Компьютерная математика. Символьный пакет Mathematica

Часть 2

Тема 6. Порядок вычисления выражений

Лаврова Ольга Анатольевна

ММФ, кафедра дифференциальных уравнений и системного анализа (ауд. 329)

Главный цикл системы Mathematica

Основным процессом в символьном пакете *Mathematica* является процесс вычисления выражений.

Вычисление выражения начинается после активизации ячейки типа Input, содержащей данное выражение.

Главный цикл системы -- это повторяющаяся в интерактивном режиме последовательность действий с момента начала вычисления выражения при активизации ячейки типа Input до момента возврата результата вычисления выражения в ячейку типа Output.

Главный цикл системы основан на совместной работе *оболочки* (Frontend), *транспортного протокола* Mathlink и *ядра* (Kernel).

1. *Оболочка* обрабатывает текстовую информацию выражения (тип String). В частности, проверяет синтаксис выражения.
2. *Транспортный протокол* передает информацию о выражении в *ядро*.
3. *Ядро* вычисляет выражение в строго определенной последовательности.
4. *Транспортный протокол* передает информацию из *ядра* в *оболочку*.
5. *Оболочка* переводит результат вычислений выражения в текст (тип String) и отображает его в ячейке типа Output.

Стандартный порядок вычисления выражения ядром

1. Не вычисляются атомарные выражения: числа (Integer, Rational, Real, Complex), строки (String), а также символы (Symbol), не имеющие собственных значений (OwnValues).
2. Символы, имеющие собственные значения (OwnValues), например, при их предварительном присваивании $x = 5$, заменяются на собственные значения, выражение переписывается и процесс вычисления применяется снова к переписанному выражению.
3. Для неатомарных выражений
   - 3.1. сначала вычисляется голова,
   - 3.2. затем последовательно вычисляются аргументы слева направо,
   - 3.3. затем вычисляется неатомарное выражение целиком.

Процесс вычисления всех подвыражений исходного выражения (голова, аргументы) осуществляется рекурсивно по указанному правилу.

4. После вычисления головы и аргументов выполняются преобразования выражения согласно атрибутам символа-головы: Orderless, Flat, Listable. Выражение переписывается и процесс вычисления применяется снова к переписанному выражению.
5. Далее применяются глобальные правила преобразований, определенные пользователем в текущей сессии, для которых левая часть правила соответствует выражению. Списки значений символа просматриваются в следующем порядке: UpValues (верхние значения символа), DownValues (нижние значения символа), SubValues (дальние значения символа).
6. В последнюю очередь применяются глобальные правила преобразований, встроенные в *ядро*.
7. После применения глобального правила преобразований (пользовательского или встроенного) выражение переписывается и процесс вычисления применяется снова к переписанному выражению.
8. Если не существует глобальных правил преобразований (пользовательских или встроенных), чья левая часть соответствует выражению, то вычисления прекращаются. Другими словами, если в процессе вычисления ни одна часть выражения не изменилась, то вычисления прекращаются.

Основной вывод: **вычисление выражения осуществляется за счет итерационного переписывания исходного выражения.**

Примеры вычисления выражения ядром

Проиллюстрируем порядок вычисления выражения с использованием функции **TracePrint,** которая выводит на экран последовательность вычисления всех подвыражений в процессе вычисления исходного выражения.

```
? TracePrint
```

TracePrint[*expr*] prints all expressions used in the evaluation of *expr*.

TracePrint[*expr*, *form*] includes only those expressions which match *form*.

TracePrint[*expr*, *s*] includes all evaluations which use transformation rules associated with the symbol *s*. ≫

Пример 1 (выражение со встроенными глобальными правилами преобразований)

Рассмотрим процесс вычисления выражения Sin[1+1.], которое содержит глобальные правила преобразований Sin и Plus.

```
Sin[1 + 1.] // TracePrint
```

```
Sin[1 + 1.]
Sin
1 + 1.
Plus
1
1.
2.
Sin[2.]
0.909297
0.909297
```

На данном примере хорошо виден процесс рекурсивного вычисления выражения.

Пример 2 (выражение с символами, для которых определены атрибуты)

Рассмотрим процесс вычисления выражения Sin@{1,1.}, в котором для символа Sin определен атрибут Listable

```
Sin@{1, 1. + 2} // TracePrint
```

```
Sin[{1, 1. + 2}]
```

```
 Sin
```

```
 {1, 1. + 2}
```

```
  List
```

```
  1
```

```
  1. + 2
```

```
   Plus
```

```
   1.
```

```
   2
```

```
  3.
```

```
 {1, 3.}
```

```
Sin[{1, 3.}]
```

```
{Sin[1], Sin[3.]}
```

```
 List
```

```
 Sin[1]
```

```
  Sin
```

```
  1
```

```
 Sin[3.]
```

```
  Sin
```

```
  3.
```

```
 0.14112
```

```
{Sin[1], 0.14112}
```

```
{Sin[1], 0.14112}
```

Пример 3 (выражение с пользовательскими глобальными правилами преобразований)

Покажем последовательность вычисления выражения с глобальным правилом преобразования, определенным пользователем

```
Unprotect[Sin];
```

```
Sin[x_Real] := "Changed Sin-function"
```

```
Sin@{1, 1.} // TracePrint
```

```
Sin[{1, 1.}]
 Sin
 {1, 1.}
{Sin[1], Sin[1.]}
 List
 Sin[1]
  Sin
  1
 Sin[1.]
  Sin
  1.
 Changed Sin-function
{Sin[1], Changed Sin-function}
{Sin[1], Changed Sin-function}
```

Удалим введенное правило

```
Sin[x_Real] =.
```

```
Protect[Sin]
```

```
{Sin}
```

Проверяем, что система корректно использует встроенные определения

```
Sin[{1, 1.}]
```

```
{Sin[1], 0.841471}
```

Пример 4 (выражение с локальными правилам преобразований)

```
{1, 1., 2} /. _Real -> Random[] // TracePrint
```

```
{1, 1., 2} /. _Real → Random[]
ReplaceAll
{1, 1., 2}
_Real → Random[]
Rule
_Real
Random[]
Random
0.323716
_Real → 0.323716
_Real → 0.323716
Rule
_Real
0.323716
{1, 1., 2} /. _Real → 0.323716
{1, 0.323716, 2}
{1, 0.323716, 2}
```

Выражения, вычисляемые нестандартно

Большинство глобальных правил преобразований (функций), встроенных в *ядро*, вычисляются стандартно.

Существуют важные функции, вычисление аргументов которых осуществляется нестандартно:

- первый аргумент не вычисляется
- вычисляется только первый аргумент, потом вычисляется функция, потом остальные аргументы, если нужно
- сначала вычисляется функция и только потом вычисляются все аргументы

Изменение стандартного порядка вычисления выражений осуществляется за счет применения к символу функции атрибутов с именем Hold*

- атрибут **HoldFirst** -- не вычисляется первый аргумент
- атрибут **HoldRest** -- вычисляется только первый аргумент
- атрибут **HoldAll** -- не вычисляются все аргументы

Функция Set вычисляется нестандартно

При вычислении функции Set первый аргумент вычисляется только после выполнения функции Set. Такая особенность функции Set реализована с помощью атрибута **HoldFirst** для символа Set

```
Attributes[Set]
```

```
{HoldFirst, Protected, SequenceHold}
```

```
?HoldFirst
```

HoldFirst is an attribute that specifies that the first argument to a function is to be maintained in an unevaluated form. ≫

Атрибут HoldFirst блокирует вычисление первого аргумента.

Пример

```
ClearAll[f]
```

```
f[x_] := x^2
```

```
f[x] = Sin[1.] // TracePrint
```

```
Sin[1.]
Sin
1.
0.841471
0.841471
```

```
f[x] + f[y]
```

$0.841471 + y^2$

```
? f
```

```
Global`f
```

```
f[x] = 0.841471
```

```
f[x_] := x^2
```

Функция If вычисляется нестандартно

```
? If
```

If[*condition*, *t*, *f*] gives *t* if *condition* evaluates to True, and *f* if it evaluates to False.

If[*condition*, *t*, *f*, *u*] gives *u* if *condition* evaluates to neither True nor False. >>

При вычислении функции If[condition, t, f] вычисляется или второй аргумент или третий аргумент.

Такая особенность функции If реализована с помощью атрибута **HoldRest** для символа If

```
Attributes[If]
```

```
{HoldRest, Protected}
```

```
? HoldRest
```

HoldRest is an attribute which specifies that all but the first argument to a function are to be maintained in an unevaluated form. >>

Атрибут HoldRest блокирует вычисление всех аргументов, кроме первого.

Пример

```
If[True, Sin[1], Cos[1]] // TracePrint
```

```
If[True, Sin[1], Cos[1]]
```

```
If
True
Sin[1]
Sin
1
```

```
Sin[1]
```

```
If[False, Sin[1], Cos[1]] // TracePrint
```

```
If[False, Sin[1], Cos[1]]
If
False
Cos[1]
Cos
1
```

```
Cos[1]
```

Функция Plot вычисляется нестандартно

При вычислении функции Plot не вычисляются все аргументы. Такая особенность функции Plot реализована с помощью атрибута **HoldAll** для символа Plot

```
Attributes[Plot]
```

```
{HoldAll, Protected, ReadProtected}
```

```
? HoldAll
```

HoldAll is an attribute that specifies that all arguments to a function are to be maintained in an unevaluated form. ≫

Атрибут HoldAll блокирует вычисление всех аргументов.

Пример

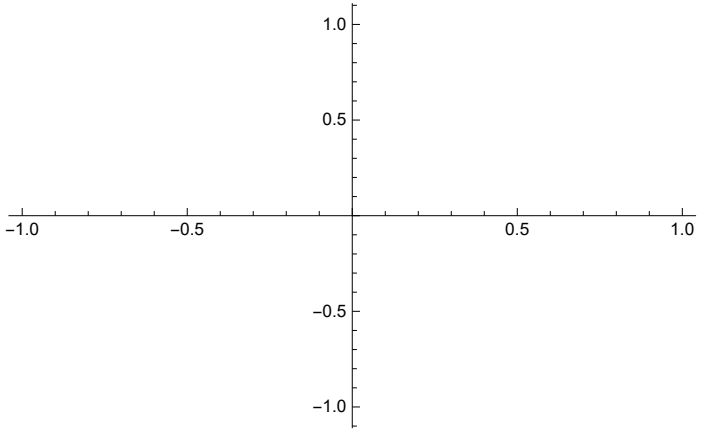
```
Plot[D[x^2, x], {x, -1, 1}]
```

General: –0.999959 is not a valid variable.

General: –0.959143 is not a valid variable.

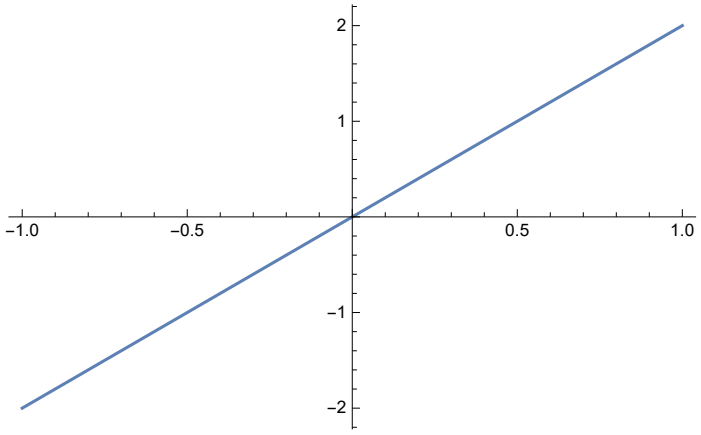
General: –0.918326 is not a valid variable.

General: Further output of General::ivar will be suppressed during this calculation.



Для изменения порядка вычислений в рассматриваемом примере применим функцию **Evaluate** к первому аргументу функции Plot

```
Plot[Evaluate@D[x^2, x], {x, -1, 1}]
```



Встроенные функции, которые влияют на порядок вычисления выражения

? Evaluate

Evaluate[*expr*] causes *expr* to be evaluated even if it appears as the argument of a function whose attributes specify that it should be held unevaluated. ≫

? Unevaluated

Unevaluated[*expr*] represents the unevaluated form of *expr* when it appears as the argument to a function. ≫

? Hold

Hold[*expr*] maintains *expr* in an unevaluated form. ≫

? HoldPattern

HoldPattern[*expr*] is equivalent to *expr* for pattern matching, but maintains *expr* in an unevaluated form. ≫

Некоторые особенности вычислений выражений

? %

% n or Out[n] is a global object that is assigned to be the value produced on the n^{th} output line.

% gives the last result generated.

%% gives the result before last. %% … % (k times) gives the k^{th} previous result. ≫