Mathematica basics

Subject: Functional programming, pattern matching.

Introduction

Mathematica is a functional language, based on rewrite rules and an infinite evaluation model. Informally, this means it is based on the idea of "expressions" and "functions", with "replacement rules" that transform them. Just like ordinary mathematical functions, `Mathematica` functions can be composed, applied recursively, etc. in order to transform a given expression into a desired output. It is important to keep this mentality present in order to write code that is simultaneously efficient, concise, bug-free and "natural".

In this problem we will review some basic ways in which these ideas can be realized in simple yet paradigmatic cases.

Problem statement

Everything in Mathematica is an expression (except for terminals, such as a `Symbol`). An expression consists of a head and zero or more arguments `arg1`, `arg2`, ..., and is represented by `head[arg1, arg2, ...]`. You can use `FullForm` to display an expression in this form even when the notebook frontend typically displays it in more readable notation.

```
In[1]:=FullForm[a + d^2 + f + b*c]
Out[1]=Plus[a,Times[b,c],Power[d,2],f]
In[2]:=FullForm[{1,a,2,c}]
Out[2]=List[1,a,2,c]
```

FullForm examples.

As seen above, expressions can be nested using simpler expressions as arguments to create more complicated ones. Note that this nested structure lends itself to a tree-like representation, which can be displayed using `TreeForm` as shown in Fig. 1.

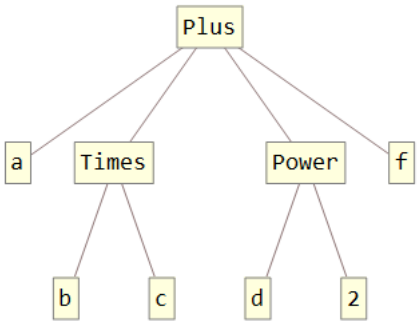

Figure 1: Example of `TreeForm[a + d^2 + f + b*c]`.

The head of an expression can be accessed using the `Head` function, and other parts of it using the `Part` function (with shorthand notation `[[···]]`).

Because everything in `Mathematica` is an expression, it is important to understand the various ways in which expressions can be transformed. There are two main ways to do this, namely applying functions to expressions or replacing parts of them with other expressions.

Function application

One way to modify an expression is to apply a function to it. If the function has been defined for this expression (either in `Mathematica` or by the user), it will immediately apply the definition to transform the expression to which it is applied.[1] Otherwise, it simply adds nodes to the expression tree. First, let us see how to apply functions to the various parts of an expression, and later we will move on to function definitions.

- You can apply a function `f` to an expression `expr` using `f[expr]`, `f@expr` or `expr//f`.

[1] It is in this sense that `Mathematica` works on an "infinite evaluation model": it eagerly tries to apply definitions as soon and as much as possible, instead of waiting until it is being required specifically by the user.

```
In[1]:= f@{a, b, c}
Out[1]= f[{a, b, c}]
```

Example: Function application.

- You can replace the head of an expression using `Apply` or `@@`. This evaluates the function on the arguments of the original expression.

```
In[1]:= f@@{a, b, c}
Out[1]= f[a, b, c]
```

Example: Apply.

Use `TreeForm` to see the difference between `f@{a,b,c}` and `f@@{a,b,c}`.

- `Apply` can take a level specification to replace the heads of an expression at some given level(s) in the expression tree. Level specifications are common in `Mathematica` so try to familiarize yourself with them. Note that the shorthand notation `@@@` replaces the heads at the first level.

```
In[1]:= f@@@{a[1], b[2], c[3]}
Out[1]= {f[1], f[2], f[3]}
```

Example: Apply on the first level.

- You can map a function over an expression using `Map` or `/@`.

```
In[1]:= f /@ {a, b, c}
Out[1]= {f[a], f[b], f[c]}
```

Example: Map.

Use `TreeForm` to see the difference between `f/@{a[1],b[2],c[3]}` and `f@@@{a[1],b[2],c[3]}`. Note that `Map` can also take a level specification.

```
In[1]:= Map[f, {a[1], b[2], c[3]}, {2}]
Out[1]= {a[f[1]], b[f[2]], c[f[3]]}
```

Example: Map on the second level only.

Pattern matching

Patterns represent classes of expressions, and can be constructed using various elements. You may test whether an expression belongs to a class, *i.e.* "matches" the corresponding pattern, using the `MatchQ` function.

- The underscore `_` represents any `Mathematica` expression, and you can give a name to the matched expression by preceeding it with a variable name, as in `x_`. You may also impose the restriction of matching only expressions with a given head by appending the head specification after the underscore.

```
In[1]:=MatchQ[f[a],f[x_]]
Out[1]=True
In[2]:=MatchQ[f[a],_g]
Out[2]=False
```

Example: Blank pattern.

- The double underscore `__` represents any sequence of one or more `Mathematica` expressions. Again, you can give a name to the matched sequence or match only expressions with a given head as above.

```
In[1]:=MatchQ[f[a,b],f[__]]
Out[1]=True
```

Example: Blank sequence pattern.

  Do note that the matched pattern is a `Sequence` and not a `List` (make sure you understand what this entails!).

- The triple underscore `___` represents any sequence of zero or more `Mathematica` expressions. Again, you can give a name to the matched sequence or match only expressions with a given head as before.

```
In[1]:=MatchQ[f[a,b,c],f[a,___,b,c]]
Out[1]=True
```

Example: Blank null sequence pattern.

  Once more, be aware that the matched pattern is a `Sequence`.

- You may impose conditions on a pattern using `/;` followed by an expression (typically containing variables named in the pattern) evaluating to a boolean value.

```
In[1]:=MatchQ[f[-2], f[x_]/;x>0]
Out[1]=False
```

Example: Pattern constraints.

A different way to impose a constraint on the expressions matched by a pattern consists in using `?` followed by a boolean testing function.

```
In[1]:=MatchQ[3, x_?EvenQ]
Out[1]=False
```

Example: Boolean pattern constraints.

Optional

Read about `Default` and the use of patterns such as `_.`, `__.` and `___.`.

Function definition

Using patterns, you can define your own functions in order to modify expressions (or perform computations, which in `Mathematica` is essentially the same thing).

- The `SetDelayed` operator `:=` defines the expression on its l.h.s. (or class thereof) to be equal to the expression on its r.h.s. Note that the assignment is "delayed" in that the expression on the r.h.s. will only be evaluated each time the l.h.s. is encountered, and not before.

```
In[1]:= f[x_]:=x^2
In[2]:= f/@{1,2,3}
Out[2]={1,4,9}
```

Example: SetDelayed.

- The `Set` operator `=` defines the expression on its l.h.s. (or class thereof) to be equal to the expression on its r.h.s., but contrary to `:=` the r.h.s. is immediately evaluated upon first making the assignment. This is

sometimes subtly different than using delayed assignment, as shown below.

```
In[1]:= f[x_]:=x+Random[]
In[2]:= g[x_]=x+Random[]
Out[2]= 0.967505 + x
In[3]:= f/@{a, b, c}
Out[3]= {0.602966 + a, 0.391871 + b, 0.242709 + c}
In[4]:= g/@{a, b, c}
Out[4]= {0.967505 + a, 0.967505 + b, 0.967505 + c}
```

Example: Set vs. SetDelayed.

Note

In `Mathematica` definitions apply from most specific to most general, so that for example we have

```
In[1]:= f[x_]:=x^2
In[2]:= f[3]:=-1
In[3]:= f/@Range@5
Out[3]= {1, 4, -1, 16, 25}
```

Example: Definition priorities.

Pure functions

Sometimes one may only need to use a very simple function once, in which case it may be desirable to avoid explicitly defining the function (assigning it and its arguments a name). It is possible to do this using "pure functions", *i.e.* functions that have no head or named arguments.

- A pure function consists of an expression where any occurrence of `#` is interpreted as the argument of the function, with the ending of the pure function being marked by `&`.

```
In[1]:=#^2&/@{1,2,3}
Out[1]={1,4,9}
```

Example: A pure function of one argument.

- Pure functions of more than one argument are defined similarly, using `#1`, `#2`, ... to denote the first, second, ... arguments.

```
In[1]:=(#2+1)^2-(#1+2)^2&[1,4]
Out[1]=16
```

Example: A pure function of two arguments.

Replacement rules

A different way to modify an expression is to use replacement rules, which replace all or part of an expression with a different one. This is particularly useful when applied in conjunction with pattern matching, but one should keep in mind that actually finding the parts of the expression matching the desired pattern is sometimes slow.

- The `->` operator (or `Rule`, displayed as →) defines a replacement rule, so that `lhs -> rhs` is interpreted as a rule for replacing the (class of) expressions `lhs` with `rhs`. Replacement rules can be grouped in lists, but do note that their ordering is important: replacements are applied in the order they appear, and this affects the results.

- The `:>` (or `RuleDelayed`) defines a replacement rule for which `rhs` will only be evaluated upon performing the replacement, and not before (similarly to the `SetDelayed` case for assignment). Again, this is subtly different than immediate evaluation (which occurs only once, upon the definition of the rule), and it is usually convenient when `lhs` is actually a class of expressions defined through a pattern with named variables to be referenced in `rhs`.

- `Replace` applies a replacement rule to all of an expression. While this may not seem very useful, it can be when used in conjunction with a level specification.

```
In[1]:=Replace[f[x,g[x],h[k[x]]],x->y,{2}]
Out[1]=f[x,g[y],h[k[x]]]
```

Example: Making a replacement at the second level.

- `ReplaceAll` (also denoted `/.`) performs a replacement for any subexpression matching the `lhs` of the rule

```
In[1]:= f[x, g[x], h[k[x]]] /. {_[x]->y, x->z}
Out[1]= f[z, y, h[y]]
```

Example: ReplaceAll with patterns and non–trivial ordering of rules.

Again, note that the use of patterns and delayed rules can subtly affect the result of a replacement:

```
In[1]:= rep={u_[x]:>u[Random[]], u_[y]->u[Random[]]}
Out[1]= {u_[x] :> u[Random[]], u_[y] -> u[0.421669]}
In[2]:= {a[x], b[x], c[y], d[y]} /. rep
Out[2]= {a[0.672257], b[0.407123], c[0.421669],
    d[0.421669]}
```

Example: Rule vs. RuleDelayed.

- `ReplaceRepeated` (also denoted `//.`) performs a `ReplaceAll` replacement repeatedly until a fixed point is reached (*i.e.* until the expression ceases to change).

```
In[1]:= f[f[f[x]]] /. f[a_]:>a+1
Out[1]= 3+x
```

Example: ReplaceRepeated.

Details

Task 1 (Factorial): Define the factorial function $n!$ using replacement rules only. What would be an equivalent defintion using a function?

Task 2 (Hello World): Use a replacement rule to transform

$$\{\{\texttt{H}, \texttt{e}, \{\texttt{l}, \{\}, \{\texttt{l}, \{\texttt{o}\}\}\}\}, \{\}, \texttt{W}, \{\{\texttt{o}, \{\texttt{r}\}\}, \texttt{l}\}, \texttt{d}\}$$

into {`H,e,l,l,o,W,o,r,l,d`}. What is the `Mathematica` built-in function that achieves the same result?

Task 3 (ConsecutivePairs): Define a `ConsecutivePairs` function that creates a list of consecutive pairs of elements in its input.

```
In[1]:= ConsecutivePairs[{1, a, c, 4, 7, y}]
Out[1]= {{1, a}, {a, c}, {c, 4}, {4, 7}, {7, y}}
```

ConsecutivePairs example.

Task 4 (Fibonacci): Define a `MyFibonacci` function that computes the n-th Fibonacci number, just like the built-in function `Fibonacci` (remember $F_n = F_{n-1} + F_{n-2}$ with $F_1 = F_2 = 1$).

```
In[1]:= MyFibonacci /@ Range@10
Out[1]= {1, 1, 2, 3, 5, 8, 13, 21, 34, 55}
```

MyFibonacci example.

How large can the argument of your function be? Use `AbsoluteTiming` to compare the performance of `MyFibonacci` to that of a replacement rule definition (*à la* Task 1 above) and the built-in `Fibonacci` function. Make sure you understand what is the reason behind the different evaluation times, and find a way to tackle this problem in your implementation (**hint:** use memoization).

Task 5 (Total): Use pattern-matching only to define a `MyTotal` function acting as the built-in `Total` (no use of loops, `Sum` or any other built-in functions is allowed!).

Task 6 (Encode/Decode): A convenient way to encode a list of lists as a single (flat) list is to represent it as

$$\{\texttt{list1}, \texttt{list2}, \dots\} \mapsto \{|\texttt{list1}|, \texttt{list1 elements}, |\texttt{list2}|, \texttt{list2 elements}, \dots\}$$

Define an `Encoding` function that encodes a list in this way, *i.e.* works as follows:

```
In[1]:= Encoding@{Range@3, Range@2, Range@5}
Out[1]= {3, 1, 2, 3, 2, 1, 2, 5, 1, 2, 3, 4, 5}
```

List of lists encoding example.

Make sure your function can work with any number of sublists, but don't worry about malformed input. Once you are confident in your `Encoding` implementation, define the corresponding `Decoding` function, so that for example

```
In[1]:= Decoding@{3,1,2,3,2,1,2,5,1,2,3,4,5}
Out[1]= {{1,2,3},{1,2},{1,2,3,4,5}}
```

List of lists decoding example.

Task 7 (DeleteDuplicates): Define a function mimicking the built-in `DeleteDuplicates` function, *i.e.* removing duplicate elements in a list respecting the order of first appearance.

```
In[1]:= RandomInteger[5, 10]
Out[1]= {3, 3, 2, 4, 1, 1, 1, 5, 0, 0}
In[2]:= MyDeleteDuplicates@%
Out[2]= {3, 2, 4, 1, 5, 0}
```

MyDeleteDuplicates example.

Take this opportunity to understand the use of `%`, `%%`, ..., and more generally of the `Out` function.

Task 8 (BubbleSort): Define a function that sorts a list.

```
In[1]:= RandomInteger[10, 10]
Out[1]= {5, 7, 9, 7, 9, 3, 4, 4, 10, 4}
In[2]:= MySort@%
Out[2]= {3, 4, 4, 4, 5, 7, 7, 9, 9, 10}
```

MySort example.

Does your function work with any list, or just lists of numbers? If the latter, check out `OrderedQ` and define a more general sorting function that works with lists of arbitrary `Mathematica` expressions.

Task 9 (Riffle): Define a function that mimicks `Riffle` (be careful with the last element!).

```
In[1]:= MyRiffle[{{1, 2}, 7, 3, 4, {5}}, x]
Out[1]= {{1, 2}, x, 7, x, 3, x, 4, x, {5}}
```

MyRiffle example.

Now convert your definition to a series of pure function applications, *i.e.* to the form `func1@func2@func3@...` where each of the steps is defined as a pure function.

Task 10 (SecondSort): Use a pure function comparator in `Sort` to define a `SecondSort` function that sorts a list according to the second component of each of its elements.

```
In[1]:= SecondSort[{f[1,4],g[2,3],f[5,5],h[6,1]}]
Out[1]= {h[6, 1], g[2, 3], f[1, 4], f[5, 5]}
```

SecondSort example.

Compare the performance of completing this task using using `Sort` and `SortBy`.

Task 11 (CentralMoment): Define a function working analogously to the built-in `CentralMoment`, taking as arguments a list of n-dimensional vectors and n moment orders.

```
In[1]:= MyCentralMoment[{{1,2}, {3,4}, {5,6}}, {2,4}]
Out[1]= 128/3
```

MyCentralMoment example.

Task 12 (Differential operator): Define a partial differentiation operator acting linearly and respecting the product rule. Add definitions for derivatives of polynomials, `Sin`, `Cos`, `Exp` and `Log` functions.

```
In[1]:= d[x][x^2 y + Sin[x y] Exp[-x] + Log@Cos@x]
Out[1]= 2 x y + E^-x y Cos[x y] - E^-x Sin[x y] - Tan[x]
```

Differential operator example.

Useful functions

You may find it useful to read the `Mathematica` help pages of the following functions:

- **List manipulation:** `Part`, `Transpose`, `Flatten`, `Take`, `Drop`, `Sort`, `SortBy`, ...
- **Profiling functions:** `AbsoluteTiming`, `AbsoluteTime`, ...
- **Other functions:** `OrderedQ`, `FreeQ`, ...