Chapter 6
Mathematica as a Programming Language

6.1 Compound Expressions

A compound expression consists of several expressions separated by the operator ;. They are calculated left to right. The value of a compound expression is the value of the last (rightmost) expression. The values of all the other expressions are thrown away; they are calculated only for side effects. The operator ; has a low priority, so that it is often necessary to put a compound expression inside brackets. The last expression may be empty. Its value (and hence the value of the compound expression) is the symbol Null which is not printed. Therefore, if you want to suppress printing of the result of some calculation (e.g., because it is lengthy), put ; after it.

```
In[1] := fac[0] = 1;
In[2] := fac[n_] := (Print["n=",n]; n * fac[n - 1])
In[3] := fac[4]
n=4
n=3
n=2
n=1
Out[3] = 24
In[4] := Clear[fac]
In[5] := Null
In[6] := FullForm[x;]
Out[6]//FullForm =
      Null
In[7] := FullForm[Hold[a;b]]
Out[7]//FullForm =
      Hold[CompoundExpression[a, b]]
```

A. Grozin, *Introduction to Mathematica® for Physicists*, Graduate Texts in Physics,
DOI 10.1007/978-3-319-00894-3_6, © Springer International Publishing Switzerland 2014

6.2 Conditional Expressions

If

In[8] := del[x_, y_] := If[x == y, 1, 0]
In[9] := del[a, a]
Out[9] = 1
In[10] := del[1, 2]
Out[10] = 0
When *Mathematica* cannot determine if the condition is true, a conditional expression is returned unevaluated. If such a possibility will appear later, an unevaluated If will be simplified.
In[11] := u = del[a, b]
Out[11] = If[a == b, 1, 0]
In[12] := a = b = x; u
Out[12] = 1
And what to do if several actions should be performed in the branches of If? Use compound expressions, of course! The priority of the operator; is higher than that of, (which separates function arguments, in particular, those of If).
In[13] := f[x_] := If[x > 0, Print["x>0"]; 1, Print["x<=0"]; 0]
In[14] := f[1]
x>0
Out[14] = 1
In[15] := Clear[a, b, u, del, f]

Which

This is a choice with many branches. Arguments of the function Which form pairs: a condition and a result. The conditions are evaluated left to right. As soon as a true one is found, the corresponding result is evaluated and returned. Often (but not always) the last condition is True; the corresponding result is returned when none of the previous conditions is satisfied. When *Mathematica* cannot decide if the conditions are true, the function Which returns unevaluated.
In[16] := sign[x_] := Which[x > 0, 1, x < 0, -1]
In[17] := sign[0.1]
Out[17] = 1
In[18] := sign[0]
In[19] := sign[a]
Out[19] = Which[a > 0, 1, a < 0, -1]
In[20] := Clear[sign]

Conditions

What can be used as conditions in If and Which? The operator == returns True if its left-hand side and right-hand side are the same expression. Let's stress: mathematically equivalent expressions written in different forms don't qualify. If the left-hand side and the right-hand one are not identical, this operator returns unevaluated. It is used for writing equations, for example, for the function Solve.

In[21] := {a == a, a == b}

Out[21] = {True, a == b}

In contrast to this, the operator === returns False if its arguments are not identical (even if they are mathematically equivalent).

In[22] := {a === a, a === b}

Out[22] = {True, False}

Not[a == b] is written as a!=b, and Not[a === b] as a=!=b.

The function NumberQ checks if its argument is a number (integer, rational, real, complex).

In[23] := {NumberQ[3.14], NumberQ[Pi]}

Out[23] = {True, False}

The function NumericQ returns True also for symbolic mathematical constants.

In[24] := {NumericQ[3.14], NumericQ[Pi]}

Out[24] = {True, True}

The function FreeQ returns True if its first argument contains no subexpressions given by the second argument. It is often used to check if an expression contains a given symbol.

In[25] := {FreeQ[Sin[a + b], a], FreeQ[Sin[b + c], a]}

Out[25] = {False, True}

The function MatchQ checks if the expression—its first argument— matches the pattern, its second argument. When designing a system of substitutions, use this function often in order to check if your ideas about the structure of expressions agree with those of *Mathematica*.

In[26] := MatchQ[x^2, a_^b_]

Out[26] = True

In[27] := MatchQ[1/x, a_^b_]

Out[27] = True

In[28] := MatchQ[x, a_^b_]

Out[28] = False

Switch

The function Switch starts from evaluating its first argument. All the remaining arguments form couples: a pattern and a result. The first argument is matched against the patterns from left to right. As soon as a match is found, the corresponding result is evaluated and returned. Often (but not always) the last pattern is $x\_$ (or just _ because we don't need the value of x).

```
In[29] := f[x_] := Switch[x, _Plus, "A sum", _Times, "A product", _,
     "Neither a sum nor a product"]
In[30] := f[a + b]
Out[30] = A sum
In[31] := f[a * b]
Out[31] = A product
In[32] := f[a^b]
Out[32] = Neither a sum nor a product
In[33] := Clear[f]
```

6.3 Loops

Do

This loop is very convenient to those pupils who were ordered by a teacher to write "I shall behave well" 100 times.

```
In[34] := Do[Print["OK"], {4}]
OK
OK
OK
OK
```

In this loop the parameter varies from 1 to an upper limit.

```
In[35] := Do[Print[x^i], {i, 4}]
```

x
x^2
x^3
x^4

And here—from a lower limit to an upper one.

```
In[36] := Do[Print[x^i], {i, 0, 4}]
1
```

x
x^2
x^3
x^4

And now with a given step.

```
In[37] := Do[Print[x^i], {i, 0, 4, 2}]
1
```

x^2
x^4

This loop takes the elements of a list.

```
In[38] := Do[Print[x^i], {i, {0, 1, 4}}]
1
```

x
x^4

While

While the list is not empty, we print and remove its first element.

In[39] := $l = \{a, b, c\}$**;**

In[40] := **While[**l**!={},Print[First[**l**]];** $l =$ **Rest[**l**]]**

a

b

c

In[41] := l

Out[41] $= \{\}$

In[42] := **Clear[**l**]**

For

This is a C style loop. First the initialization (the first argument) is executed. If the condition (the second argument) is satisfied, then the loop body (the fourth argument) is executed. Then the increment (the third argument) is performed. The condition is checked again, and so on.

In[43] := **For[**$i = 0, i < 5, i{+}{+}$**,Print[**$x\hat{}i$**]]**

1

x

x^2

x^3

x^4

In[44] := i

Out[44] $= 5$

A loop running through several parameters (or data structures) in parallel can be easily written.

In[45] := **For[**$i = 0; j = 1, i + j < 20, i{+}{+}; j* = 2$**,Print[**$x\hat{}i + y\hat{}j$**]]**

$1 + y$

$x + y^2$

$x^2 + y^4$

$x^3 + y^8$

In[46] := **Clear[**i, j**]**

6.4 Functions

The function Function returns an anonymous function. Its first argument is a formal parameter (or a list of formal parameters), and the second one is an expression containing these formal parameters. When the function is called, the actual parameters are substituted for the formal ones in this expression, and the result of its evaluation is returned as the value of the function. A note for experts: the function Function

is a λ-expression. Of course, an anonymous function can be assigned to a variable. This is similar to a function defined by $f[x\_] := \cdots$, but more efficient. The usual method of assigning a function body to a pattern is more general, because it is possible to construct a function which is defined only for arguments which satisfy some condition (such a function returns unevaluated if the conditions are not satisfied). This is not possible in the case of Function.

In[47] := f = Function[x,x^2]
Out[47] = Function $\left[x, x^2\right]$
An anonymous function can be just applied to some arguments.
In[48] := Function[{x,y},x^2+y^3][a,b]
Out[48] = $a^2 + b^3$

The function Map applies the function given by its first argument to each element of the list given by the second argument and returns the list of results.

In[49] := Map[f,{a,b,c}]
Out[49] = $\left\{a^2, b^2, c^2\right\}$
In[50] := Clear[f]
An anonymous function can be the first argument of Map. Any function with arguments (e.g., Plus) can be the second argument, not just a list. The function given by the first argument is applied to each argument of the expression—the second argument. An expression having the same Head (as the second argument) and the calculated results is constructed and returned.

In[51] := Map[Function[x,x^2],a+b+c]
Out[51] = $a^2 + b^2 + c^2$

The function Apply$[f,l]$ applies f to the list of arguments l; this simply means that the Head of l is replaced by f.

In[52] := Apply[f,{a,b,c}]
Out[52] = $f[a,b,c]$
In[53] := Apply[Times,a+b+c]
Out[53] = abc

The first argument of the function Select is a list. It returns the list of those elements which satisfy the condition given by the second argument. In order to avoid inventing names for such disposable things, anonymous functions are often used as the second argument.

In[54] := Select[{1,5,3,6},Function[x,x > 4]]
Out[54] = $\{5,6\}$

Function Generator

Here's an interesting example. The function Adder has a formal parameter n and returns a function which adds n to its argument, that is, Adder is a function generator.

In[55] := Adder = Function[n,Function[x,x+n]]
Out[55] = Function[n,Function[$x, x+n$]]
Specific functions can be obtained from it. This one, for example, adds 2 to its argument.

In[56] := Add2 = Adder[2]
Out[56] = Function[x$, x$ + 2]
In[57] := Map[Add2, {3, x}]
Out[57] = {5, 2 + x}
In[58] := Clear[Add2, Adder]

6.5 Local Variables

When writing a function which can be used as a black box by a user, it is crucial to use local variables. Assigning a value to a local variable does not change the global one with the same name (which can store some value precious for the user). To this end the function Module is used. Its first argument is a list of local variables.
In[59] := x = 1
Out[59] = 1
In[60] := Module[{x}, x = 2; x]
Out[60] = 2
In[61] := x
Out[61] = 1
In[62] := Clear[x]
Coding functions like this means inviting big troubles.
In[63] := f = Function[{a, b}, x = a; x * b]
Out[63] = Function[{a, b}, x = a; xb]
In[64] := f[c, x]
Out[64] = c^2
In[65] := x
Out[65] = c
In[66] := Clear[f, x]
This is much better.
In[67] := f = Function[{a, b}, Module[{x = a}, x * b]]
Out[67] = Function[{a, b}, Module[{x = a}, xb]]
In[68] := f[c, x]
Out[68] = cx
In[69] := x
Out[69] = x
In[70] := Clear[f]
What happens if a local variable escapes from its scope? We can see that *Mathematica* implements local variables in the most trivial way—by renaming.
In[71] := Module[{x}, x]
Out[71] = x$103
The function Block introduces another kind of local variables. As you value your life or your reason keep away from the function Block. Especially in those dark hours when the powers of evil are exalted.

Local Constants

Local variables which cannot be changed after initialization can be introduced. This is done by the function With. Such local constants can be considered temporary notations introduced to make writing a single expression easier.

In[72] := $x = 1$
Out[72] = 1
In[73] := With[{$x = a + 1$}, Print[x^2]]
Out[73] = $(1+a)^2$
In[74] := x
Out[74] = 1
In[75] := Clear[x]

6.6 Table

The function Table constructs a list of values of an expression where a parameter varies in a given way (like in the Do loop).

In[76] := Table[0, {4}]
Out[76] = $\{0,0,0,0\}$
In[77] := Table[x^i, {i, 4}]
Out[77] = $\{x, x^2, x^3, x^4\}$
In[78] := Table[x^i, {i, 0, 4}]
Out[78] = $\{1, x, x^2, x^3, x^4\}$
In[79] := Table[x^i, {i, 0, 4, 2}]
Out[79] = $\{1, x^2, x^4\}$
In[80] := Table[x^i, {i, {0, 1, 4}}]
Out[80] = $\{1, x, x^4\}$
Let's turn the list into a product.
In[81] := Table[$x + i$, {i, 0, 4}]/. List->Times
Out[81] = $x(1+x)(2+x)(3+x)(4+x)$

6.7 Parallelization

Now most computers have multi-core processors. The function Parallelize tries to calculate its argument faster by starting several *Mathematica* kernels and ordering them to calculate parts of the expression and then collecting these parts together.

In[82] := Parallelize[Table[$KernelID, {$n$, 0, 7}]]
Out[82] = $\{4,4,3,3,2,2,1,1\}$
($KernelID is the number of the kernel in which a particular list element has been evaluated). In addition to Table, it can handle Map$[f, \{\ldots\}]$ (or$f[\{\ldots\}]$ where f is a Listable function) and some other cases. Note that the slave *Mathematica*

kernels started by Parallelize don't know definitions made in the master process; only built-in functions can be used. If you need a user-defined function f to be available in slave processes, you should explicitly distribute it.

In[83] := f[n_] := Integrate[x^n * Sin[x], {x, 0, Pi}]
In[84] := DistributeDefinitions[f]
Out[84] = {f}
In[85] := Parallelize[Table[{f[n], $KernelID}, {n, 0, 7}]]
Out[85] = $\{\{2,4\},\{\pi,4\},\{-4+\pi^2,3\},\{\pi\left(-6+\pi^2\right),3\},\{48-12\pi^2+\pi^4,2\},$
$\{\pi\left(120-20\pi^2+\pi^4\right),2\},\{-1440+360\pi^2-30\pi^4+\pi^6,1\},$
$\{\pi\left(-5040+840\pi^2-42\pi^4+\pi^6\right),1\}\}$
In[86] := Clear[f]

6.8 Functions with an Index

Something like an array of functions can be constructed. Let $f[1]$ be the function adding 1 to its argument and $f[2]$—the function adding 2 to its argument; $f[n]$ is undefined for other values of n.

In[87] := f[1] = Function[x, x + 1]
Out[87] = Function[x, x + 1]
In[88] := f[2] = Function[x, x + 2]
Out[88] = Function[x, x + 2]
In[89] := {f[1][a], f[2][a], f[n][a]}
Out[89] = {1 + a, 2 + a, f[n][a]}
In[90] := Clear[f]

6.9 Hold and Evaluate

Assignment

Assignment does not evaluate its left-hand side. This is natural: the value of the right-hand side is assigned to the variable given by the left-hand side, not to its value.

In[91] := a = x
Out[91] = x
In[92] := a = y
Out[92] = y
In[93] := a
Out[93] = y
In[94] := x
Out[94] = x

The attribute HoldFirst is responsible for this.
In[95] := Attributes[Set]
Out[95] = {HoldFirst, Protected, SequenceHold}

Evaluate

It is possible to assign a value to *the value* of the left-hand side. To this end the function Evaluate is used.
In[96] := a = x
Out[96] = x
In[97] := Evaluate[a] = y
Out[97] = y
In[98] := x
Out[98] = y
In[99] := a
Out[99] = y
In[100] := Clear[x]; a
Out[100] = x
In[101] := Clear[a]

Delayed Assignment

Delayed assignment does not evaluate also its right-hand side. The attribute HoldAll is responsible for this.
In[102] := Attributes[SetDelayed]
Out[102] = {HoldAll, Protected, SequenceHold}
You can use these attributes for your functions, too.
In[103] := x = 1; y = 2; z = 3;
In[104] := Attributes[f] = {HoldAll}; f[x,y,z]
Out[104] = f[x, y, z]
In[105] := Attributes[f] = {HoldFirst}; f[x,y,z]
Out[105] = f[x, 2, 3]
In[106] := Clear[x,y,z]
In[107] := ClearAll[f]

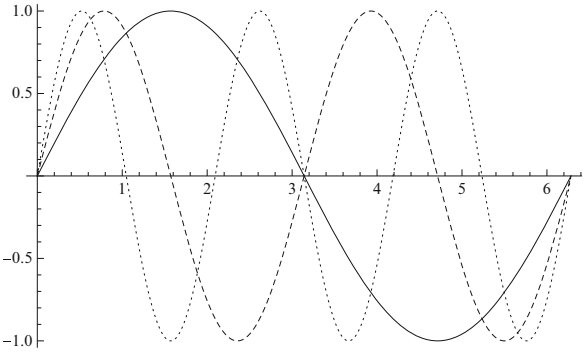
The First Argument of the Function Plot

The function Plot does not evaluate its arguments.
In[108] := Attributes[Plot]
Out[108] = {HoldAll, Protected}
The first argument $f[x]$ can be meaningful only for numerical values of x, not for symbolic x. Therefore it is better not to evaluate $f[x]$ before the function Plot will

call it for numerical values of x. But if the first argument is a command which generates the list of expressions to draw, it will not work. We want the command to be executed; to this end Evaluate is used.

In[109] := Plot[Evaluate[Table[Sin[n * x], {n, 1, 3}]], {x, 0, 2 * Pi}]

Out[109] =



Hold

The function Hold suppresses evaluation of its argument.

In[110] := a = x

Out[110] = x

In[111] := b = Hold[a]

Out[111] = Hold[a]

This suppression can be removed by the function ReleaseHold.

In[112] := ReleaseHold[b]

Out[112] = x

The function Hold is simple—it has the attribute HoldAll, i.e., it does not evaluate its arguments.

In[113] := Attributes[Hold]

Out[113] = {HoldAll, Protected}

In[114] := Clear[a, b]