

7. Rule-Based Programming



7.1 Introduction

- What is a rule-based program?
- Objectives

7.2 Constraining Rules with Simple Patterned Arguments

- Rules with different number of arguments
- Rules with **List** arguments
- Rules with other argument patterns
- Rules with constant arguments
- The basic concept of rule-based programming
- Conflicting rules
- Exercises

7.3 Constraining Rules with x_Head

- Types of argument
- Using the **Head** of an argument
- Exercise

7.4 Constraining Rules through Predicates

- Built-in predicates
- Other predicate forms
- Boolean operations
- Using built-in predicates to constrain functions
- Exercises

7.5 Constraining Rules through Conditions

- Using predicates which impose a condition on several arguments
- Exercises

7.6 Problems

- Problem 1: Creating a gear selector program
- Problem 2: Creating a unit conversion program
- Problem 3: Writing functions which includes special cases
- Problem 4: A more challenging program

7.1 Introduction

Rule-based programs

A *rule-based program* consists of a *collection* of function definitions for the same function, each definition valid for arguments of a given pattern only.

Note carefully that the word "rule" in rule-based programming refers to a *function definition*, not specifically to rules of the sort that use \rightarrow and \Rightarrow . The reason is, as we will see more fully later, that function definitions are just another example of *rewrite rules*, just like those using \rightarrow and \Rightarrow . In line with common usage, we will from now on, often refer to a function definition as a rule.

Rule-based programs can be an easy way to program because you just write a collection of rules to be used in different circumstances, and leave it up to the software to make the decision as to which one to use in any *given* circumstance.

In the past, it was just this sort of decision making process which older program languages required you to do, which led to what was called "spaghetti" programming, where it was very difficult to trace the logic (strand of spaghetti) for debugging and quality assurance.

Objectives

- To understand the concept of rule-based programming
- To understand how to constrain a rule with patterned arguments
- To understand how to constrain an argument with a given Head
- To understand how to constrain a rule through predicates
- To understand how to constrain a rule through conditions

7.2 Constraining Rules with Simple Patterned Arguments

Rules with different number of arguments

Here is a function for computing the root sum of squares which will take any number of arguments up to three.

```

Clear[f]

f[x_] := x2;

f[x_, y_] := x2 + y2;

f[x_, y_, z_] := x2 + y2 + z2;

```

This is an example of defining a function **f** by a collection of definitional rules, each with a different number of arguments. When you enter **f[anything]**, Mathematica will look at the number of arguments in **anything** and choose which rule to apply.

Note carefully that we have put all the definitional rules in one cell and separated them with semi-colons.

We test out the function with various numbers of arguments:

```

{f[], f[a], f[a, b], f[a, b, c], f[a, b, c, d]}

{f[], a2, a2 + b2, a2 + b2 + c2, f[a, b, c, d]}

```

Mathematica has only found rules for **f[a]**, **f[a,b]**, and **f[a,b,c]**. So those are the only applications of **f** which give us a result.

Rules with List arguments

It may be that we want the user to be able to enter the arguments into a function either as a sequence of values, or as a list of values. We can arrange this by defining function rules for both cases.

```

Clear[f]

f[x_, y_] := x2 + y2;

f[{x_, y_}] := x2 + y2;

```

Here, it does not matter whether we enter the arguments as two arguments, or as one list of two arguments: we get the same result.

```

{f[a, b], f[{a, b}]}

{a2 + b2, a2 + b2}

```

We can also use lists of arguments to give *different results for the same inputs*, depending on whether the input arguments are placed in a list or not.

```

Clear[f]

f[x_, y_] := x2 + y2;

f[{x_, y_}] := x2 y2;

```

```

{f[a, b], f[{a, b}]}

{a2 + b2, a2 b2}

```

Rules with other argument patterns

Remember that in Mathematica, if something works with lists, it may well work with other expressions as well.

Let us look at the left hand side of the last function definition rule. Internally `f[{x_, y_}]` is `f[List[x_, y_]]`. So we might also expect that we can write rules using something else other than `List`. For example, we might use `Plus`, `Times` or `Power`.

```

Clear[f]

f[x_, y_] := x2 + y2;

f[x_ + y_] := x3 + y3;

f[x_ y_] := x4 + y4;

f[x_y_] := x5 + y5;

```

```

{f[a, b], f[a + b], f[a b],
 f[ab], f[2, 3], f[2 + 3], f[2 3], f[23]}

{a2 + b2, a3 + b3, a4 + b4, a5 + b5, 13, f[5], f[6], f[8]}

```

Note however, that because Mathematica *evaluates the arguments before choosing the rule*, patterns which reduce to something *for which there is no rule* may not give the output expected. Here `2+3` got evaluated to `5`, `2 3` to `6` and `23` to `8`. Since there was no rule defined for just one argument, nothing happened.

Mathematica evaluates the arguments before choosing the rule.

Rules with constant arguments

You can set up definitional rules for particular *values* rather than patterns. That is, the arguments can be constants. If you have rules with constant arguments, Mathematica will always try to apply them first before looking at your other rules..

```
Clear[f]

f[1] := 0;

f[2] := 0;

f[n_] := n2;
```

```
{f[1], f[2], f[3], f[4]}
{0, 0, 9, 16}
```

Here, although the inputs **1** and **2** also satisfied the input pattern for the last rule, Mathematica found a specific rule to apply in each of these special cases **f[1] := 0** and **f[2] := 0**, and so it applied the more specific rule rather than the more general one **f[n_] := n²**.

The basic concept of rule-based programming

The basic concept of rule-based programming is that you can have a different definitional rule for each argument pattern on the left hand side that Mathematica can distinguish as different. That is, provided there is no ambiguity about which rule should be applied in any given application of the function, you can have as many rules as you please for the same function.

For example, here are a number of different left-hand sides taking two arguments, but which have different patterns, and hence can have different rules defined for them.

```
f[x_, y_]
f[{x_, y_}]
f[{{x_, y_}}]
f[x_, {y_}]
f[{x_}, {y_}]
f[g[x], h[y]]
f[x_ + y_]
f[x_y_]
f[Exp[x_] Sin[y_]]
f[{Exp[x_] Sin[y_]}]
```

Distinguish your rules by using different patterns for the arguments.

Mathematica has *many* different ways of distinguishing the left hand sides of rules to apply in different circumstances. In what follows we will discuss some more of them.

Conflicting rules

If there is any ambiguity in which rule to apply in a given circumstance, Mathematica will not complain, but will apply one of them. Since it is unlikely to be clear which one it will choose, *you should make sure that your collection of rules is unambiguous.*

Here is an example of *bad* rule-based programming because there is no essential difference in the pattern of the arguments to distinguish the rules

```
Clear[f]

f[x_] := A;

f[y_] := B;
```

```
{f[2], f[a]}
{B, B}
```

Can you see what has happened here?

Most errors in rule-based programming occur because you think you have written a collection of unambiguous rules, but upon carefully following through the logic of pattern matching of your input to see which rule Mathematica will choose to apply, you see that you may not always be getting what you originally intended.

For example, suppose we want a function that gives **A** if there is one argument, but gives **B** if there is a sum of two arguments. We could write the (*bad*) function:

```
Clear[f]

f[x_] := A;

f[y_ + z_] := B;
```

This works okay for symbolic input

```
{f[a], f[a + b]}
{A, B}
```

But does not give the expected corresponding result for numeric input

```
{f[2], f[2 + 3]}
{A, A}
```

Make sure your collection of rules is unambiguous.

Exercises

◆ **Exercise: Understanding evaluation in rules**

Explain why the function definition in the previous example does not give **B** as output when **f[2+3]** is entered.

◆ **Exercise: Using list arguments**

Define a function using two rules $f[x_] := x^2$ and $f[\{x_-\}] := x^3$ with different right-hand sides. Apply your function to the arguments **2**, **{2}**, **{{2}}** and **{2,3}**. Explain what happened in each case.

◆ **Exercise: Using constant arguments**

Define a function that returns the exponent **b** of a simple positive integer power a^b . Define it so that it works for **b=1** also.

◆

7.3 Constraining Rules with x_Head

Types of argument

The *head* of an expression, for example, **f[x,y,z]** is **f**. We can find out the head of an expression by applying the function **Head** to it.

```
Head[f[x, y, z]]
```

```
f
```

We have seen in the previous sections how rules can be constrained by giving the *form* of their arguments different patterns.

Rules can also be constrained according to the *type* of argument. The type of an argument can be specified by its **Head**. For example, arguments might be of type **Integer**, **Real**, **Rational**, **Complex**, **Symbol**. Other expressions can have heads like **List**, **Plus**, **Times**, **Power**, ...

Using the `Head` of an argument

A function definition with an argument of the form `x_Head` will only evaluate if `x` has that head. For example, suppose we had a rule which took the `Log` of a number only if it is an integer, and in no other circumstances. We could write

```
Clear[f]
f[x_Integer] := Log[x]
```

```
{f[2], f[2.0], f[a]}
{Log[2], f[2.], f[a]}
```

Sometimes we want to ensure that the input to a function is a list without specifying its contents. For example

```
Clear[f]
f[x_List] := Reverse[x]
```

```
{f[{2}], f[{a,b,c,d}], f[2], f[a,b,c,d]}
{{2}, {d, c, b, a}, f[2], f[a, b, c, d]}
```

Exercise

◆ Exercise: Using the `Head` of an argument

Write a function that computes `Sin[x]` if `x` is a real number, `Exp[x]` if `x` is an integer, `x` if `x` is a symbol; and returns your input `x` unevaluated if it is neither a real number, an integer or a symbol.

◆

7.4 Constraining Rules through Predicates

Built-in predicates

Constraining rules through predicates is another way of ensuring you can write rules just for the types of arguments you want.

A predicate is an expression whose value is either True or False.

Mathematica has quite a few built-in predicates for testing various properties of expressions. The convention is that their names all end in the letter **Q**. We can get a list of all the predicates by entering **?*Q**.

?*Q

ArgumentCountQ	LinkConnectedQ	OrderedQ
AtomQ	LinkReadyQ	PartitionsQ
DigitQ	ListQ	PolynomialQ
EllipticNomeQ	LowerCaseQ	PrimeQ
EvenQ	MachineNumberQ	SameQ
ExactNumberQ	MatchLocalNameQ	StringMatchQ
FreeQ	MatchQ	StringQ
HypergeometricPFQ	MatrixQ	SyntaxQ
InexactNumberQ	MemberQ	TrueQ
IntegerQ	NameQ	UnsameQ
IntervalMemberQ	NumberQ	UpperCaseQ
InverseEllipticNomeQ	NumericQ	ValueQ
LegendreQ	OddQ	VectorQ
LetterQ	OptionQ	

For example, if we wanted to test if something is a list, we can use **ListQ**:

```
{ListQ[x], ListQ[{x, y]}}
{False, True}
```

Or, if we wanted to see which are the even numbers in a list, we could use **EvenQ** (which is listable):

```
EvenQ[{1, 2, 3, 4, 4.12, 5.65, a, b}]
{False, True, False, True, False, False, False, False}
```

Other predicate forms

There are other common predicate forms that are worth knowing. These include **less than**, **less than or equal to**, **equal**, **greater than or equal to**, **greater than**, **greater than zero**, **less than zero**, ...

```
{1 < 2, 1 <= 2, 1 == 2, 1 == 1,
 1 >= 2, 1 > 2, Positive[3], Negative[3]}
{True, True, False, True, False, False, True, False}
```

Boolean operations

We operate on predicates with *Boolean operations*.

The common Boolean operations are **And**, **Or** and **Not**.

Each of these has a shorthand infix form: **&&**, **|**, and **!** respectively.

```
FullForm[{x && y, x || y, ! x}]
List[And[x, y], Or[x, y], Not[x]]
```

Remember that the only arguments permitted to **And**, **Or** and **Not** are expressions which evaluate to **True** or **False**. For example

```
{(1 < 2) && (2 < 1), (1 < 2) || (2 < 1), (1 < 2) && ! (2 < 1)}
{False, True, True}
```

Using built-in predicates to constrain functions

Predicates can be used to restrict argument types in almost the same way as we used **x_Head**. Only here we use a **?** before the predicate name. For example

```
Clear[f]
f[x_?EvenQ,y_?OddQ] := x-y

{f[0,1],f[1,1]}
{-1, f[1, 1]}
```

You can also make up your own predicates

```
Clear[testQ]
testQ[x_] := If[x==2 || x==3 || x==4, True, False]

{testQ[1], testQ[2], testQ[3], testQ[4], testQ[5]}
{False, True, True, True, False}
```

You can then use this predicate **testQ** in other functions. For example

```
Clear[f]
f[x_?testQ] := x^2

{f[1],f[2],f[3],f[4],f[5]}
{f[1], 4, 9, 16, f[5]}
```

Exercises

◆ **Exercise: Exploring TrueQ**

Explore the predicate **TrueQ**.

◆ **Exercise: Comparing Equal and SameQ**

Compare the predicates **==** and **===**.

◆ **Exercise: Revising Boolean operations**

Revise your understanding of the Boolean operations **And**, **Or**, and **Not**.

Note carefully the way **And** and **Or** evaluate immediately they find a (respectively) **False** or **True** value, and do not worry about what comes next.

◆ **Exercise: Defining a your own predicate**

Define a predicate called **zeroOut** which yields **True** if a number is in the range 2 to 5 or 7 to 9, and **False** otherwise. Use this predicate to define a function which returns 0 if the input is a number in the range 2 to 5 or 7 to 9, and returns the number unevaluated otherwise. Test your predicate to make sure it works as required with a symbolic input.

◆ **Exercise: Defining a function using your own predicate**

Define a function using **zeroOut** which squares its input only if it is a number in the range 2 to 5 or 7 to 9.

◆

7.5 Constraining Rules through Conditions

Using predicates which impose a condition on several arguments

We have seen two ways on which we can impose a condition on the values that we want arguments to take, using the forms **x_Head** or **x_?Predicate**.

If we want a function definition to work only when some condition on *several* arguments is satisfied we can write a predicate involving all the several arguments and then use **Condition** (shorthand **/;**)

For example, suppose we wanted a function to compute the sum of squares of three arguments only under the condition that the sum of the arguments equals 10. We include this condition in the function definition as follows

```

Clear[f]

f[x_, y_, z_] /; x + y + z == 10 := x2 + y2 + z2

{f[1, 2, 3], f[4, 4, 2], f[2, 3, 5], f[a, b, c]}
{f[1, 2, 3], 36, 38, f[a, b, c]}

```

You can read **f[x_, y_, z_] /; x+y+z==10** as **f[x_, y_, z_]** *provided that* **x+y+z==10**.

Although there are other places you may see such a condition put, for example, after the right hand side of a function definition, putting it after the left hand side is more efficient.

◆ Example

Write a function called **lister** which only works on lists, returns the list if its length is even, and returns the reverse of the list if its length is odd. But if the input is not a list, returns the text string "The function lister works only on lists."

Here, we make sure that the function only works on lists as inputs by using a constraint in the form **x_Head**, that is in this case **x_List**. We then write a separate definitional rule for each of the two cases required, using two mutually exclusive conditions to differentiate between them.

Text strings must always be enclosed in double quotation marks. Although in this simple example we have used them to return a sort of message to the user, Mathematica has a much more sophisticated way that you can have your function advise users of their errors. We will discuss these later when we discuss packaging your functions for others to use.

```

Clear[lister]

lister[x_List] /; EvenQ[Length[x]] := x;

lister[x_List] /; OddQ[Length[x]] := Reverse[x];

lister[x_] := "The function lister works only on lists."

```

Note that we have used the last rule as a "catch-all" which will fire if neither of the first two more specific rules were found to be applicable. This is a useful trick to ensure that you always give the user some output, no matter what input they give.

We can test **lister** out on a number of arguments by mapping it over a test list of arguments.

```

lister /@ {{1}, {x, y}, {1, 2, 3}, {a, b, c, d}, {a, b, c, d, e}}
{{1}, {x, y}, {3, 2, 1}, {a, b, c, d}, {e, d, c, b, a}}

```

Testing it out on something which is not a list gives us what we expect.

```
lister[a]
```

The function `lister` works only on lists.

Exercises

◆ Exercise: Composing a rule based program

Write a function that returns **A** if the input is a list of length 7, **B** if it is a positive real number, **C** if it is a negative integer, and **D** otherwise.

◆ Exercise: Composing a rule based program

Write a function that returns **A** if the input is a symbol, **B** if it is an integer multiple of a symbol, **C** if it is a list of symbols, and **D** otherwise.

◆

7.6 Problems

Problem 1: Creating a gear selector program

Write a function called `gearSelector` that takes a gear ratio in the form of an integer or a quotient of integers (rational number) and returns the list of design parameters **A** if the ratio is greater than or equal to 1 but less than 2, **B** if the ratio is greater than or equal to 2 but less than 8, and **C** if the ratio is greater than or equal to 8. (Make up your own (arbitrary) lists **A**, **B**, and **C**).

If none of these is applicable return the text string "See the HyperGear Manual, page 2059."

[Hint: Make up your own predicate to constrain the input to be only either **Integer** or **Rational**.]

◆

Problem 2: Creating a unit conversion program

Define a function called `convert` which takes an argument in the form `convert[x_inch]` and returns the value in mm, with the symbol `mm` after it.

For example `convert[2 inch]` should yield `50.8 mm`.

(You do not need the `x` provided you have a space.)

Extend the function to convert from millimetres to inches.

Extend the function further to convert between **ft** and **metre** and between **pound** and **kg**.



Problem 3: Writing functions which includes special cases

Write a function that takes two arguments of the form \mathbf{x}^n and \mathbf{y}^m , where \mathbf{x} and \mathbf{y} are symbolic, and \mathbf{m} and \mathbf{n} are integers, and computes the expression

$$(\mathbf{x}^n + \mathbf{y}^m)^{\frac{1}{n+m}}$$

Ensure that it works for the cases involving \mathbf{n} equal to **1** and \mathbf{m} equal to **1**.



Problem 4: A more challenging program

Write a function that takes a *list* of data triples of the form $\{\mathbf{a}, \mathbf{b}, \mathbf{c}\}$ and performs the following operations on each triple:

- 1) If \mathbf{b} is more than $\mathbf{a+c}$, return $\{\mathbf{b}, \mathbf{a+c}\}$.
- 2) If \mathbf{c} is more than $\mathbf{a-b}$, return $\mathbf{Max}[\mathbf{a}, \mathbf{b}]$.
- 3) If $\mathbf{a}^2 \mathbf{Exp}[\mathbf{b}] \mathbf{c}$ is greater than **1000**, return $\mathbf{Abs}[\mathbf{a} \ \mathbf{b} \ \mathbf{c}]$.

However, because these conditions are *not mutually exclusive*, design your function to apply the definitions in the order given.

[*Hint:* Develop your main function to work only on one triplet then define your final function using **Map**. Use a condition to make sure it only works on a list of triples.]

