Lecture 3: Introduction to Mathematica II

Sept. 11 2006

Functions and Rules

Besides MATHEMATICA® 's large set of built-in mathematical and graphics functions, the most powerful aspects of MATHEMATICA® are its ability to recognize and replace patterns and to build functions based on patterns. Learning to *program* in MATHEMATICA® is very useful and to learn to program, the basic programmatic elements must be acquired.

The following are common to almost any programming language:

Variable Storage A mechanism to define variables, and subsequently read and write them from memory.

Loops Program structures that iterate. A well-formulated loop will always be guaranteed to exit[2].

Variable Scope When a variable is defined, what other parts of the program (or other programs) will be able to read its value or change it? The scope of a variable is, roughly speaking, the extent to which it is available.

Switches These are commands with outcomes that depend on a quality of variable, but it is unknown, when the program is written, what the variable's value will be. Common names are If, Which, Switch, IfThenElse and so on.

Functions Reusable sets of commands that are stored away for future use.

All of the above are, of course, available in MATHEMATICA® .

The following are common to Symbolic and Pattern languages, like MATHEMATICA® .

Patterns This is a way of identifying common structures and make them available for subsequent computation.

Recursion This is a method to define function that obtains its value by calling itself. An example is the Fibonacci number $F_n \equiv F_{n-1}+F_{n-2}$ (The value of F is equal to the sum of the two values that preceded it.) F_n cannot be calculated until earlier values have been calculated. So, a function for Fibonacci must call itself recursively. It stops when it reaches the end condition $F_1 = F_2 = 1$.

[2]Here is a joke: "Did you hear about the computer scientist who got stuck in the shower?" "Her shampoo bottle's directions said, 'wet hair, apply shampoo, rinse, repeat'."

Lecture 03 MATHEMATICA® Example 1

Procedural Programming

Download notebooks, pdfs, or html from http://pruffle.mit.edu/3.016-2006.

Simple programs can be developed by sequences of variable assignment.

1: Here is a simple program that is just a sequence of statements that reassigns `a`. In MATHEMATICA® , a semicolon— `;`—just indicates that output should be suppressed.

4: However, it would be cumbersome and unaesthetic if the example had to be typed many many times. This is where *program loops* come in. `Do` is a simple way to loop over an expression a fixed number of times. This is equivalent to item 1, but could be easily generalized to more iterations.

6: Here an equivalent example, but extra `Print` statements are added so that *intermediate output* can be observed.

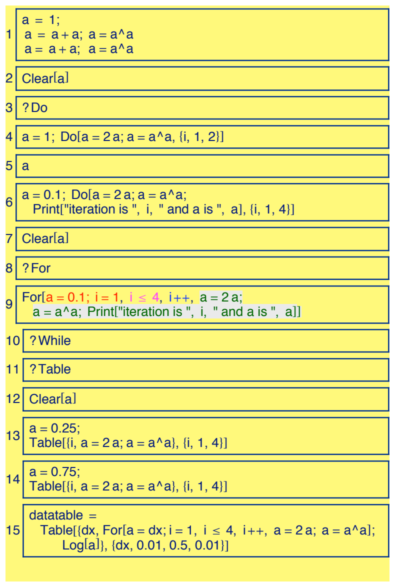
9: A `For` loop is another loop structure that enforces good programming style: Its arguments provide: an initialization, an exit condition, an iteration operator, and a function statement, and is equivalent to item 6.

10: The are many types of loop constructs; `While` is yet another.

11: `Table` is a very useful MATHEMATICA® function. While it iterates, it leaves intermediate results in a List structure.

13: Except for the *intial iteration value* of `a`, this is programmatically equivalent to items 1, 4, 6, and 9, but each iteration's result is a member of a List.

15: Here we generalize, but putting a `Table` and a `For` together. The result is a list of (lists of length 2). The first entry in each list is the initial increment value and the second entry is the result of the `For`-loop after four iterations. A special *increment structure* is utilized—it sets initial and final values as well as the increment size.



```
1   a = 1;
     a = a + a; a = a^a
     a = a + a;  a = a^a
2   Clear[a]
3   ?Do
4   a = 1; Do[a = 2 a; a = a^a, {i, 1, 2}]
5   a
6   a = 0.1; Do[a = 2 a; a = a^a;
      Print["iteration is ", i, " and a is ", a], {i, 1, 4}]
7   Clear[a]
8   ?For
9   For[a = 0.1; i = 1, i ≤ 4, i++, a = 2 a;
      a = a^a; Print["iteration is ", i, " and a is ", a]]
10  ?While
11  ?Table
12  Clear[a]
13  a = 0.25;
    Table[{i, a = 2 a; a = a^a}, {i, 1, 4}]
14  a = 0.75;
    Table[{i, a = 2 a; a = a^a}, {i, 1, 4}]
15  datatable =
      Table[{dx, For[a = dx; i = 1, i ≤ 4, i++, a = 2 a; a = a^a];
          Log[a]}, {dx, 0.01, 0.5, 0.01}]
```

Lecture 03 MATHEMATICA® Example 2

Plotting Lists of Data and Examples of Deeper MATHEMATICA® Functionality

Download notebooks, pdfs, or html from http://pruffle.mit.edu/3.016-2006.

This demonstrates how visualizing data can be combined with other functions to perform analysis.

1: The data produced from the last example can be plotted. It is apparent that there is a minimum between initial values of 0.1 and 0.3. But, it will difficult to see unless the visualization of the plot can be controlled.

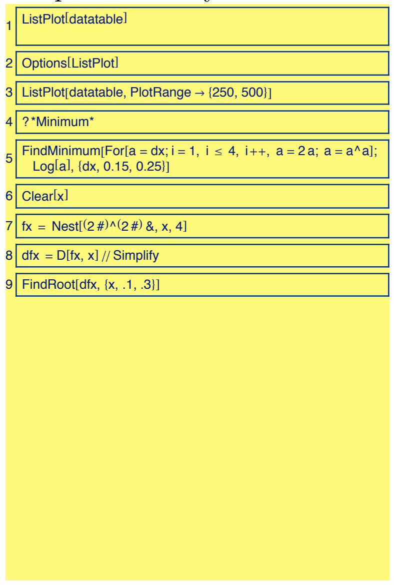
3: By specifying one of `ListPlot`'s option for the range of the y-like variable, the character of minimum can be approximately quantified.

5: `FindMinimum` is a fairly sophisticated function to obtain the minimum of an expression in a specified range, even if the function only returns a numerical result. Here `FindMinimum` is used, to find a very high precision approximation to the minimum observed in item 3.

7: This is a fairly advanced example—beginning students should not worry about understanding it yet. `Nest` is a sophisticated method of repeated application of a function (i.e., $f(f(f(x)))$ is nesting the function f three on an argument x). It is equivalent to the previous methods of producing the iterative stucture. This concept uses *Pure Functions*.

8: The minimum of the function can be analyzed the standard way, here by taking derivatives with `D`.

9: `FindRoot` is sophisticated numerical method to obtain the zero of an expression in a specified range.



```
1  ListPlot[datatable]
2  Options[ListPlot]
3  ListPlot[datatable, PlotRange → {250, 500}]
4  ?*Minimum*
5  FindMinimum[For[a = dx; i = 1, i ≤ 4, i++, a = 2a; a = a^a];
     Log[a], {dx, 0.15, 0.25}]
6  Clear[x]
7  fx = Nest[(2#)^(2#) &, x, 4]
8  dfx = D[fx, x] // Simplify
9  FindRoot[dfx, {x, .1, .3}]
```

Very complex expressions and concepts can be built-up by loops, but within MATHEMATICA® the complexity can be buried so that only the interesting parts are apparent and shown to the user.

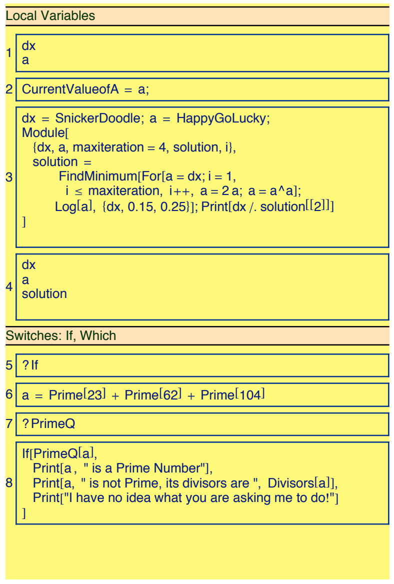
Sometimes, as complicated expressions are being built up, intermediate variables are used. Consider the value of `i` after running the program
`FindMinimum[For[a = dx; i = 1, i ≤ 4, i++, a = 2a; a = a∧a]; Log[a], {dx, 0.15, 0.25}]`,
the value of `i` (in this case 5) is has no useful meaning anymore. If you had defined a symbol such as `x = 2i` previously, then now `x` would have the value of 10, which is probably not what was intended. It is much safer to localize variables—in other words, to limit the scope of their visibility to only those parts of the program that need the variable and this is demonstrated in the next example. Sometimes this is called a "Context" for the variable in a programming language; MATHEMATICA® has contexts as well, but should probably be left as an advanced topic.

Lecture 03 MATHEMATICA® Example 3

Making Variables Local and Using Switches to Control Procedures

Download notebooks, pdfs, or html from http://pruffle.mit.edu/3.016-2006.

Describes the use of `Module` to "hide" a variable: consider the variable `a` from the first item in the above example, its intermediate values during iteration are not always important. Suppose you wish to use the symbol `a` later—that it played an intermediate role and then was not used may easily be forgotten. It is good practice to make such variables 'local' to their own functions.
A example of a logical switches is demonstrated for `If`.



Local Variables

```
1  dx
   a

2  CurrentValueofA = a;

3  dx = SnickerDoodle; a = HappyGoLucky;
   Module[
     {dx, a, maxiteration = 4, solution, i},
     solution =
          FindMinimum[For[a = dx; i = 1,
            i ≤ maxiteration, i++, a = 2 a; a = a^a];
         Log[a], {dx, 0.15, 0.25}]; Print[dx /. solution[[2]]]
   ]

4  dx
   a
   solution
```

Switches: If, Which

```
5  ?If

6  a = Prime[23] + Prime[62] + Prime[104]

7  ?PrimeQ

8  If[PrimeQ[a],
     Print[a , " is a Prime Number"],
     Print[a, " is not Prime, its divisors are ", Divisors[a]],
     Print["I have no idea what you are asking me to do!"]
   ]
```

1: The symbols `dx` and `a` are left over from the last example, even though they played only an intermediate role for a final result.

2: This could lead us to mistakenly use its value later as though it might be undefined. This is a common error.

3: The production of such errors can be reduced with a programming praction known as *localized varibles* (also known as *scoping!of variables*). The idea is to hide the variable within its own structure—the variable is said to have a limited *scope*. `Module` provides a function for doing this. Here symbols `dx` and `a` have set values before the call to `Module`, but any value that is changed *inside* of `Module` has no effect on its "global" value.

4: Even though `Module` changed symbols `dx`, `a`, and used `solution`, their should be no effect outside of `Module`.

6: Here, a simple example of the use of `If` will be applied to a symbol which is the sum of the 23rd, 62nd, and 104th prime numbers.

8: Here is a simple program. First, it first checks if `a` is prime. If the check is true, then it prints a message saying so, and then returns control to the MATHEMATICA® kernel. If the check is false, then it prints out a message and some more useful information about the fact it isn't prime. If the statement cannot be determined to be true or false, a message to that effect is printed.

Patterns are extremely important in mathematics and in MATHEMATICA® . In MATHEMATICA® , the use of the underscore, _, means "this is a placeholder for something that will be used later." It is a bit like teaching like teaching a dog to fetch—you cock an arm as if to throw _something_, and then when something gets thrown your dog runs after the "something." The first _something_ is a place holder for an object, say anything from a stick to a ball to the morning paper. The second something is the actual object that is actually tossed, that finally becomes the "something" your dog uses as the actual object in the performance of her ritual response to the action of throwing.

Usually, one needs to name to call the pattern to make it easier to refer to later. The pattern gets named by adding a head to the underscore, such as `SomeVariableName_`, and then you can refer to what ever pattern matched it with the name `SomeVariableName`.

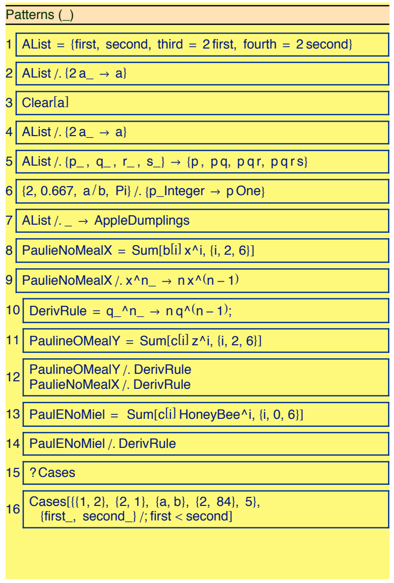
This is a bit abstract and probably difficult to understand without the aid of a few examples:

Lecture 03 MATHEMATICA® Example 4

Operating with Patterns

Download notebooks, pdfs, or html from http://pruffle.mit.edu/3.016-2006.

Learning to use expression and variable patterns is the beginning of intermediate use of MATHEMATICA® . Patterns are identified by the underscore character _, the matched pattern can be named for later use (e.g., `thematch_`) and it can be further qualified as demonstrated below.



```
Patterns (_)
1  AList = {first, second, third = 2 first, fourth = 2 second}
2  AList /. {2 a_ → a}
3  Clear[a]
4  AList /. {2 a_ → a}
5  AList /. {p_ , q_ , r_ , s_} → {p , p q, p q r, p q r s}
6  {2, 0.667, a/b, Pi} /. {p_Integer → p One}
7  AList /. _ → AppleDumplings
8  PaulieNoMealX = Sum[b[i] x^i, {i, 2, 6}]
9  PaulieNoMealX /. x^n_ → n x^(n – 1)
10 DerivRule = q_^n_ → n q^(n – 1);
11 PaulineOMealY = Sum[c[i] z^i, {i, 2, 6}]
12 PaulineOMealY /. DerivRule
   PaulieNoMealX /. DerivRule
13 PaulENoMiel = Sum[c[i] HoneyBee^i, {i, 0, 6}]
14 PaulENoMiel /. DerivRule
15 ?Cases
16 Cases[{{1, 2}, {2, 1}, {a, b}, {2, 84}, 5},
     {first_, second_} /; first < second]
```

2: Here a rule is applied to `AList` through the use of the operator `/.` (short-hand for `ReplaceAll`). The pattern here is "two multiplied by something." The symbol `a` should a placeholder for *something*, but `a` was already defined and so the behavior is probably not what was wanted. Another (probably better) usage is the *delayed ruleset* `:->`.

4: After `a` has been cleared, the symbol `a` is free to act as a placeholder; so the effect of applying the rule is that 2×*all somethings* are replaced by the pattern represented by `a`.

6: The types of things that get pattern-matched can be restricted by adding a *pattern qualifiers* to the end of the underscore.

8: For a simple (incomplete) example of the use of patterns, an example producing symbolic derivative of a polynomial will be developed. Here, a polynomial `PaulNoMealX` in `x` is defined using `Sum`.

9: A rule is applied, which replaces patterns `x` to a power with a derivative rule. Only the power is used later, so it is given a place-holder name `n`. This technique would only work on polynomials in `x`.

10: To generalize, a place-holder is defined the dependent variable.

14: This will not work for the constant and linear terms in a polynomial. This could be fixed, but the example would become too complicated and not as good as MATHEMATICA® 's built-in differention rules.

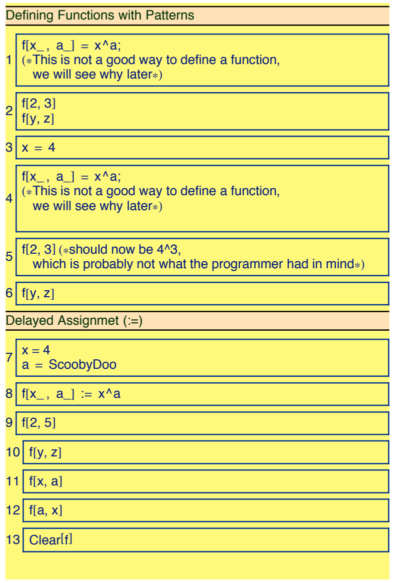
16: Patterns can also be used in conjunction with `Condition` operator `/;`. Here is an example of its use in `Cases`. The pattern is any two-member list *subject to the condition* that teh first member is less than the second.

Lecture 03 MATHEMATICA® Example 5

Creating Functions using Patterns and Delayed Assignment

Download notebooks, pdfs, or html from http://pruffle.mit.edu/3.016-2006.

The real power of patterns and replacement is obtained when defining functions. Examples of how to define functions are presented.



Defining Functions with Patterns

```
1  f[x_, a_] = x^a;
   (*This is not a good way to define a function,
     we will see why later*)
2  f[2, 3]
   f[y, z]
3  x = 4
4  f[x_, a_] = x^a;
   (*This is not a good way to define a function,
     we will see why later*)
5  f[2, 3] (*should now be 4^3,
     which is probably not what the programmer had in mind*)
6  f[y, z]
```

Delayed Assignmet (:=)

```
7  x = 4
   a = ScoobyDoo
8  f[x_, a_] := x^a
9  f[2, 5]
10 f[y, z]
11 f[x, a]
12 f[a, x]
13 Clear[f]
```

1: Here is an example of a pattern: a symbol `f` is defined such that if it is called as a function with a pattern of two named arguments `x_` and `a_`, then the result is what ever x^a evaluated to be when the function was defined. **Don't emulate this example—it is not usually the best way to define a function.**

4: This example shows why this can be a bad idea. `f` with two pattern-arguments, is assigned *when it is defined*, and therefore if either `x` or `a` was previously defined, then the definition will permanently reflect that definition.

5: Calling the function now, doesn't produce the result the user probably expected.

8: For beginning users to MATHEMATICA® , this is the best way to define functions. This involves use *delayed assignment.* In a delayed assignment, the right-hand-side is *not evaluated until the function is called* and then the patterns become *transitory until the function returns its result.* This is usually what we mean when we write $y(x) = ax^2$ mathematically—if y is given a value x, then it operates and returns a value related to that x *and not any other* x *that might have been used earlier.* **This is the prototype for function definitions.**

It is probably a good idea to define all function with delayed assignment (`:=`) instead of immediate assignment (`=`). With delayed assignment, MATHEMATICA® does not evaluate the right-hand-side *until* you ask it to perform the function. With immediate assignment, the right-hand-side is evaluated when the function is defined making it much less flexible because your name for the pattern may get "evaluated away."

Defining functions are essentially a way to eliminate repetitive typing and to "compactify" a concept. This "compactification" is essentially what we do when we define some function or operation (e.g., $\cos(\theta)$ or $\int f(x)dx$) in mathematics—the function or operation is a placeholder for something perhaps complicated to describe completely, but sufficiently understood that we can use a little picture to identify it.

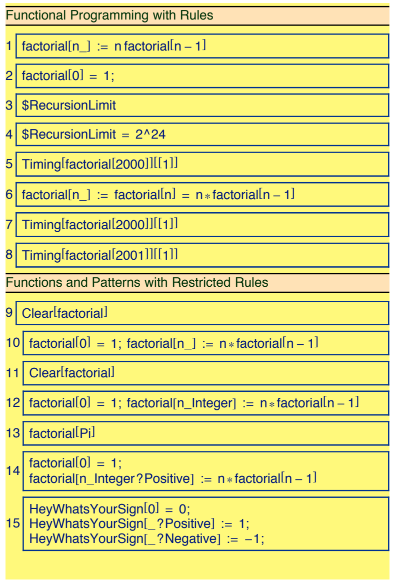
Of course, it is desirable for the function to do the something reasonable even if asked to do something that might be unreasonable. No one would buy a calculator that would try to return a very big number when division by zero occurs—or would give a real result when the arc-cosine of 1.1 is demanded. Functions should probably be defined so that they can be reused, either by you or someone else. The conditions for which the function can work should probably be encoded into the function. In MATHEMATICA® this can be done with restricted patterns.

Lecture 03 MATHEMATICA® Example 6

Functional Programming with Rules and Pattern Restrictions

Download notebooks, pdfs, or html from http://pruffle.mit.edu/3.016-2006.

Demonstration of increasingly careful factorial function defintions which will result with something sensible for non-integer or negative arguments.



```
Functional Programming with Rules
1  factorial[n_] := n factorial[n – 1]
2  factorial[0] = 1;
3  $RecursionLimit
4  $RecursionLimit = 2^24
5  Timing[factorial[2000]][[1]]
6  factorial[n_] := factorial[n] = n∗factorial[n – 1]
7  Timing[factorial[2000]][[1]]
8  Timing[factorial[2001]][[1]]
Functions and Patterns with Restricted Rules
9  Clear[factorial]
10 factorial[0] = 1; factorial[n_] := n∗factorial[n – 1]
11 Clear[factorial]
12 factorial[0] = 1; factorial[n_Integer] := n∗factorial[n – 1]
13 factorial[Pi]
14 factorial[0] = 1;
   factorial[n_Integer?Positive] := n∗factorial[n – 1]
15 HeyWhatsYourSign[0] = 0;
   HeyWhatsYourSign[_?Positive] := 1;
   HeyWhatsYourSign[_?Negative] := –1;
```

1: This is a functional definition that will produce the factorial function by recursion because $(n+1)! = (n+1)n!$. However, trying this function now will not give a satisfactory result because...

2: It is necessary to define a place for the recursion to stop. This is done by *defining* the factorial of zero to be unity.

3: So that recursive functions don't run for ever, leaving no way to get contact MATHEMATICA® 's kernel, a sensible limit is placed on the number of times a function can call itself.

5: Consider using the function to find the factorial of 2000, the currently-defined function must call itself about 2000 times to return a value. Suppose a short time later, value of 2001! is requested. The function must again call itself about 2000 times, even though all the factorials less than 2001's were calculated previously. Unless computer memory is scarce, it seems like a waste of effort to repeat the same calculations over and over.

6: Here is an example where computation speed is purchased at the cost of memory. When the function is called, it makes an assigment as well as the computation.

12: However, what if the previously-defined function were called on a value such as π? It would recursively call $(\pi - 1)!$ which would call $(\pi - 2)!$ and so on. This potential misuse can be eliminated by placing a pattern restriction on the argument of *factorial* so that it is only defined for integer arguments.

14: To prevent unbounded recursion with a call on the previous definition for negative integers, a case switch on the pattern restriction is used.

15: An example of a function that returns the `Sign` of a number if it can.