Exposé: A Tree Pattern Function in Mathematica

For: “Wissenschaftliches Arbeiten,” Software Engineering at Fachhochschule Oberösterreich, Campus Hagenberg.

The present author has implemented the following set of Wolfram Language functions in a recent Wolfram Community project (Heseltine, 2023) and would like to expand upon the underlying concepts and the language used, in an effort to further his own understanding and present a helpful guide to the language and its functional programming paradigm support.

```
In[19]:= proofID[Grid[{___, {ID, id_}, ___}, ___]] := id;

subproofs[
   Grid[{___, {Proofs, OpenerView[{Arguments, Column[subproofs_, ___]}, ___]}, ___},
    ___]] := subproofs;
subproofs[proof_] := {};

getLeanTree[proof_] := Tree[proofID[proof], getLeanTree /@ subproofs[proof],
  TreeElementLabelStyle → All → Directive[White, 16, FontFamily → "Times New Roman"],
  TreeElementStyle → All → Directive[EdgeForm[Black], RGBColor["#B6094A"]]]
```

An exposé of the concepts follows, drawing on all three functions (one function is overloaded to produce two forms of the same function), but first an introduction of both the problem space and language will be helpful. The complete code base is available as a G itHub repository.

The Problem Space

The objective is to extract a tree data structure in the form of certain integer mathematical proof IDs and the related children IDs from a grid expression in Wolfram Language. The grid is interpreted by Mathematica but comes from an external program called LEAN, an automated and interactive theorem prover. LEAN and the particular implementation details of processing the tree can be treated as a black box for the purposes of this study.

The Wolfram Language

Wolfram Research (2023) describes the language on a high level like this: “The Wolfram Language is a highly developed knowledge-based language that unifies a broad range of programming paradigms and uses its unique concept of symbolic programming to add a new level of flexibility to the very concept of programming.” Also, on the point of functional programming: “Functional programming is a highly developed and deeply integrated core feature of the Wolfram Language, made dramatically richer and more convenient through the symbolic nature of the language.”

The functional aspects of Wolfram Language to the point needed to understand the function of interest will be of particular import here, but less so the symbolic side. Mathematica is the concrete programming environment used to explore the concepts with examples.

Lists and Replacements

Going back to our object of interest, here is the first of the set of functions we are presently studying.

```
In[ ]:= proofID[Grid[{___, {ID, id_}, ___}, ___]] := id;
```

They use lists (demarcated with curly brackets, here in a nested fashion) in constructing a pattern, but lists can also be generated: Lists are central, general objects in the Wolfram Language since they can be made to represent other objects.

Internally, they are, of course, functions.

```
In[ ]:= FullForm[{1, 2, 3}]
Out[ ]//FullForm=
List[1, 2, 3]
```

Starting with list creation utilities, here are some helpful functions for handling lists.

```
In[ ]:= Range[3]
Out[ ]=
{1, 2, 3}
```

This is the same as:

```
In[ ]:= Range[1, 3, 1]
Out[ ]=
{1, 2, 3}
```

Compare this to Table, which takes a Wolfram Language iterator in the form *{i, imin, imax, step}*:

```
In[ ]:= Table[2 k, {k, 1, 10, 2}]
Out[ ]=
{2, 6, 10, 14, 18}
```

As with Range, *imin* and *step* can be omitted (set to 1).

One might like to change the display orientation, here using “%” to access the evaluation immediately prior:

```
In[ ]:= Column[%]
Out[ ]=
2
6
10
14
18
```

Matrices are similarly a matter of display, of nested lists. A construction using two iterators might look as follows.

```
In[•]:= Table[i * j, {i, 1, 3}, {j, 1, 4}] // MatrixForm
```

Out[•]//MatrixForm=

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 4 & 6 & 8 \\ 3 & 6 & 9 & 12 \end{pmatrix}$$

It is important to note the difference to this expression reversing the order of the iterators.

```
Table[i * j, {j, 1, 4}, {i, 1, 3}] // MatrixForm (* // TableForm *)
```

Out[•]//MatrixForm=

$$\begin{pmatrix} 1 & 2 & 3 \\ 2 & 4 & 6 \\ 3 & 6 & 9 \\ 4 & 8 & 12 \end{pmatrix}$$

That is, the first iterator determines the number of rows, in the 2D context. Another way to say this is that the inner iterator determines the value of the outer iterator. The mathematical expression being evaluated is, of course, commutative, however.

For measuring lists, the Length-Function gives the outermost dimension, whereas *Dimensions[]* gives all dimensions of nested lists. The number of dimensions is given by *ArrayDepth[]*.

For testing and sampling lists the following functions are available. *Position[list, element]* gives the (list of) index positions at which *element* sits, whereas *Select[list, predicate]* tests elements of the list against *predicate,* sampling those elements that satisfy, say, even parity with *EvenQ.* Given a position, elements can be extracted using *Part[]* or its short form in double-square-brackets notation, here as a matrix-example.

```
In[•]:= Table[a_{i,j}, {i, 3}, {j, 3}] // MatrixForm
```

Out[•]//MatrixForm=

$$\begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{pmatrix}$$

```
In[•]:= %[[2, 1]]
```

Out[•]=

$a_{2,1}$

Take[] is similar to *Part[]* but allows sampling consecutively placed elements in a list.

This section ends with an analysis of the "_"-character to symbolize any other Mathematica object (Wolfram Language expressions), which is useful for patterns in general, and particularly, the proofID function at the beginning of this section. The crucial difference is between two underscores, "__" or BlankSequence, standing for any sequence of one or more expressions, and three underscores, "___" or BlankNullSequence, which also allows zero such expressions.

The following example, taken from Wolfram Research (2023), makes the limitation of BlankSequence clear.

```
In[•]:= f[x__] := Length[{x}]
```

```
In[•]:= {f[x, y, z], f[]}
```

Out[•]=

```
{3, f[]}
```

Contrast this with use of the BlankNullSequence:

```
In[]:= f2[x___] := p[x, x]
In[]:= {f2[], f2[1], f2[1, a]}
Out[]= {p[], p[1, 1], p[1, a, 1, a]}
```

Blank ("_") is the most limited pattern character in that it can stand for any Wolfram Language Expression, but only exactly one such expression.

```
In[]:= f3[x_] := "evaluated"
In[]:= {f3[], f3[1], f3[1, "a"]}
Out[]= {f3[], evaluated, f3[1, a]}
```

Functional Programming

With lists there is now a background in place to consider Wolfram Language's functional aspects in more detail. For a definition, I refer to the Mathematica Cookbook (Mangano, 2010, p. 31): "All functional languages emphasize the evaluation of expressions to produce values rather than commands or statements that are executed for their side effects." Since Mathematica also allows for functions like *Do[],* the language is not purely functional, instead supporting at least also the procedural paradigm. The focus here however is functional programming by a given example, not a comparison of paradigms or more analysis of the functional one.

With this, the following core functional programming "primitives" are summed up in a helpful reference table in (Mangano, 2010, p. 26) and will lead to a discussion of the relative idioms and their application.

Out[107]=

Function	Operator	Description
Map[f, expr]	/@	Return the list that results from executing f on each element of an expr
Apply[f, expr]	@@	Return the result of replacing the head of a list with function f
Apply[f, expr, {1}]	@@@	Applies f at level 1 inside list. In other words, replace the head of all elements
Fold[f, x, {a1, a2, a3}]	N/A	If list hast length 0, return x, otherwise return f[f[f[x, a1], a2], a3]...
FoldList[f, x, {a1, a2, a3, ...}]	N/A	Return the list {x, f[x, a1], f[f[x, a1], a2], ...}
Nest[f, expr, n]	N/A	Return the list f[f[f[...f[expr]...]]] (i.e. f applied n times)
NestList[f, expr, n]	N/A	Return the list {x, f[expr], f[f[expr]], ...} where f repeats up to n times

This also covers almost all of the sort-hand notation after blanks, where ampersand ("&") and hash ("#") still need mentioning: this ties in perfectly with the topic of functional operations and goes to show that indeed, functional is at the heart of the language, since the short-hand notation offered caters to the paradigm. This example illustrates the usages, in so-called pure functions (lambda expressions), starting with the named-function counter-example.

```
In[14]:= h[x_] := f[x] + g[x]
```

Combine this with *Map[]* in the following way.

```
In[15]:= Map[h, {a, b, c}]
Out[15]=
{f[a] + g[a], f[b] + g[b], f[c] + g[c]}
```

The pure-function alternative is written as follows.

```
In[16]:= Map[f[#] + g[#] &, {a, b, c}]
Out[16]=
{f[a] + g[a], f[b] + g[b], f[c] + g[c]}
```

Here ampersand demarcates the end of the function expression, and hash marks the arguments. The non-short-hand version is:

```
In[18]:= Map[Function[x, f[x] + g[x]], {a, b, c}]
Out[18]=
{f[a] + g[a], f[b] + g[b], f[c] + g[c]}
```

I refer to the Functional Operators Tutorial (Wolfram Research, 2023) for the minor details including, for example, the modifications of the hash operator, such as *#n* or *##*. (Mangano, 2010) offer the helpful notion of idioms in this conext, where the functions *Map[]* and *Apply[]* that we have already seen present one such idiom, useful for summing sublists: here is the relevant example (p. 33), in short-hand.

```
In[23]:= Plus @@ # & /@ {{1, 2, 3}, {4, 5, 6, 7, 8}, {9, 10, 11, 12}}
Out[23]=
{6, 30, 42}
```

Plus is applied and mapped to each element, here each sublist, hence this would be the Map-Apply idiom. The same can be accomplished in other ways.

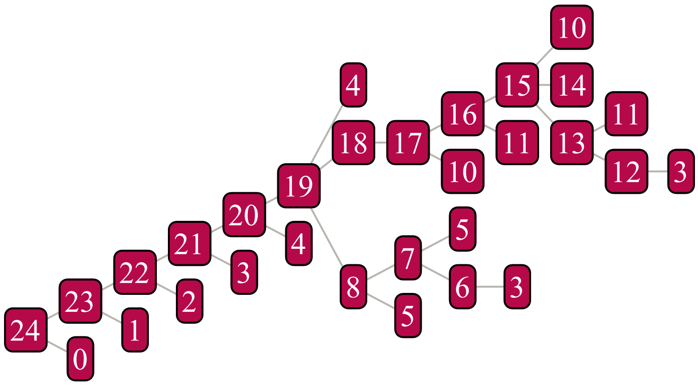
```
In[24]:= Plus @@@ {{1, 2, 3}, {4, 5, 6, 7, 8}, {9, 10, 11, 12}}
Out[24]=
{6, 30, 42}
```

(Mangano, 2010) goes on to present further use cases and appropriate idioms in the functional programming chapter of his book. The functional programming elements discussed so far allow us to analyze this part of the code, the third of the three functions, we are investigating:

```
In[25]:= getLeanTree[proof_] := Tree[proofID[proof], getLeanTree /@ subproofs[proof],
   TreeElementLabelStyle → All → Directive[White, 16, FontFamily → "Times New Roman"],
   TreeElementStyle → All → Directive[EdgeForm[Black], RGBColor["#B6094A"]]]
```

There are a lot of options here, concerned with output style. (Something like the following gets generated as the final return value: the presentation is controlled by the options.)

Out[∘]=



There is also a recursive element here, which we will ignore just for the moment, in fact we want to focus in on just this part:

In[106]:=

```
getLeanTree /@ subproofs[proof]
```

Out[106]=

```
{}
```

The "/@" - Mapping of the (recursively called) function getLeanTree[] is the Map - Apply idiom in action, where the function is applied to the output of another function subproofs[], which is a list, with usually two or three elements, behind the scenes .

In[∘]:=

```
subproofs[leanProof]
```

Out[∘]=

{

Goal	p : ℕ
ID	0
Rule	Assumption

,

Goal	∀ {m n : ℕ}, nat.prime p → p ∣ m ^ n → p ∣ m
ID	23
Rule	∀I
Proofs	▼ Arguments

Goal	m : ℕ
ID	1
Rule	Assumption

Goal	∀ {n : ℕ}, nat.prime p → p ∣ m ^ n → p ∣ m
ID	22
Rule	∀I
Proofs	▶ Arguments

}

Rule-based Programming

The grid structures presented here immediately bring us to pattern-oriented programming in Wolfram Language and we move on to the second function of our three to investigate this aspect.

In[108]:=

```
subproofs[
   Grid[{___, {Proofs, OpenerView[{Arguments, Column[subproofs_, ___]}, ___]}, ___},
    ___]] := subproofs;
subproofs[proof_] := {};
```

To clear up a technical detail: the opener view is the collapsable arrow or triangle shape leading to more text display, here in a nested form when viewing in a Mathematica notebook. In a way, the pattern to tree functions are all about this un-collapsing of the grid structure, representing a mathematical proof in Lean. For completeness, here is another level of the grid expanded and viewable in PDF-file-format to illustrate the functionality. (This structure will become important again once we look at recursion.)

Out[ ]=

```
{
  Goal | p : ℕ
  ID   | 0
  Rule | Assumption
,
  Goal   | ∀ {m n : ℕ}, nat.prime p → p ∣ m ^ n → p ∣ m
  ID     | 23
  Rule   | ∀I
  Proofs | ▼ Arguments
             Goal | m : ℕ
             ID   | 1
             Rule | Assumption

             Goal   | ∀ {n : ℕ}, nat.prime p → p ∣ m ^ n → p ∣ m
             ID     | 22
             Rule   | ∀I
             Proofs | ▼ Arguments
                        Goal | n : ℕ
                        ID   | 2
                        Rule | Assumption

                        Goal   | nat.prime p → p ∣ m ^ n → p ∣ m
                        ID     | 21
                        Rule   | ∀I
                        Proofs | ▶ Arguments
}
```

On the level of analysis of the function code however, ... TODO TR

Recursion

TODO FR

Limitations and Summary

This study did not cover numerics, graphics programming, front end programming, or writing packages, all major topics related to the current discussion: An Introduction to Programming with Mathematica (2005, Wellin et al.) completes the picture at the intended overview level of analysis. The focus here has been on one particular example, solving one problem, in Wolfram Language; it is involved enough to be viewed in context of the major themes chosen after introducing the language in broad strokes. These themes were lists; functional programming as considered for itself but also relative to other paradigms; rule-based programming as the particularly helpful approach in this situation; and recursion, which we could not have easily done without.

References

Heseltine, J. (2023) *Empirical metamathematics: extending the Lean-to-Mathematica bridge.* Wolfram Community. Available at: https://community.wolfram.com/groups/-/m/t/2957419 (Accessed:

26 July 2023).
Mangano, S. (2010) *Mathematica Cookbook.* Sebastopol, CA: O'Reilly Media, Inc.
Wellin, P. R. (2005) *An Introduction to Programming in Mathematica.* Cambridge, UK: Cambridge University Press.
Wolfram Research (2023) Wolfram Language & System Documentation Center. Available at: https://reference.wolfram.com/language/ (Accessed: 23 July 2023).