Введение в программирование на языке Maple

Основные элементы

Алфавит языка Maple включает 26 букв латинского алфавита, строчных (`a-z`) и прописных (`A-Z`), десять цифр (`0-9`) и 32 специальных символа:

Символ	Описание	Символ	Описание
	Пробел	(	Левая круглая скобка
;	Точка с запятой	)	Правая круглая скобка
:	Двоеточие	[	Левая квадратная скобка
+	Плюс	]	Правая квадратная скобка
-	Минус или дефис	{	Левая фигурная скобка
*	Звездочка	}	Правая фигурная скобка
/	Косая черта	`	Обратный апостроф или обратные кавычки
^	"Крышка"	'	Апостроф или одинарные кавычки
!	Восклицательный знак	"	Двойные кавычки
=	Равенство	|	Вертикальная черта
<	Меньше	&	Амперсанд
>	Больше	_	Подчеркивание
@	Коммерческое АТ	%	Процент
$	Знак доллара	\	Обратная косая черта
.	Точка	#	Номер
,	Запятая	?	Вопросительный знак

Язык Maple чувствителен к регистру — строчные и прописные буквы алфавита различаются.

Алфавит используется для формирования *лексем* — минимальных лексических единиц, распознаваемых интерпретатором (или компилятором) языка программирования. К ним относятся ключевые слова (зарезервированные слова, используемые для формирования языковых конструкций), символы допустимых операций, имена переменных, функций и команд, строки, натуральные и целые числа, а также разделители.

Ключевые слова в языке используются для формирования его предложений: конструкций ветвления и цикла, определения процедур и модулей и т. д.:

```
and         break       by          catch       description
do          done        elif        else        end
error       export      fi          finally     for
from        global      if          in          intersect
local       minus       mod         module      next
not         od          option      options     or
proc        quit        read        return      save
stop        then        to          try         union
use         while
```

Унарные операции

Символ операции	Описание
+	Унарный плюс (знак числа)
-	Унарный минус (знак числа или операция смены знака содержимого переменной)

!	Факториал (постфиксная операция, применяется к натуральным числам)
$	Создание последовательности (применяется к диапазону)
not	Логическое отрицание
&string	Формирование нейтральной операции
.	Десятичная точка в представлении вещественного числа, отделяющая целую часть от дробной (постфиксная и префиксная операция)
%целое	Метка

Бинарные операции

Операция	**Описание**	**Операция**	**Описание**
+	Сложение	<	Меньше чем
-	Вычитание	<=	Меньше или равно
*	Умножение	>	Больше чем
/	Деление	>=	Больше или равно
^	Возведение в степень	<>	Не равно
$	Построение последовательности	->	Создание функции
@	Композиция функций	union	Объединение множеств
@@	Повторная композиция	minus	Разность множеств
$string	Нейтральная операция	intersect	Пересечение множеств
,	Разделитель выражений	::	Объявление типа
||	Конкатенация	and	Логическое И
.	Десятичная точка	or	Логическое ИЛИ
..	Диапазон	mod	Вычисление по модулю
:=	Присваивание	.	Некоммутативное умножение

В Maple существует три *операции без операндов*: %, %% и %%%, которые ссылаются, соответственно, на три предыдущих вычисленных выражения.

Символьные имена представляют собой последовательности буквенно-цифровых символов и символа подчеркивания, начинающиеся с буквы. Большое количество символьных имен защищено, и их нельзя использовать в качестве имен пользовательских переменных без снятия защиты командой `unprotect`. Эти имена имеют определенный смысл в языке и представляют имена встроенных функций, типов данных и названия команд Maple. С другой стороны, если необходимо из каких-либо соображений защитить имя, т. е. запретить пользователю его использование в левой части оператора присваивания, можно воспользоваться командой `protect()`. Снять защиту можно в любой момент командой `unprotect()`. Такие действия имеет смысл применять к определяемым пользователем процедурам или константам:

```
> Modulus_Steel:=2.1E5;
```

$$Modulus\_Steel := 210000.$$

```
> protect('Modulus_Steel');
> Modulus_Steel:=2E5;
Error, attempting to assign to `Modulus_Steel` which is protected
```

Индексные имена представляют собой символьные имена с нижним индексом. Для их задания следует после символьного имени в квадратных скобках задать последовательность выражений, причем само индексное имя также может иметь нижний индекс:

```
> j:=1;
```

$$j := 1$$

```
> A[2,3];
```

$$A_{2,3}$$

```
> A[j^2+1,k+1][float];
```

$$A_{2,k+1_{float}}$$

Отметим, что каждое индексное выражение в индексном имени вычисляется, т. е. если в нем встречается переменная, которой присвоено значение, то вместо нее подставляется это значение, как видно при задании последнего индексного имени в приведенном примере.

При проверке типа имени, которому не присвоено никакого значения (так называемого не вычисленного имени), команда `whattype()` возвращает значение `symbol` для символьного имени и `indexed` для индексного имени.

> Семантика индексного имени не предполагает, что используемое в нем символьное имя является массивом или таблицей. Так, в нашем примере `A` не является массивом, однако, если в дальнейшем имя `A` будет определено как массив, то индексное имя `A[2,3]` будет ссылаться на соответствующий элемент массива `A`.

Для создания имен можно также использовать операцию конкатенации `||` или команду `cat()`:

```
> x || 9,x || ABC;
```

$$x9, xABC$$

```
> n:=4: x || (3*n+2);
```

$$x14$$

Строка — это последовательность символов, заключенная в кавычки. Ее типом является тип `string`. Для задания символа кавычки (") в строке перед ним необходимо ввести символ обратной наклонной черты (\).Аналогично вводится и символ обратной наклонной черты, так как одиночная обратная наклонная черта трактуется как символ переноса длинной строки на другую строку ввода. Для выделения подстроки или символа из строки используется индексная запись.

```
> "a\"a"; length(%); # Количество символов в строке
```

"a\"a"

3

```
> "a\\a"; length(%); # Количество символов в строке
```

"a\\a"

3

```
> "This is \
> one string!";
```

"This is one string!"

```
> %[6..-1]; # Выделение подстроки, начиная с шестого символа и до конца.
```

"is one string!"

Натуральные числа — это любая последовательность цифр. Maple игнорирует стоящие слева нули. *Целые числа* — это натуральные числа или натуральные числа со знаком `+` или `-` перед ними. Следует отметить, что при проверке типа натурального числа команда `whattype()` возвращает целый тип `integer`, так как натуральный тип `natural` рассматривается как один из подтипов целого типа. Для уточнения подтипа целого числа следует использовать команду `type()`, вторым параметром которой может быть одно из следующих значений, описывающих подтипы целого типа: `natural` (натуральный), `negint` (отрицательное целое), `posint` (положительное целое), `nonnegint` (не отрицательное целое), `nonposint` (не положительное целое), `even` (четное), `odd` (нечетное) и `prime` (простое).

Выражения и типы

Выражение может состоять из числовых констант, имен переменных и неизвестных, булевых выражений, функций, рядов и других структур данных (объектов Maple), над которыми выполняются допустимые операции.

Для определения типа выражения можно использовать функцию `whattype()`. Более точно определить к какому типу принадлежит выражение можно командой `type`, в которой первый параметр является выражением, а второй указывает допустимый тип или подтип.

Каждое выражение подвергается синтаксическому разбору, результатом которого является построение *дерева выражения*. В первом, корневом узле отмечается тип выражения, а каждая ветвь соответствует одному из составляющих выражение членов, или операндов. Узел в каждой ветви соответствует типу операнда, так как он сам может быть сложным выражением, а его ветви определяют составляющие члены этого члена выражения. Этот процесс продолжается до тех пор, пока не дойдет до листьев дерева, представляющих имена переменных или числовые константы.

Пользователь может самостоятельно осуществить синтаксический разбор выражения, используя команды `whattype()`, `type()`, `nops()` и `op()`.

В результате вычисления выражение может оказаться равным одному из основных объектов Maple:

Объект	Тип	Подтип
Строка	`string`	
Имя	`name`	`symbol, indexed`
Целое число	`integer`	`negint, posint, nonnegint, nonposint, even, odd, prime`
Дробь, или рациональное число	`rational`	`integer, fraction`
Десятичные числа	`numeric`	`integer, fraction, float`
	`extended_numeric`	`integer, fraction, float, infinity, undefined`

Комплексные числа	`complex(integer)`
	`complex(rational)`
	`complex(float)`
	`complex(number)`	

Список	`list`	
Множество	`set`	
Вызов функции (например, `g(x)`)	`function`	

Операции отношения <, >, <=, >=, = и <>, связывающие два алгебраических выражения, формируют новый тип выражения, семантика которого зависит от контекста, в котором оно встречается.

В *алгебраическом контексте*, который встречается при выполнении присваивания или простого задания выражения без присваивания его значения какой-либо переменной, операции отношения формируют уравнения и неравенства. Тип уравнения представляется символом `` `=` ``, а тип неравенства может быть представлен одним из символов `` `<>` ``, `` `<` `` или `` `<=` ``. Любой тип отношения имеет два операнда: левую и правую часть, которые можно выделить с помощью команд `lhs()` и `rhs()`.

```
> g:=2>=x;
```

$$g := x \le 2$$

```
> whattype(g);
```

$$<=$$

```
> op(g);
```

$$x, 2$$

```
> lhs(g);
```

$$x$$

В *булевом контексте*, который возникает при использовании отношений в качестве условия в операторе `if` или после ключевого слова `while` в операторах цикла, отношение может быть равным `true` или `false`. Вычислить отношение в булевом контексте можно и командой `evalb()`, передав его ей в качестве параметра.

Логические операции `and`, `or` и `not` также позволяют формировать новое выражение, которое вычисляется в булевом контексте, причем первые две операции являются бинарными, а последняя — префиксная унарная.

Иногда необходимо провести более тонкую проверку типа выражения, чем проверка командой `type()` на соответствие выражения простому типу Maple. Например, проверка выражения x^2 на соответствие типу `` `^` `` может оказаться недостаточной, так как не дает пользователю никакой информации о типе основания и показателя степени. В таких случаях следует использовать *сложный*, или *структурный тип*, который формируется из простых типов Maple путем построения из них выражений, используемых в параметре задания типа команды `type()`:

```
> type(x^2, name^integer);
```

true

После выполнения указанной проверки будет точно известно, является ли показатель степени целым числом, а основание неизвестной величиной Maple, так как тип `name` именно ей и соответствует:

```
> type(x^(3/2),name^integer);
```

false

```
> type((x+a)^3,name^integer);
```

false

Результат выполнения проверки на структурный тип `name^integer` двух выражений примера показывает, что ни одно из них не принадлежит к выражению указанного типа. Действительно, в первом выражении показатель степени не целый, а во втором основание не является именем переменной величины Maple. В этом последнем случае можно использовать тип `anything`, который соответствует любому выражению:

```
> type((x+a)^3,anything^integer);
```

true

Если необходимо проверить выражение на соответствие *одному* из типов из некоторого их набора, то проверяемые типы следует задавать в виде множества, причем типы могут быть совершенно разные, а подобную конструкцию можно использовать в задании структурного типа:

```
> type(2.678, {integer, float});
```

true

```
> type(2.678^x, {integer, float}^{name, float});
```

true

Для проверки типов элементов множества или списка можно воспользоваться, соответственно, структурными типами `set(`тип`)` и `list(`тип`)`. В них также в качестве параметра допускается задавать множество типов для проверки неоднородных множеств и списков.

```
> type({4,x,2.3,x^2,6.7},set({prime,name,float,name^integer}));
```

false

```
> type([1..2, "a".."g"],list(range));
```

true

Так как число 4 не является простым, то результат проверки множества `false`.

При задании структурного типа можно использовать имена встроенных или пользовательских функций для проверки, является ли выражение вызовом соответствующей функции с определенным типом параметров. При этом следует имя функции задавать в одинарных кавычках, чтобы Maple не стал вычислять такие имена как вызов соответствующих функций:

```
> type(exp(2),'exp'(integer));
```

true

```
> type(int(f(x),x=1..2),int(anything,anything));
Error, testing against an invalid type
```

В последнем примере команда проверки типа `type()` печатает сообщение о не существующем типе, так как при передаче в нее второго параметра он был вычислен как интеграл

```
> int(anything,anything);
```

$$\frac{1}{2}\,anything^2$$

и полученное значение не может быть интерпретировано как допустимый тип выражения. Для корректной проверки следует имя функции вычисления интеграла `int` в задании структурного типа заключить в одинарные кавычки:

```
> type(int(f(x),x=1..2),'int'(anything,anything));
```

true

Для проверки выражения на соответствие вызова определенной функции с произвольным числом параметров, в том числе нулевым, существует специальный тип `specfunc(`тип, имя_функции`)`, в котором первый параметр определяет допустимый тип параметров функции, имя которой задается вторым параметром. Проверить, что выражение является вызовом любой функции с произвольным количеством параметров заданного типа, позволяет структурный тип `function(`тип`)`, а тип `anyfunc(`тип_1, ..., тип_n`)` проверяет выражение на вызов произвольной функции с заданным количеством параметров соответствующего типа.

```
> type(diff(f(x),x),specfunc({function(anything),name},diff));
```

true

```
> type(f(1,2),function(integer));
```

true

```
> type(f(1,2,3),anyfunc({integer,float},integer));
```

false

При задании структурного типа можно использовать логические операции `And`, `Or` и `Not` (именно с прописной буквы) для формирования булевых комбинаций типов:

```
> x:=1/2*0.897; type(x,'And(constant,float)');
```

$$x := .4485000000$$

$$true$$

```
> type(Pi,'And(constant, Not(numeric))');
```

$$true$$

Операторы

Одним из наиболее употребительных операторов Maple является оператор присваивания `:=`

```
> polynom3:=sum(c['i']*x^'i','i'=0..3);
```

$$polynom3 := c_0 + c_1 x + c_2 x^2 + c_3 x^3$$

```
> diff_polynom3:=diff(s,x);
```

$$diff\_polynom3 := c_1 + 2\, c_2 x + 3\, c_3 x^2$$

В Maple ветвление в программе реализуется оператором `if`, общая форма которого имеет следующий синтаксис (в квадратных скобках в соответствии с принятой в программировании нотацией записи синтаксиса задаются необязательные части оператора):

```
if булево_выражение then последовательность_операторов
[ elif булево_выражение then последовательность_операторов ]
[ else последовательность_операторов ]
end if
```

Семантика этого оператора проста: если истинно булево выражение после ключевого слова `if`, то выполняется последовательность операторов после ключевого слова `then` до первого встретившегося `elif`, `else` или `end if`, если его значение равно `false`, то проверяется на истинность булево выражение после ключевого слова `elif`, если оно задано, и в случае истинности выполняются операторы после ключевого `then`. Если ни одно из булевых условий не истинно, то выполняются операторы блока `else`, опять-таки, в случае его задания. Блоков `elif` может быть сколько угодно, тогда как блок `else` всегда только один.

```
> if type(polynom3,function(name)) then
       print("This is function call.");
  else
       print(polynom3);
  end if;
```

$$c_0 + c_1 x + c_2 x^2 + c_3 x^3$$

```
> x:=5:
> if x<0 then
       g:=-1;
  elif x<1 then
       g:=0;
  else
       g:=1;
  end if;
```

$$g := 1$$

Синтаксис Maple позволяет использовать вложенные конструкции `if`, т. е. последовательности операторов в блоках `then` и `else` могут содержать операторы ветвления.

Операция `` `if` `` предназначена для использования в выражениях и аналогична тернарной операции условия (?) в языке C. Она имеет следующий синтаксис:

```
`if`(условие, операнд1, операнд2)
```

Семантика ее такова: вычисляется значение параметра `условие`, который должен быть булевым выражением, и если он равен `true`, то результатом операции будет `операнд1`, в противном случае `операнд2`:

```
> a:=4: b:=3:
> c:=`if`(a>b,a,b)+sin(`if`(a>b,a,b));
```

$$c := 4 + \sin(4)$$

Для организации повторяющихся вычислений в Maple предусмотрены две формы операторов цикла: `for-from` и `for-in`. Первый оператор цикла является универсальным и включает в себя как циклы,

повторяющиеся заданное число раз, так и циклы, выполняющиеся пока некоторое булево выражение является истинным. Вторая форма цикла `for` реализует цикл по элементам списка или множества, и в других языках программирования он известен как цикл `for-each`.

Наиболее общий синтаксис оператора цикла `for-from` следующий:

```
[for имя] [from выражение] [by выражение] [to выражение]
      [while булево_выражение]
      do последовательность_операторов end do;
```

В блоке `for` задается имя переменной цикла, блоки `from` и `to` определяют, соответственно, начальное и конечное значение диапазона изменения переменной цикла, а в блоке `by` задается шаг ее изменения (он может быть и отрицательным). Выполнение цикла начинается с присваивания переменной цикла начального значения, после чего проверяется, не превосходит ли оно конечного значения, и в случае положительного ответа, выполняются операторы тела цикла, заданные в блоке `do...end do`, переменная цикла увеличивается на значение шага и алгоритм проверки начинается снова. Если значение переменной цикла превосходит конечное значение, то цикл прекращает свое выполнение. Отметим, что если при проверке начального значения переменной цикла, оно превосходит конечное значение, то цикл завершает свое выполнение, а операторы тела цикла, таким образом, не выполняются ни одного раза.

Если задан блок `while`, то одновременно с проверкой значения переменной цикла проверяется на истинность булево выражение этого блока, и цикл также завершает работу, если его значение оказывается равным `false`.

Все перечисленные блоки являются необязательными и могут задаваться в произвольном порядке за одним исключением: если присутствует блок `for`, то он должен быть задан первым. Если какой-либо блок не задан, то его параметр по умолчанию принимает значение:

Блок	**Значение параметра**
`for`	Фиктивная переменная
`from`	`1`
`by`	`1`
`to`	`infinity`
`while`	`true`

```
> # Не выполняется ни разу
> for i from 1 to -1 do evalf(sqrt(i)) end do;
> # Выполняется два раза
> for i from 1 to 2 do evalf(sqrt(i)) end do;
```

$$1.$$

$$1.414213562$$

```
> # Определяет первое простое число, большее 500000, но меньшее 500010
> for i from 5*10^5 to 500010 while not isprime(i) do end do;
> i;
```

$$500009$$

Вторая форма оператора цикла `for-in`, как уже отмечалось, организует цикл по элементам объекта, который может быть представлен последовательностью, списком, множеством, суммой, произведением или строкой. Его общий синтаксис имеет вид:

```
for имя in объект
[while булево_выражение]
do последовательность_операторов end do;
```

Переменная цикла, определяемая в блоке `for...in`, последовательно принимает значения операндов объекта, как они определяются командой `op()`. Цикл выполняется столько раз, сколько операндов задано в объекте, если только булево выражение в необязательном блоке `while` не станет ложным раньше, чем будут последовательно перебраны все операнды объекта.

```
> # Вычисление произведения слагаемых суммы
> s:=1:
> for z in sin(x)+cos(x)
  do s:=s*z end do: s;
```

$$\sin(x)\cos(x)$$

```
> # Вычисление суммы первых 2-х сомножителей произведения
> s:=0: i:=1:
> for z  in sin(x)*sin(2*x)*sin(3*x) while i<=2
```

```
  do s:=s+z; i:=i+1 end do: s;
```

$$\sin(x)+\sin(2\,x)$$

```
> # Цикл по символам строки
> for z in "ONE" do z end do;
```

"O"

"N"

"E"

```
> # Суммирование элементов множества
> S:={x,y,z}: s:=0:
> for i in S do s:=s+i end do: s;
```

$$x+y+z$$

Обычно нормальное завершение любого цикла происходит либо когда значение переменной цикла превысило заданное конечное значение (в случае положительного шага), либо она стала меньше этого значения (в случае отрицательного шага), либо булево выражение условия в блоке `while` стало ложным. Иногда, однако, необходимо прервать дальнейшее выполнение итераций цикла при выполнении некоторого условия. Для подобных случаев в языке Maple предусмотрен оператор `break`, основное предназначение которого завершить выполнение цикла. Он используется совместно с оператором условия `if`.

```
> # Суммирование первых двух элементов множества
> S:={x,y,z}: s:=0: j:=1:
> for i in S do
      s:=s+i;
      if j>2 then break; end if;
      j:=j+2;
  end do: s; j;
```

$$x+y$$

3

Часто приходится при программировании алгоритма применять конструкции цикла, но при некоторых условиях пропускать выполнение части или даже всех операторов тела цикла, переходя на следующий цикл с изменением значения переменной цикла. Такой эффект достигается применением оператора `next`, который немедленно прекращает выполнение текущей итерации цикла и переключает оператор цикла на выполнение очередной итерации.

```
> # Цикл только по нечетным значениям переменной цикла
> for i from 1 to 5 do
      if is(i,even) then next end if;
      i;
  end do;
```

1

3

5

В отличие от других языков программирования, где для организации циклов используется подобная либо оператору `for`, либо оператору `while` конструкция, Maple предлагает программисту ряд команд, в которых реализованы часто используемые в работе циклы. Эти команды позволяют писать программы намного быстрее, освобождая программиста от "рутинного" программирования некоторых видов циклов, и, что также важно, делают их более наглядными и информативными.

К таким командам можно отнести команду `map()`, которая в цикле выполняет вызов функции, определяемой первым параметром команды, и использует в качестве первого параметра функции операнды выражения, определяемого вторым параметром команды `map()`:

```
> map(abs,x^2+sin(x));
```

$$|x|^2+|\sin(x)|$$

```
> map(int,[x^2,sin(x),f(x)],x);
```

$$\left[\frac{1}{3}x^3, -\cos(x), \int \mathrm{f}(x)\,dx\right]$$

Если для выполнения команды необходимы дополнительные параметры, то все они задаются после второго параметра команды `map()`.

Аналогично команде `map()` работают и команды `select()` (выбрать), `remove()` (удалить) и `selectremove()` (выбрать и удалить), но их объектом могут быть только элементы списка:

```
> L:=[seq(i,i=1..10)];
```

$$L := [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$$

```
> select(type,L,'odd');
```

$$[1, 3, 5, 7, 9]$$

```
> remove(type,L,'odd');
```

$$[2, 4, 6, 8, 10]$$

```
> selectremove(type,L,'odd');
```

$$[1, 3, 5, 7, 9], [2, 4, 6, 8, 10]$$

Команда `zip(f, list1, list2)` создает новый список с элементами, являющимися результатом выполнения функции двух переменных `f`, в качестве параметров которой попарно выступают элементы двух списков `list1` и `list2`. Обычно создается список, длина которого равна длине наименьшего из двух заданных списков. Если при обращении к этой команде задается четвертый параметр, то длина нового списка равна длине наибольшего списка, а четвертый параметр используется в качестве недостающего параметра при вызове функции `f`.

```
> L1:=[seq(i,i="a".."b")];
```

$$L1 := [\text{"a"}, \text{"b"}]$$

```
> L2:=[seq(i,i=2..6)];
```

$$L2 := [2, 3, 4, 5, 6]$$

```
> zip((x,y)->x||y,L1,L2);
```

$$[\text{"a2"}, \text{"b3"}]$$

```
> zip((x,y)->x||y,L1,L2,phi);
```

$$[\text{"a2"}, \text{"b3"}, \phi 4, \phi 5, \phi 6]$$