> 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		
j := 5	}	3-я итерация
j := 4		
4	}	4-я итерация
j := 3		
j := 2	}	5-я итерация
2		
j := 1	}	6-итерация
	}	7-я итерация

Оператор цикла while: примеры

Пример. Алгоритм перевода десятичного числа в двоичное, остатки записываются в последовательность.

```
> x := 19; s := NULL;
```

```
x := 19
```

```
s :=
```

```
> while x>0 do printf("Остаток от деления %a на 2 равен %a", x, irem(x,2)); s:=s,irem(x,2); printf("Частное от деления %a на 2 равно %a", x, iquo(x,2)); x:=iquo(x,2); end do;
```

```
Остаток от деления 19 на 2 равен 1
```

```
s := 1
```

```
Частное от деления 19 на 2 равно 9
```

```
x := 9
```

```
Остаток от деления 9 на 2 равен 1
```

```
s := 1, 1
```

```
Частное от деления 9 на 2 равно 4
```

```
x := 4
```

```
Остаток от деления 4 на 2 равен 0
```

```
s := 1, 1, 0
```

```
Частное от деления 4 на 2 равно 2
```

```
x := 2
```

```
Остаток от деления 2 на 2 равен 0
```

```
s := 1, 1, 0, 0
```

```
Частное от деления 2 на 2 равно 1
```

```
x := 1
```

```
Остаток от деления 1 на 2 равен 1
```

```
s := 1, 1, 0, 0, 1
```

```
Частное от деления 1 на 2 равно 0
```

```
x := 0
```

```
> convert(19, base, 2);
```

```
[1, 1, 0, 0, 1]
```

```
> convert(19,'binary');
```

```
10011
```

Оператор цикла for/from

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

Сокращенная форма: **for** *counter* **to** *final* **do** *expressions* **end do**;

Значения по умолчанию:

initial=1, increment=1, final= ∞

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

4

5

6

7

8

Пример. Четные числа от 8 до 2 в убывающем порядке

> **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

Оператор цикла for/in

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

Параметр цикла последовательно принимает значения элементов выражения Maple (например, слагаемых в сумме, сомножителей в произведении, символов в строке, элементов списка, множества и т. д.)

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

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

x^2

$3x$

$\frac{1}{x}$

```
> s := "ЦИКЛ" :
```

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

"Ц"

"И"

"К"

"Л"

Оператор цикла for/in: примеры

Пример. Вычисление суммы полных квадратов от 1 до 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

Полный синтаксис операторов цикла for/from...while, for/in...while (смешанные циклы)

- **for** *counter* **from** *initial* **by** *increment* **to** *final* **while** *condition* **do** *expressions* **end do**;
 - **for** *variable* **in** *expression* **while** *condition* **do** *expressions* **end do**;
-
- Если в полном синтаксисе оператора цикла **for/from** задано конечное значение параметра цикла (**to final**) и присутствует проверка условия (**while condition**), то сначала выполняется сравнение текущего значения параметра цикла с конечным final, затем – проверка условия while. Если условие while станет ложным, произойдет выход из цикла.
 - Если в полном синтаксисе оператора цикла **for/in** присутствует проверка условия (**while condition**), то для каждого значения параметра сначала выполняется проверка условия, а после этого выполняется тело цикла. Параметр последовательно принимает значение элементов выражения Maple до тех пор, пока не встретится элемент, для которого условие станет ложным. При этом выход из структуры цикла может произойти раньше, чем будут исчерпаны все элементы выражения Maple (см. пример далее).

Смешанные циклы: примеры

Пример 1.

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

1

3

5

```
> x
```

7

Пример 2. Вывод на экран тех элементов списка, которые являются числами.

```
> 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
```

#двоеточие здесь не подавляет вывод на экран

2

3

4

3.5

```
> x
```

c

Пример 3. Вывод на экран последовательности первых десяти простых чисел.

```
> COUNT:=0: PRIMES:=NULL: (сначала задается пустая последовательность)
```

```
> 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;
```

Итеративные команды Maple

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

Итеративные команды: seq

- Оператор формирования последовательности
expr \$ name = initial .. final;
- Создание последовательности
seq(expression, name = initial .. final, step); #аналог цикла **for/from**
seq(expression, name in expression); #аналог цикла **for/in**

```
> $ 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]);
```

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

Итеративные команды: add, mul

- Вычисление суммы

add(expression, name = initial .. final); #аналог цикла **for/from**

add(expression, name in expression); #аналог цикла **for/in**

- Вычисление произведения

mul(expression, name = initial .. final); #аналог цикла **for/from**

mul(expression, name in expression); #аналог цикла **for/in**

```
> 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]);
```

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

Итеративные команды: `select`, `remove`

Команды извлечения и удаления (аналог цикла `for/in`)

- **`select(proc,expression)`** - извлекает те операнды из выражения *expression*, которые удовлетворяют булевой функции (процедуре) *proc*.
- **`remove(proc,expression)`** - извлекает те операнды из выражения *expression*, которые не удовлетворяют булевой функции (процедуре) *proc*.
- **`selectremove(proc,expression)`** - сначала извлекает те операнды из выражения *expression*, которые удовлетворяют булевой функции (процедуре) *proc*, а затем те, которые ей не удовлетворяют.
- Каждая команда возвращает объект того же типа, что и исходное выражение

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]
```

Итеративные команды: map

- Применение команды или процедуры к каждому члену выражения
map(proc,expression); #аналог цикла **for/in**

> 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);

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

Разложим на множители все составные числа из списка **integers**.

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

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

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

Итеративные команды Maple: `zip`

- Применение команды или процедуры покомпонентно к паре элементов двух списков или векторов **`zip`**. Команда объединяет два списка или вектора в один.
- **`zip(proc,a,b)`**
`zip(proc,a,b,fill)`
- Длина результирующего списка или вектора равна $\min(m,n)$, где m и n - длины исходных списков.
- Значение параметра *fill* используется для заполнения элементов результирующего списка в случае списков разной длины, при этом результирующий список будет иметь длину $\max(m,n)$

```
> 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]
```


Обзор операторов программирования

Все операторы программирования в 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>

Пример

- **break** – оператор прерывания цикла

```
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"
```

Пользовательские процедуры в Maple

- Функциональные операторы
- Простые процедуры
- Общие сведения о процедуре Maple
- Полный синтаксис процедуры
- Вывод кода процедуры на экран
- Локальные и глобальные переменные
- Работа с аргументами процедур
- Рекурсивные процедуры
- Вывод сообщений об ошибках, выход из процедуры

Функциональный оператор

- **Процедура** – это подпрограмма, состоящая из команд и выражений Maple. По сути, процедура также является командой Maple, но не встроенной, а созданной пользователем. Упрощенным понятием процедуры является функциональный оператор.

Общий синтаксис:

- **$f := \text{var} \rightarrow \text{result}$**
- **$f := (\text{var1}, \text{var2}, \dots) \rightarrow \text{result}$**

С помощью функционального оператора можно задавать функции одной и многих переменных, а также вектор-функции:

- $f := x \rightarrow x^2$ – функция одной переменной $R \rightarrow R$
- $f := (x, y) \rightarrow x^2 + y^2$ – функция двух переменных $R^2 \rightarrow R$
- $f := x \rightarrow (2*x, 3*x^4)$ – вектор-функция одной переменной $R \rightarrow R^2$
- $f := (x, y, z) \rightarrow (x*y, y*z)$ – вектор-функция трех переменных $R^3 \rightarrow R^2$

Функциональный оператор: примеры

Создание функционального оператора из выражения и входящих в него переменных

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

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

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

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

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

С помощью функционального оператора можно создавать небольшие процедуры:

```
> 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);
```

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

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

Общие сведения о процедуре, краткий синтаксис процедуры

- Процедура – это пользовательская команда Maple.
- Процедура может не иметь аргументов, иметь один аргумент или несколько аргументов (тогда они перечисляются через запятую).
- Все описание процедуры должно находиться в одной выполнимой группе (execution group). Переход на новую строку осуществляется с помощью **Shift+Enter**

Краткий синтаксис процедуры

- *proc_name* := **proc** (*parameterSequence*)
statementSequence;
end proc;

Слова, выделенные **синим** могут отсутствовать.

- *proc_name* – имя процедуры
- **proc ... end proc** – служебные слова, начало и конец процедуры
- *statementSequence* – последовательность выражений, реализующих *тело* процедуры. Возвращается значение последнего выражения (команды) из этой последовательности или значение выражений, указанных после команды **return**.

Пример простейшей процедуры без аргументов

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

Вызов и выполнение процедуры:

```
> ex ( );
```

$\sqrt{2}$

Пример простой процедуры с параметром

- Процедура возвращает последовательность n первых простых чисел

```
> 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

Полный синтаксис процедуры

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

Слова, выделенные **синим**, могут отсутствовать.

- *parameterSequence* – последовательность формальных параметров (*аргументов*) процедуры. Каждому формальному параметру можно предписать (*декларировать*) определенный тип данных с помощью оператора двойного двоеточия :: и следующего за ним названия типа данных *type*. При вызове процедуры в случае несоответствия какого-либо параметра его заявленному типу будет выдаваться системное сообщение об ошибке. Для параметров можно также задать значения по умолчанию.
- *returnType* – необязательный предполагаемый тип возвращаемого значения процедуры. По умолчанию, если тип возвращаемого значения не соответствует предполагаемому, ошибки не происходит.

Полный синтаксис процедуры (продолжение)

- **local** – служебное слово для описания последовательности локальных переменных *localSequence*. Локальными называются переменные, которые используются только внутри данной процедуры. Для локальных переменных можно задавать тип в виде `:: returnType`.
- **global** – служебное слово для описания последовательности глобальных переменных *globalSequence*. Глобальными называются переменные, которые не являются локальными, но также используются данной процедурой. Описание глобальных переменных используется в том случае, если этим переменным внутри процедуры будут присвоены какие-то значения. Для глобальных переменных нельзя задать тип внутри процедуры.
- **option** – служебное слово для описания последовательности опций процедуры *optionSequence*. В качестве опций используются специальные слова, например, **arrow** (стрелка), **builtin** (встроенная процедура), **operator** (оператор), **remember** (опция для эффективной работы рекурсивных процедур), **Copyright...** и некоторые другие.
- **description** – служебное слово, за которым следуют комментарии *descriptionSequence* о назначении процедуры и ее работе (одна или несколько строк). В отличие от комментариев, задаваемых символом #, данная информация выводится на экран при печати кода процедуры.
- **uses** – служебное слово для описания последовательности *usesSequence* связанных имен и модулей, которые будут использованы в теле процедуры. Может быть использовано для подключения пакетов, например: **uses StringTools;**

Примеры процедур: обязательные аргументы, количество аргументов при вызове процедуры, декларирование типов аргументов

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

Примеры процедур: декларирование типов аргументов, несколько типов данных, значения по умолчанию

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

Примеры процедур: описатели *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
```

Возврат нескольких значений из процедуры

Процедура находит все простые числа на заданном интервале и выводит их количество и сами числа в виде списка.

```
> 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]
```

Вывод кода процедуры на экран

Вывод кода пользовательской процедуры на экран

- `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
```

Вывод кода процедуры на экран (продолжение)

Вывод кода процедуры из библиотеки Maple на экран
(кроме встроенных процедур, с опцией builtin)

- **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(isqrt(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
```

Локальные и глобальные переменные: декларирование локальных переменных

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]

Локальные и глобальные переменные: декларирование локальных переменных

Если удалить строку описания локальных переменных, то будут выведены предупреждения о том, что в процедуре используются переменные `max` и `i`, которые будут декларироваться локальными:

```
>
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`

Различие между локальными и глобальными переменными

1) Декларирование локальной переменной

```
> 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
```

2) Декларирование глобальной переменной

```
> 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
```

Работа с аргументами процедуры: параметры `_passed` и `_npassed` для всех переданных аргументов

- **`_passed`** – последовательность всех аргументов, переданных процедуре при ее вызове (устаревший вариант: **`args`**), имеет тип *exprseq*
- **`_npassed`** – число всех аргументов, переданных процедуре при ее вызове (устаревший вариант: **`nargs`**)

Пример процедуры с использованием имен `_passed` и `_npassed`

Процедура находит максимум из произвольной последовательности чисел.

```
> 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

Работа с аргументами процедуры: параметры `_rest` и `_nrest` для «лишних» аргументов

Если при вызове процедуры число переданных аргументов больше числа обязательных, то можно получить доступ к оставшимся «лишним» аргументам:

- **`_rest`** – последовательность «лишних» аргументов, переданных процедуре при ее вызове, имеет тип *exprseq*
- **`_nrest`** – число «лишних» аргументов, переданных процедуре при ее вызове

Пример процедуры с использованием имени `_rest`

```
> 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
```

Рекурсивные процедуры: опция remember

Опция **remember** сокращает время работы рекурсивной процедуры

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.

Вывод пользовательского сообщения об ошибке и аварийный выход из процедуры

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

В строке сообщения об ошибке вместо %1 подставляется значение *par1*, вместо %2 подставляется значение *par2* и т. д.

```
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)

При выполнении команды **error** все оставшиеся команды в теле процедуры игнорируются.

Выход из процедуры в любом месте ее тела и возврат значений

- **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

При выполнении команды **return** все оставшиеся команды в теле процедуры игнорируются.

Обобщение процедур

- Модули
- Макроопределения `alias` и `macro`
- Создание пакетов и библиотек

Модуль

- **Модуль** – это обобщение процедуры. Процедура позволяет создать команду из последовательности Maple-команд. С помощью модуля можно создать более сложную структуру, включающую набор процедур и данных. Модуль позволяет *экспортировать переменные*

Полный синтаксис модуля

- **module()**
 export eseq;
 local lseq;
 global gseq;
 option optseq;
 description dseq;
 uses usesSequence;
 statementSequence
end module
- Описатель **export** задает имена процедур и переменных, доступных для вызова
- Остальные описатели аналогичны соответствующим описателям в процедуре

Сравнение модуля и процедуры

• Процедура

```
> makezp := proc(a, b) local plus, times; plus := (a, b) → a + b; times := (a, b) → a · b; plus, times end proc:
> makezp(5, 10)
                                     plus, times
> z := makezp( )
                                     z := plus, times
> z[1](5, 10)
                                     15
> z[2](5, 10)
                                     50
> z(5)
Error, invalid input: plus uses a 2nd argument, b, which is missing
> z(5, 10, 20)
                                     15, 50
```

• Модуль

```
> z := module( ) export plus, times; plus := (a, b) → a + b; times := (a, b) → a · b end module:
> z:-plus(5, 10)
                                     15
> z:-times(5, 10)
                                     50
```

- Обращение к экспортируемым объектам модуля – с помощью команды : –

Макроопределения: alias

Новые имена для уже существующих функций

- **alias(e1, e2, ..., eN)** – аббревиатуры для имен и функций, где **ei** имеют вид **new=old**
- Нельзя определить одну аббревиатуру alias через другую

<pre>> binomial(4, 2);</pre> 6 <pre>> alias(C = binomial) :</pre> <pre>> C(4, 2)</pre> 6 <pre>> alias($F = F(x), Fx = \frac{d}{dx} F(x)$)</pre> C, F, Fx <pre>> $\frac{\partial}{\partial x} F$</pre> Fx <pre>> $\frac{\partial}{\partial x} Fx$</pre> $\frac{\partial^2}{\partial x^2} F$	<pre>> restart,</pre> <pre>> alias(v = LinearAlgebra[VandermondeMatrix])</pre> v <pre>> v([3, 2, 1])</pre> $\begin{bmatrix} 1 & 3 & 9 \\ 1 & 2 & 4 \\ 1 & 1 & 1 \end{bmatrix}$
---	---

Макроопределения: macro

Макросы

- **macro(e1, e2, ..., eN)** – макро-аббревиатуры для имен и функций, где **ei** имеют вид **new=old**
- Можно использовать для **переопределения** имеющихся команд

```
> macro( F = combinat[ 'fibonacci' ] )
```

F

```
> F(30);
```

832040

```
> subs( x = 2, sin( x ) );
```

sin(2)

```
> macro( subs = evalf@subs ) :
```

```
> subs( x = 2, sin( x ) );
```

0.9092974268

Создание пакетов

- **Пакет** – это набор пользовательских процедур для решения задач определенного класса.

Сначала создаются и тестируются процедуры для пакета. Созданный пакет нужно сохранить в виде .m-файла

- **save** pack_name, "filename.m" – сохранение пакета pack_name в файл
- **read** "filename.m" – чтение файла с пакетом с диска
- **with**(pack_name) – подключение пакета

```
> Circle[Area] := proc(r) Pi·r2; end proc;  
> Circle[Length] := proc(r) 2·Pi·r; end proc;
```

```
> save Circle, "circle.m";
```

```
> restart,
```

```
> with(Circle) :
```

```
Error, invalid input: with expects its 1st argument,  
pname, to be of type {`module`, package}, but received  
Circle
```

```
> read "circle.m";
```

```
> with(Circle);
```

[Area, Length]

```
> Area(R)
```

πR^2

Создание библиотек

- **Библиотека** – это набор пользовательских процедур и пакетов.

Сначала нужно создать файл для библиотеки и сохранить путь к нему. Затем задать (подключить) пакеты, процедуры, модули.

- **march** – команда для работы с файлами библиотек
- **savelibname** – переменная с сохраненным путем к библиотеке
- **savelib(name1,name2,..)** – сохранение пакетов и процедур в библиотеку
- **libname** – переменная с путем к библиотеке

> restart,

Создание библиотеки в каталоге c:\temp (должен существовать)

> march('create', "c:\\temp\\my.lib");

> savelibname := "c:\\temp\\my.lib" ;

> read "circle.m"; # чтение файла с предварительно созданным пакетом

> with(Circle);# подключение пакета

[Area, Length]

> sqr := proc(x) x^2; end proc; # определение процедуры

> savelib(sqr, Circle) ;

> restart,

Подключение дополнительно созданной библиотеки

> libname := libname, "c:\\temp\\my.lib";

libname := "C:\\Program Files (x86)\\Maple 11\\lib", "c:\\temp\\my.lib"

> with(Circle) :

> Length(a); sqr(a);

$2 \pi a$

a^2

Работа с файлами.

Команды ввода и вывода

- Команды для работы с файлами
- Сохранение в файл
- Запись в файл
- Чтение из файла

Работа с файлами: сохранение переменных, открытие файла, чтение из файла

- **save name1, name2, ..., namek, fileName.ext** – сохранение переменных в файл **fileName** (расширение **.m** – внутренний формат Maple, можно указать другое)

Открытие файла

- **open("fileName.ext",mode)** – открытие файла (без буферизации), в качестве **mode** можно указать **READ** или **WRITE**
- **fopen("fileName.ext",mode,type)** – открытие файла (с буферизацией), в качестве **mode** можно указать **READ**, **WRITE** или **APPEND**, в качестве **type** – **BINARY** или **TEXT**

Чтение из файла

- **read fileName.ext** – чтение информации из файла **filename**
- **readdata(fileID, format, n)** – чтение числовой информации в заданном формате из файла
- **readline("fileName.ext")** – чтение следующей строки из заданного файла

Работа с файлами: запись в файл, закрытие файла

Запись в файл

- **writeto("fileName.ext")** – запись всех последующих команд в файл,
- **appendto ("fileName.ext")** – добавление команд без перезаписи содержимого;
- **writeto(terminal)** – возврат к записи на экран
- **writedata(fileID, data, format)** – запись числовой информации в файл в заданном формате
- **writeline("fileName.ext", str ...)** – запись строк в файл
- **fprintf(file, fmt, x1, ..., xn)** – форматированная запись в файл

Закрытие файла

- **fclose(file ...)** – закрытие файла без буферизации
- **close(file ...)** – закрытие файла с буферизацией

Сохранение переменной в файл: использование команды save

- **file.m** – файл внутреннего формата Maple

```
> a := sqrt(2) :
```

```
> save a, "d:\\my_file.m" # файл будет создан
```

Содержимое файла "myfile.m" :

M7R0

I"a*\$"###""F\$6"

```
> save a, "d:\\my_file1.txt"
```

Содержимое файла "my_file1.txt" :

a := 2^(1/2);

```
> restart,
```

```
> read "d:\\my_file.m"
```

```
> a,
```

$\sqrt{2}$

Запись в файл и на экран: использование команды `writeto`

Установка рабочей директории

- **`currentdir()`** – вывод на экран пути к каталогу с файлом (текущая рабочая директория)
- **`currentdir(dirName)`** – задание нового пути к каталогу с файлом

```
[> currentdir("d:\\"); writeto("myfile.mw")  
[>  $\frac{d}{dx} (x^3 - x^2)$   
[> writeto(terminal)  
[>  $a := x + y$   
                                 $a := x + y$   
[> writeto("myfile.mw")  
[>  $x := \text{solve}(x + 5)$ 
```

Содержимое файла `myfile.mw` будет перезаписано

Чтение числовой информации из текстового файла

- **readdata(fileID, format, n)** – чтение числовой информации
fileID – описатель файла (переменная)
n - число колонок в файле с данными, колонки должны быть разделены пробелами
format – формат данных **integer, float, string** или список этих типов

Файл data.txt содержит данные:

```
1 1 50
1 2 55
2 1 55
2 2 70
```

```
> restart,
```

```
> currentdir("d:\\");# файл находится там
```

"d:\\"

```
> readdata("data.txt", 3)#чтение как вещественных чисел по умолчанию
```

[[1., 1., 50.], [1., 2., 55.], [2., 1., 55.], [2., 2., 70.]]

```
> readdata("data.txt", integer, 3) # чтение целых чисел
```

[[1, 1, 50], [1, 2, 55], [2, 1, 55], [2, 2, 70]]

```
> readdata("data.txt", integer) # чтение целых первой колонки
```

[1, 1, 2, 2]

```
> readdata("data.txt", [integer, integer, float]) # чтение колонок в разных форматах
```

[[1, 1, 50.], [1, 2, 55.], [2, 1, 55.], [2, 2, 70.]]

Запись числовой информации в текстовый файл

- **writedata(fileID, data)**
- **writedata(fileID, data, format)**
fileID – описатель файла (переменная)
data - список, вектор или матрица
format – формат данных **integer**, **float**, **string** или список этих типов

```
> currentdir("d:\");# файл будет там
```

```
> A := array([ [1.5, 2.2, 3.4], [2.7, 3.4, 5.6], [1.8, 3.1, 6.7] ])
```

$$A := \begin{bmatrix} 1.5 & 2.2 & 3.4 \\ 2.7 & 3.4 & 5.6 \\ 1.8 & 3.1 & 6.7 \end{bmatrix}$$

```
> writedata(terminal, A, integer)
```

```
1      2      3
2      3      5
1      3      6
```

```
> writedata("data.txt", A, integer)# перезапись имеющегося файла
```

Теперь содержимое файла data.txt:

```
1 2 3
2 3 5
1 3 6
```

```
> writedata("data1.txt", A, float)
```

Файл data1.txt содержит данные :

```
1.5 2.2 3.4
2.7 3.4 5.6
1.8 3.1 6.7
```

Чтение строк из текстового файла: использование команды `readline`

```
> restart, currentdir("d:\\") :# файл будет там
```

Содержимое файла data.txt:

```
1 2 3
```

```
2 3 5
```

```
1 3 6
```

```
> f := "data.txt" :
```

```
> first := readline(f); #чтение первой строки
```

```
first := "1      2      3"
```

```
> second := readline(f); #чтение следующей строки
```

```
second := "2      3      5"
```

```
> third := readline(f); #чтение следующей строки
```

```
third := "1      3      6"
```

```
> restart, currentdir("d:\\") : f := "data.txt" : line := readline(f);  
while line ≠ 0 do line := readline(f) :end do; # чтение строк до конца
```

```
line := "1      2      3"
```

```
line := "2      3      5"
```

```
line := "1      3      6"
```

```
line := 0
```

Запись строк в текстовый файл: использование команд `writeline` и `fprintf`

```
> f1 := fopen("d:\\data2.txt", WRITE, TEXT);  
   writeline(f1, "Строка", "Еще строка") # создание нового файла и запись в него  
   fclose(f1);
```

 data2 — Блокнот

Файл Правка Формат Вид Справка

Строка

Еще строка

`fprintf(file, fmt, x1, ..., xn)` – запись выражений в файл на основе заданного формата строки

Некоторые форматы:

`d` – целое, `f` – с плавающей точкой

```
> restart, currentdir("d:\\") :  
> x := 3 : y := 12356 :  
> fd := fopen("temp_file", WRITE, TEXT); fprintf(fd, "x = %d, y = %f", x, y); fclose(fd)
```

 temp_file — Блокнот

Файл Правка Формат Вид Справка

x = 3, y = 12356.000000