5. Conditionals, or how to say Maybe

Why we've included this: we have to cope with conditional structures quite often in everyday life, and if you misunderstand there are consequences (Figure 16). Even more so in programming and other formalised situations.

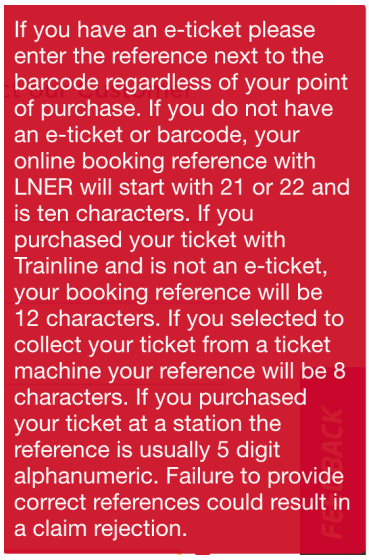

Figure 16. A message from LNER, a British railway company, about reclaiming the cost of unused rail tickets. Note the alarming choice of background colour.

Here we're going to look at structures of the form:

S1: If it's raining, take an umbrella; otherwise take sunglasses

and at slightly more complex structures:

S2: If it's raining, take an umbrella if you have one, otherwise take a raincoat; but if it's not raining take sunglasses

S3: If it's raining, take an umbrella if you have one, otherwise take a raincoat; but if it's not raining take sunglasses. If you're a pessimist, take a raincoat anyway.

These are called conditionals because what you're supposed to do depends on the condition of the weather. In normal conversation or writing, people can express a conditional in varied ways and are often neither very clear nor fully complete, relying on the listener to make sense of it and to assume sensible defaults for any possibilities not mentioned – for example, S3 might mean 'take a raincoat even if you're taking sunglasses' or 'take a raincoat even if you've got an umbrella'; but we have to use precise notational versions for many purposes. in which everything is specified, so we're going to look rather closely at some of the possibilities.

We are looking at them, for one reason, because they are pervasive; conditionals are an important part of everyday life, not just of the world of coding – in fact, any notation that is used to give instructions has to have some kind of mechanism to express conditionals. And for a second reason, because they're difficult. Knowing which choice to make is often difficult, as we all know, and if the issues are presented unhelpfully matters will be worse. Whether to take an umbrella is hardly a matter of life and death, but many situations are just that: dealing with official regulations (eg for displaced persons) or medical guidance for emergencies are

just two examples. In between is a whole spectrum of real-world cases, and we shall make use of one such – rules for computing postage for various sizes and weights, in the UK postal service, a rule that is consulted by many people every day. Some indication of both the importance and the difficulty of conditionals – 'rules about when to do what' – is given by the existence of many companies who make it their livelihood to analyse business and official procedures and transform them into 'business rules', frequently featuring a wide range of visualisations.

Here are some notational ways to express the S3 'weather rule'. First, Figure 17 shows one of several possible ways to rewrite S3 in the sort of syntax used by many programming languages. Because it's so simple, this looks perfectly easy to understand even if you don't know anything about coding.

```
if (raining)
    if (got_umbrella) take_umbrella
    else {take_raincoat
else take_sunglasses
    if (pessimist) take raincoat
```

Figure 17. The S3 weather rule in an approximation of programming syntax. There are many other possible programming syntaxes, some similar, some very different. This structure will be called 'sequential'. Note the nested conditionals are well indented.

The design of the notation of Figure 17 is heavily directional, inviting the reader to work through from the top, finding out what is the first thing to do, and then the next, and so on, in sequence; hence we shall this a 'sequential' notation. Notice that the reader has to work right through the structure, in case a later part contains another relevant instruction.

Small digression. Before we continue, we'd like to note that many coding systems use squiggly brackets, as well as indenting, to show how fragments are nested (Figure 18). We mention this because some people have strong feelings about whether to use brackets or not, and how to format them if so. There is no relevant evidence that we know of, so we shall not pursue that. *End digression*

```
if (sunny) {
   if (got_sunglasses) {
      take_sunglasses
      if (sunlotion) {take_ sunlotion}
   }
}
else {take_raincoat}
```

Figure 18. A rule using squiggly brackets to show structure; left, brackets replacing instead of indenting, right brackets supplemented by indenting

Figure 19 shows a 'decision table' equivalent to Figure 17. This is a bit different; the possible actions are listed and the conditions required for that action are shown attached.

conditions	raining	Y	Y	Y	N	N
	got an umbrella	Y	Y	N	-	-
	pessimistic	Y	N	-	Y	N
actions	take umbrella	X	X			
	take raincoat	X		X	X	
	take sunglasses				X	X

Figure 19. A decision table equivalent to the S3 weather rule. You work out which column agrees with your state of affairs, and then do the action(s) marked in the lower part of that column. The symbol '–' stands for 'doesn't matter'. This is a 'circumstantial' structure.

The decision table is designed to be read left to right, top to bottom, so it's another directional notation: but in this instance the reader is expected to sort out all the conditionals before reaching the action – to ascertain the particular circumstances that apply in a particular instance; so we shall call it a 'circumstantial' notation.

Each of these representations can easily be turned into one of the other, but nevertheless they have different cognitive and perceptual properties ('affordances') and they encourage different approaches.

A textual version of circumstantial structure can be created by writing out all the possibilities using ands and ors:

```
if raining and got an umbrella:   take umbrella
if raining and not got an umbrella:     take raincoat
if not raining and not pessimistic:     take sunglasses
if not raining and pessimistic:   take raincoat and sunglasses
```

Figure 20. A circumstantial structure in text form

We should also include the 'GOTO' command as used widely in early programming languages. It is still used in official form design, where it is not not unusual to find structures like this:

Q 14: Do you own any pets? Y / N
If 'No', go to Q 17
Q 15: please give a list of all the pets you own.
Q 16: do any of your pets qualify as licensed Emotional Support Animals?
Q 17: .....

In the programming world, arbitrary use of GOTO commands has become very unfashionable because it can be very hard to understand a complex program using GOTOs; nevertheless, some programming languages permit arbitrary GOTOs, including legacy languages like COBOL, still responsible for a sizeable proportion of industry software. (In addition many programming languages permit carefully circumscribed constructs that are disguised GOTOs like the 'break' command.) Here is the 'weather rule' expressed using GOTOs:

```
    if raining goto L1
    take sunglasses
    if not pessimistic goto L3
    take raincoat
    goto L3
L1: if got_umbrella goto L2
    take raincoat
    goto L3
L2: take umbrella
L3: finish
```

Figure 21. A GOTO program equivalent to Figure 17

Figure 21 is a sequential structure *par excellence*.

Expressing conditionals with graphics

The distinction between 'graphical' and 'textual' notations is not all that clear-cut, since text is often formatted or tabulated and graphics include text; for now, let's say that the graphical

notations are those that include lines. (Meaningful lines; lines surrounding a table are just visual markers without any actual meaning.)

The same meaning can be presented graphically. Figure 22 is a 'decision tree'; like the programming-like notation of Figure 17, it is heavily directional, and also like Figure 17, it invites the reader to work though from the top, so this is a sequential structure.

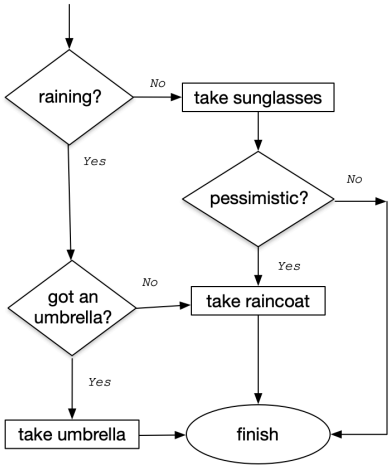

Figure 22. A 'decision tree' equivalent to Figure 17

Midway between textual notations and graphical ones is what we shall call 'formatted text' (Figure 23). Although rare, this format is sometimes used as one of the formats offered by business rule companies. In our judgment, it is not strongly directional, feeling equally easy to read in either direction: we have no empirical evidence on that, but we shall refrain from classifying it as either circumstantial or sequential.

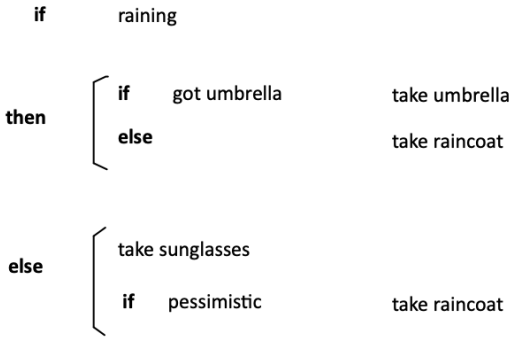

Figure 23. 'Formatted text' version of the S3 weather rule , a format midway between ordinary text and graphics.

A very different graphical style is the 'ladder diagram', whose origins go back to the days when combinations of relays and switches were used to control electro-mechanical devices. At that time, a ladder diagram was a conventionalised representation of an actual, physical circuit; for example, a circuit in which one button started a motor, and a second button stopped it. Because they were conventionalised, ladder diagrams were easier to reason about than a plain tangle of wires and components. Here is a simple ladder (Figure 24). The vertical lines L1, L2 are considered to be power rails and the horizontal 'rung' contains the components: two switches, A and B, which are off by default, and a lamp, C. When either A or B is switched on, the lamp lights. One can read that as: "if A is true *or* B is true, then do C".

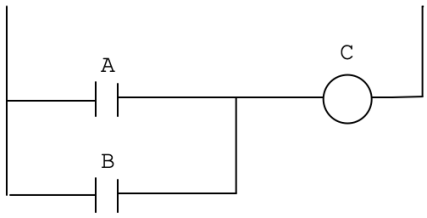

Figure 24. ladder diagram for 'A *or* B': A and B are 'normally open' buttons, switches or relay contacts. if either button is pressed, bulb C lights up.
needs rays on the lamp

Figure 25 is a diagram for a circuit in which both A and B have to be switched on (Figure 25). Read as "if A is true *and* B is true, then do C"

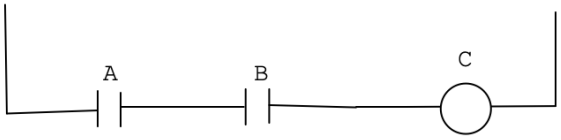

Figure 25. Ladder diagram for 'A *and* B': if both buttons are pressed at once, bulb C lights up

The last symbol we shall need is the inverted switch, one that is on by default. In Figure 26 the switch A is inverted, and 'current' can 'flow' if the switch is *not* pressed. (In electrical terms, that would be a normally-closed switch: press to open it.) The lamp is normally lit and goes off if A is pressed; ie, lit if not pressed. Read that as "if A is false, then do C".

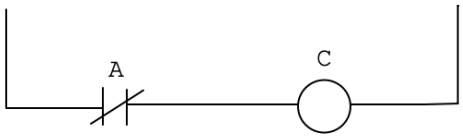

Figure 26. An inverted switch: bulb C lights up *unless* button A is pressed (because A Is normally closed)

More complex structures will require more rungs, of course, and here (Figure 27) is a ladder diagram for our weather rule. To make sure we followed conventions, the diagram was constructed using online software. Notice that the same virtual 'component' can appear on more than one rung – for example, the top two rungs share all their components – just as, in the textual forms, the same condition may be mentioned more then once.

Ladder diagrams are read from top to bottom, trying all the conditions in turn, so this is another rather directional, circumstantial notation.

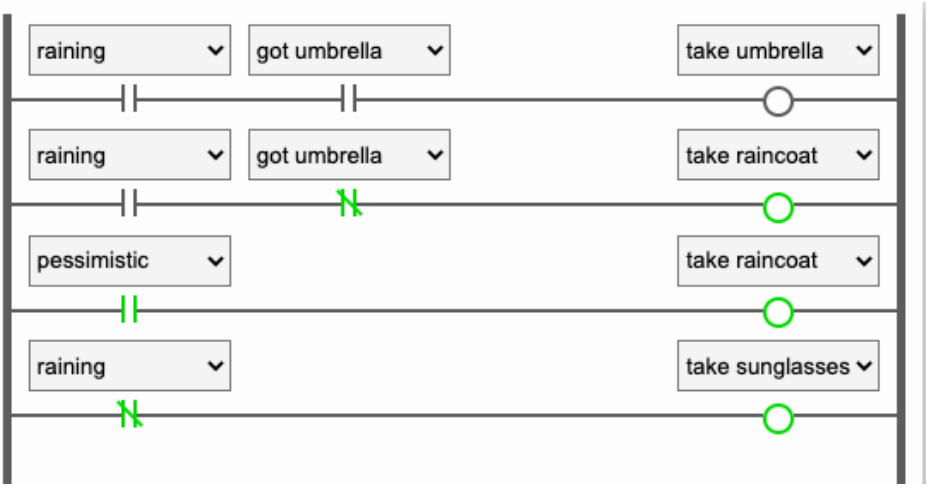

Figure 27. The weather rule as a ladder diagram. Green indicates which variables are true (= which buttons are pressed), so here we have not got an umbrella, we are pessimistic, and it is not raining; likewise, green indicates which 'lamps' are lit, so take a raincoat and take sunglasses. Observe that in this format the same component can appear on more than one rung, sometimes inverted, as in the bottom rung. Taken from https://www.plcfiddle.com

The world of electro-mechanical switches and relays has changed into that of programmable logic circuits (PLCs), containing a much wider variety of components and using integrated circuits. Ladder diagrams are still widely used, but they may contain hundreds or even thousands of rungs. At the same time, a notation with a slightly higher level of abstraction has gained favour, in which discrete logic components have replaced the ladder diagram, notably the three types used in Figure 28: a *not*-gate sends true if its input is false, an *and*-gate sends true if both its inputs are true, and an *or*-gate sends true if either of its inputs is false. This is a heavily directional notation, presumably because it only allows each virtual component to be mentioned once; fairly easy to read from left to right but harder to read from right to left
EXPAND HERE why AND is a fork

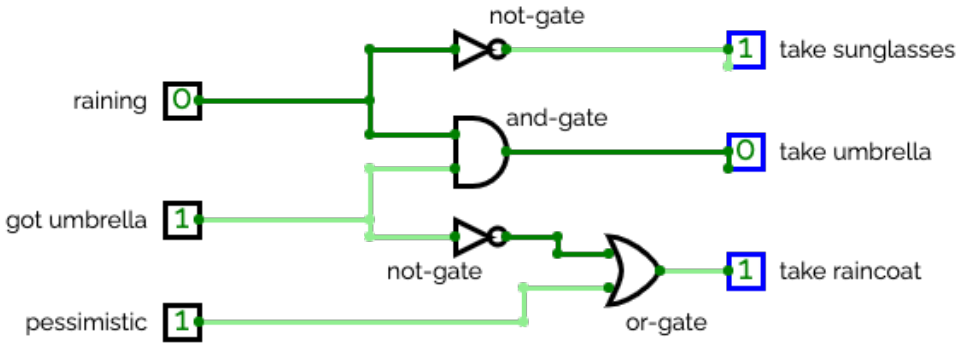

Figure 28. A logic diagram for the weather rule. Zero means false, one means true; pale green lines carry a 'true' signal, dark green ones carry false. https://circuitverse.org/simulator/edit/weather-rule-v1 REDRAW IN Omnigraffle

The idea of power flowing through wires, which was the basis of the ladder diagram, has been almost lost in the logic diagram. Instead we have signals propagated along lines either 'true' or 'false'. This is an example of a data flow diagram, of which more later when we come to visual programming.

Although the differences between different structures are trivial when the problem is sufficiently small, they quickly become significant as problems more complex – and real world issues are frequently extremely complex. Above, as a real world example of a conditional, we mentioned the computation of postage for UK standard letters, and although it is only slightly more complex than the weather rule, a few moments of experimentation will quickly reveal big differences between the structures we have described.

Format and max measurements	Max Weight	1st Class	2nd Class
Letter 24 cm long 16.5cm wide 0.5cm thick	100g	85p	66p
Large letters 35.3cm long 25cm wide 2.5cm thick	100g	£1.29	96p
	250g	£1.83	£1.53
	500g	£2.39	£1.99
	750g	£3.30	£2.70

Figure 29. UK postage costs as at March 2022. (Taken from the Post Office website https://www.postoffice.co.uk/mail/uk-standard do we need permission?)

Conditionals and empirical research

How do these different formats compare? Are there any relevant experiments? It turns out that a good deal of research has been published, but we are going to be extremely selective. The major question is, whether there is one single best format for conditionals, and it would be nice to be able to say that such-and-such a design for conditionals is always the easiest to use, but such superlativism seems to be a forlorn hope; it is rather that one design is better for some purposes, and another design for other purposes. (That, indeed, is the whole message of the cognitive dimensions framework.) We note that the various syntactic designs for conditionals come from rather different communities: forms of tabulation come from the world of information design; the program-like versions come from the world of coding; and the logic diagram and the ladder diagram come from electrical and electronic engineering worlds. These different worlds have different concerns. In particular information design is far more concerned with comprehensibility, how easy it will be for readers to understand the information; the coding world, in contrast, is at least as much concerned with how easy it is for the writer to solve the problem of expressing complex contingencies in a constrained notation, or how easy it will be to modify a structure in the future, if the requirements should change.

Information designed for use by the general public should obviously be tested by members of the general public, rather than by specialists: program structures intended for use by specialists should obviously be tested by specialists; program structures intended for use by

non-specialists or by students learning programming should be tested by those groups, and so on. Obvious thought that may seem, many of the early studies of program structures overlooked this issue and generalised from computer science students to the rest of the population.

Upstream/downstream comprehension

I have copied some of this into 'Reasoning from notations' in Part 3, so this needs working Most, though not all, of the empirical studies focused on comprehension, and they did so by getting their participants to answer questions of the form 'it *is* raining and you *do* have sunglasses but you do *not* have an umbrella, what should you do?'; ie the questions presented truth values and asked what the outcome of the program, or the instructional rule, would be. At first sight, those are the obvious questions to test. But thinking a bit further, one soon realises that other questions are also very important. Examples might be, "I want to fulfil the conditions to be awarded a grant for improving my house", "I want to fulfil the conditions to be allowed into this country", or "my program did the wrong thing, why?" These are questions of the form 'this is what happened: what must the conditions have been, to get that result?'. In the context of the sequential versions, such as Figure 17 and Figure 22, a downstream question might be 'it's not raining, what should she do?', versus upstream – 'she took her umbrella, what must the weather have been?', or 'this is what I want to happen, what must I do?' Here, 'downstream' denotes the easy way to follow through the code; 'upstream', the hard way. For the sequential versions, downstream equals forwards reasoning, and upstream equals backwards reasoning, terms introduced in the theory of problem-solving developed by Mayer (1976 iirc). When we come to the written-out circumstantial form of Figure 20, however, the directionality is reversed. The easy, downstream direction is 'she took her umbrella, what must the weather have been?', and the harder, upstream direction is, 'it's not raining, what should she do?' See Figure 30.

direction of reasoning	*sample question*	*sequential structure*	*circumstantial structure*
from conditions to consequences	it's not raining, what should she do?	forwards = downstream	backwards = upstream
from consequences to conditions	she took her umbrella, what can we infer about the weather?	backwards = upstream	forwards = downstream

Figure 30. Forwards and backwards, upstream and downstream

Sherlock Holmes specialised in upstream reasoning, but for the rest of us it's often rather difficult, and we can confidently predict that upstream questions will be harder (ie take longer, cause more mistakes) than downstream. Once we have thought about it like this, it is clear that if upstream is harder than downstream, we can expect a cross-over effect: questions that are easy to answer from the sequential notations will be hard to answer from the circumstantial, and vice versa.

During the 1970s and 1980s there was considerable enthusiasm for graphical notations, and it seemed that quite a few people believed that one notation was the best, *tout court*. In consequence quite a few empirical studies compared the comprehensibility of program-like instructions in graphical and textual formats, with rather varying results; some researchers reported that flowchart-like structures were better on one measure or another, other researchers reported the opposite, and some researchers found no worthwhile difference. In

our opinion, questions like ‘is textual better/worse than graphic’ are much too vague to be answered sensibly; the wide variety of notations of both forms completely rules out any simple answer. On the other hand, we do think it very likely that upstream comprehension is always harder than downstream comprehension, whatever the format; although the size of the effect might well be very different for different formats, we would be very surprised if it was ever reversed. Unfortunately very few studies have explicitly compared upstream questions versus downstream questions. Our highly selective survey of empirical results therefore focuses on those studies in which upstream was compared to downstream, in (selected) graphical and textual formats.

Following on earlier work by Sime, Green and Guest REF, Green (REF1, REF2) compared comprehension speed for professional programmers answering questions about (a) sequential textual programs written in the GOTO format of Figure 21, (b) programs written in the format similar to Figure 17, and (c) an extended syntax they called ‘Nest-INE’ (Figure 31), which was like the Figure 17 notations but included some elements of the circumstantial structure of Figure 20. There was very little difference in answering downstream questions, but upstream ones were much easier in the (c) style, whose extended and more circumstantial syntax helped greatly in answering the upstream questions, exactly as expected.

```
if raining:
    if got_umbrella: take_umbrella
    not got_umbrella: take_raincoat
    end got_umbrella
not raining:
    take_sunglasses
    if pessimist: take raincoat
    end pessimist
end raining
```

Figure 31. The weather rule expressed in the ‘Nest-INE’ syntax

That work was extended in a very thorough study by Curtis et al {Curtis et al., 1989, #87409}, who also found that the direction (upstream/downstream) had an effect, albeit much smaller than the differences between individuals. Although they used both textual and graphical formats, their programs used loop structures and data flow as well as conditionals, so the results are probably complicated by the need for participants to understand more about the programs. text/graphics

Finally Green and Petre (1992) reported a study comparing sequential and circumstantial versions of text conditionals, and sequential and circumstantial versions of graphical conditionals. Sequential text used the Nest-INE syntax that Green used; circumstantial text was in the form of Figure 20; and circumstantial graphics resembled Figure 28. That leaves the sequential graphics notation, which used a notation that we believe is unique to the programming language LabView. It was an interactive notation with a limited field of view, such that only one arm of the condition structure could be viewed at at a time and the user had to use the mouse to move the viewpoint from one arm to another. Figure 32 shows the weather tule expressed in that notation. Each of the rectangles expresses one of the possible cases. In Figure 32 the case is that ‘raining?’ is true, ‘got umbrella?’ is also true, and ‘pessimistic?’ is not tested; the output of this ‘case structure’ is to send ‘false’ to ‘take sunglasses’ and ‘take raincoat‘ and ‘true’ to ‘take umbrella’. To see what happens in other conditions, the user has to use the mouse to change the truth values.

Since it is hard to understand such an interactive notation by merely describing it on paper, we have set up an emulation of the weather rule: visit https://tgworkshop.me.uk/bookdemos/conditionals/labview/ to explore its working.

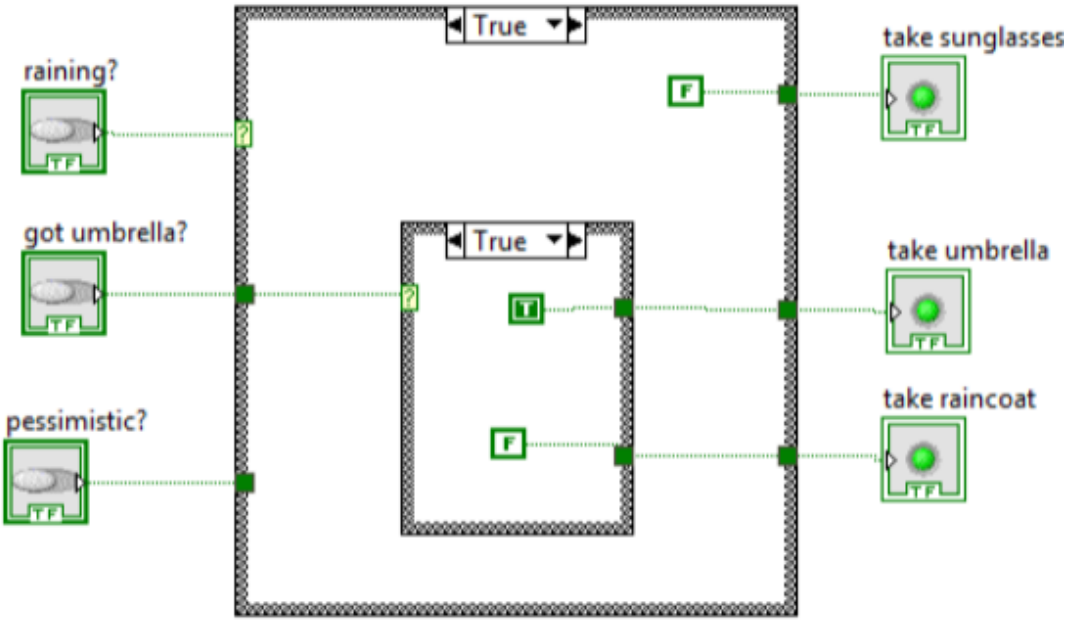

Figure 32. The weather rule expressed in the sequential graphical notation used by Green and Petre. This notation is interactive – mouse clicks on the true/false buttons toggle between the true and false arms of the conditionals. Only one arm is visible at one time. See https://tgworkshop.me.uk/bookdemos/conditionals/labview/ for a live example. With three input variables, there are $2^3 = 8$ possibilities, so there are eight possible arms, only one of wich can be viewed at any one time. The program used by Green and Petre contained six input variables, so there were $2^6 = 64$ arms. INTERACTIVE

The programs tested by Green and Petre were considerably more complex than the weather rule. Figure 33 shows one of their programs, using the Nest-INE notation that had previously been used by Sime et al and by Green, and Figure 34 shows the circumstantial graphical notation.

```
if high :
   if wide :
      if deep : weep
      not deep :
         if tall : weep
         not tall : cluck
         end tall
      end deep
   not wide :
      if long :
         if thick : gasp
         not thick : roar
         end thick
      not long :
         if thick : sigh
         not thick : gasp
         end thick
      end long
   end wide
not high :
   if tall : burp
   not tall : hiccup
   end tall
end high
```

Figure 33. The sequential textual notation ('Nest-INE') used by Green and Petre (REF). To understand the difficulties of backwards (upstream) reasoning, you are invited to discover the conditions under which the outcome is, say, 'hiccup'.

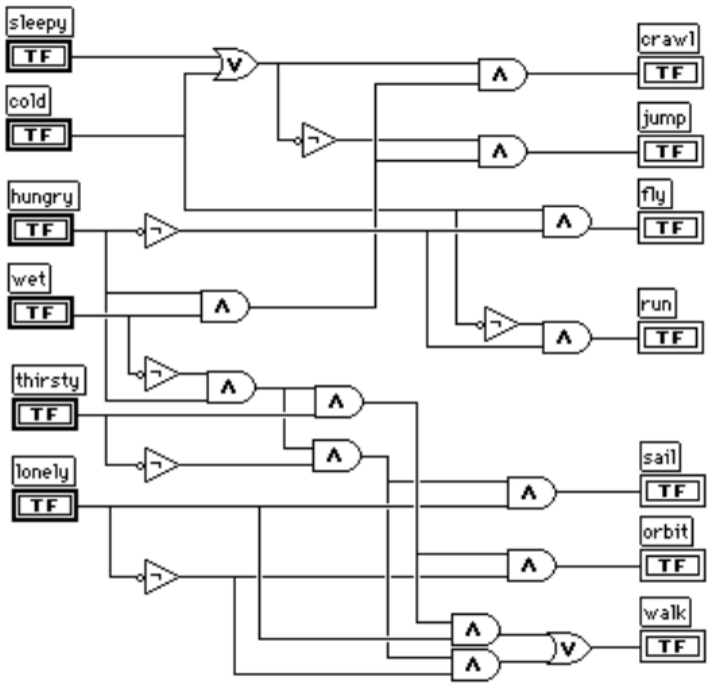

Figure 34. The circumstantial graphical notation used by Green and Petre. Cf Figure 28 This is the full size version they used; see Figure 28 for the weather rule expressed in this form. To understand the difficulties of backwards (upstream) reasoning, you are invited to discover the conditions under which the outcome is, say, 'orbit'. Is Fig 12 really circumstantial?

Remembering that forwards questions are downstream in sequential notations but upstream in circumstantial notations (and vice versa), as laid out in Figure 30, we would predict an interaction effect: that is, rather than finding that forwards were easier or harder than

backwards, we would find that forwards were easier when that was the downstream direction, and so on. Schematically, we predicted that the results would follow this pattern (Figure 35):

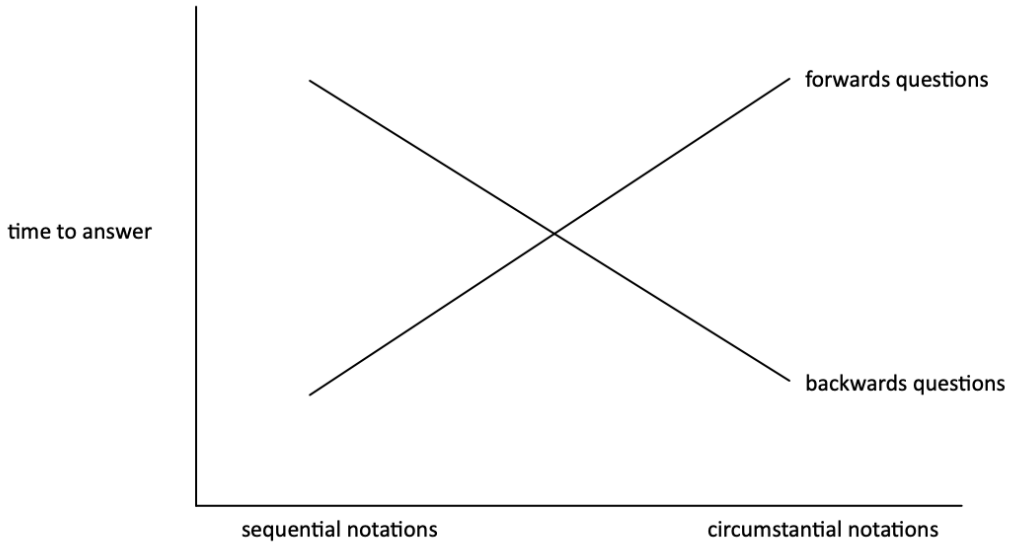

Figure 35. Schematic prediction for response times.

The results were exactly as predicted, a significant interaction effect, for both graphical and textual versions of notation. Unexpectedly, a much strong result was that graphic notations were always slower: whatever the combination of notational direction and question, times for the text-based notations were always quicker than for graphical.

Green and Petre also compared 'same-different' judgements, where the participants were asked whether two programs were the same or different. The most important of their results was that both the graphical notations were hard to work with, especially the circumstantial version. "The sight of subjects crawling over the screen with mouse or with fingers, talking aloud to keep their working memory updated, was remarkable. This structure was very difficult for our subjects, *even for [the participants] who were very experienced with it*. " (Their emphasis.)

Other intriguing findings

Although in some cases very large differences in comprehensibility were found (notably in the study by Petre and Green), Curtis et al found that the differences between notations were much smaller than the differences between the participants. But to our mind, that result does not argue for disregarding the effects of notational design. On the contrary: the less able the notation user, the more important it must surely be to provide the best design for the task at hand.

One particular type of possible individual difference that has occasionally been spoken of is the possibility of gender differences, such that one type of notation might be better for females and another type for males. We suspect that life is rarely that simple, but as we have not yet seen hard evidence concerning gender differences, we have to keep an open mind.

It has been strongly argued by Vessey and her co-workers that answering questions about conditionals and similar structures is affected by 'cognitive fit', meaning that if the external representation of the task matches the internal (mental) representation performance will be

better. For example, graphically-stated problems may quite possibly be best handled with graphically-presented instructions.They have reported ample evidence to support this thesis REF, and indeed it would be surprising were it not true. “Designers should ... concentrate on determining the characteristics of the tasks that problem solvers must address, and on supporting those tasks with the appropriate problem representations and support tools.” In the present context the implication is that designers of notations should be aware of the context in which the notation will be used, and where feasible should try to match the notation to the problem format.

Lastly, there is what appears to be a solid result that deserves further investigation. Wright and Reid REF, who were the first we know of to study this aspect of notational design, compared four formats: a circumstantial design akin to Figure 20; a graphical decision tree similar to Figure 22; a convincing mimicry of bureaucratic prose, and a terse restatement of the latter in clear, short sentences. Besides testing comprehension with the notation on display, they also asked their participants to memorise the material and then tested comprehension again, with the participants working from memory. Their main result was that clear short sentences were overall best, but they also had a surprise. Their surprise result was that at the beginning of working from memory, differences were slight, but as the trials continued participants made fewer errors on the two pure text formats (bureaucratic prose and clear short sentences), as though the participants were learning the structures; while on the graphical and tabular formats errors *increased* from trial to trial, suggesting that those structures are harder to keep in mind and that the mental representation dissolves bit by bit. One can imagine that happening because text can be repeated to oneself to keep it mind (the ‘phonological loop’ of the well-known working memory model), but spatial material cannot be rehearsed in the same way. We don’t know of any follow-up to that finding, which is unfortunate because in all sorts of real-life situations it is clearly useful to be able to work from memory.

Cognitive dimensions and conditional structures

Testing every notational design empirically is quite infeasible, so designers have to choose what they believe is likely to work best. Moreover, they have to manage compromises of various sorts. With the help of the cognitive dimensions framework we shall illustrate how some of the available designs have chosen their trade-offs. This is not meant to belittle any of the designs we mention, but instead to highlight some of the difficulties and to alert potential users and potential creators of new designs to some of the possible consequences of design decisions. Some of these systems, such as spreadsheets, are discussed in much greater detail in other sections of this book – here we shall only consider their treatment of conditionals.

Perceptual aspects

First we consider supplementary recoding, secondary notation, and perceptual parsing, three intimately related dimensions; and our thoughts turn immediately to the common spreadsheet.

We shall use the ‘large letter’ fragment of the UK Post Office postage charge table (Figure 29) to show how solutions appear in particular conditional designs. This is the fragment in question (Figure 36).

Weight	1st Class	2nd Class

negative	*too light*	
0 - 100g	£1.29	96p
101 - 250g	£1.83	£1.53
251 - 500g	£2.39	£1.99
501 - 750g	£3.30	£2.70
over 750g	*overweight*	

Figure 36. Cost of posting a ‘large letter’ in the UK, spring 2022. Taken from the Post Office information display. The two lines in italics were added by the authors.

Spreadsheets containing long formulas have the problem that they need to get all the formula into one cell of the sheet, and the cells are of a size to hold numbers and short words rather than formulas. They also have a problem of upholding tradition (‘backwards compatibility’), so that new spreadsheet apps do not depart too far from older ones; the earliest spreadsheets used capital letters and suppressed spaces in formulas, so some present-day ones do the same. The result is that a cell for computing large-letter postage might contain a long formula like Figure 37.

```
=IF(A3>750,"overweight",IF(A3>500,"£3.30/£2.70",IF(A3>250,"£2.39/£1.99",IF(A3>100,"£1.83/£1.53",IF(A3>0,"£1.29/0.96","too light")))))
```

Figure 37. A spreadsheet formula for postage, in LibreOffice. Not easy reading!

That is obviously hard to read, and for the coder, it is hard to be sure that the parentheses match up correctly. In the case of LibreOffice (but not all spreadsheet systems) it is possible to insert – by hand – ‘soft’ newlines when felt appropriate, so that a layout like Figure 38 is possible, though it’s a fiddly task. The hand-inserted newlines offer a small degree of supplementary recoding to assist perceptual parsing.

```
=IF(A3>750,"overweight",
 IF(A3>500,"£3.30/£2.70",
 IF(A3>250,"£2.39/£1.99",
 IF(A3>100,"£1.83/£1.53",
      IF(A3>0,"£1.29/0.96",
               "too light")))))
```

Figure 38. Same formula as Figure 37, with soft newlines inserted by hand.

Some spreadsheet systems, including LibreOffice, enforce the use of capital letters; others allow some components to be written in lower-case. The received opinion, according to Wikipedia, is that uppercase-only text is less legible because of poor discriminability. Presumably the designers of LibreOffice and other upper-case-only systems were anxious to preserve backwards compatibility rather than improve legibility.

One spreadsheet system that instead valorises legibility over tradition is the Apple spreadsheet for Macs, called Numbers. If that same text from LibreOffice is dropped into a Numbers spreadsheet, it is changed to look like Figure 39, where clearly the legibility has been improved.

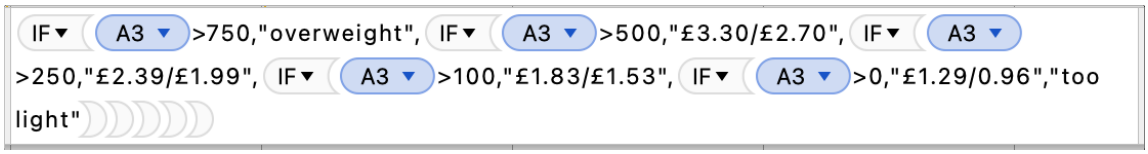

Figure 39. The same formula as seen in the MacOS app 'Numbers'.

To assist the reader – and therefore the writer – Numbers has a further feature. If one of the IF commands is highlighted, all the IFs nested below it are also highlighted, helping to show the syntactic structure (Figure 40).

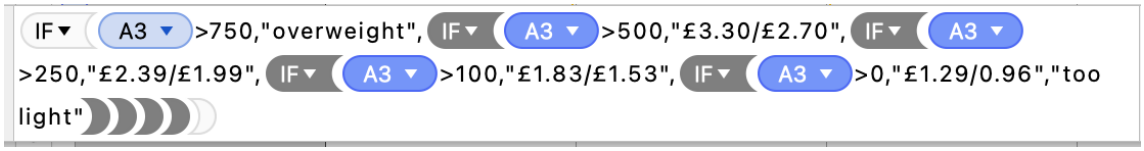

Figure 40. The second occurrence of an IF has been selected, causing all the nested IFs to be highlighted.

If soft newlines are inserted into the formula, Numbers automatically indents the layout (Figure 41).

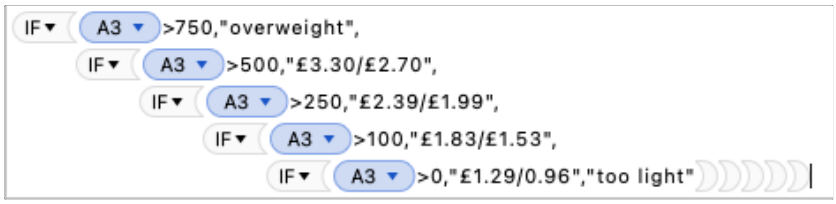

Figure 41. When soft newlines are inserted manually in Numbers, the layout is automatically indented

The Numbers presentation of conditionals is the best we know of in commercial spreadsheets, although perhaps it could be improved by offering a feature to insert newlines automatically on demand.

Secondary Notation

The optional newlines just illustrated are one of the few possibilities for secondary notation in spreadsheets; some graphical notation editors also make it hard to add comments. The textual versions, in contrast, make it very easy to add comments:

```
if (raining) {
// first we deal with raining, because that is the usual case on
holiday
    if (got_umbrella) {take_umbrella}
    else {take_raincoat}
}
else {take_sunglasses
// but if it isn't raining, for once, we don't wanted to be blinded, so
sunglasses needed
    if (pessimist) take raincoat}
}
```

Figure 42. Comments can easily be added to textual notations

Visibility and Juxtaposition vs. Space and Diffuseness

The space problem is ever-present for the designers of visual languages, and conditional structures can easily consume more than is acceptable. Complex conditionals are especially problematic for all graphical notations, because they use up so much screen space, and a variety of space-saving dodges are used. Folding the conditional up and putting it away is one possibility, well illustrated by the trade-off choices made by the designers of LabView, who restricted the field of view to exactly one case at a time, as seen above. The result, of course, is a dramatic loss of visibility and juxtaposability. How does one compare one conditional expression with another to make sure that between them they cover all relevant cases? In practice that may not be too frequent a problem. LabView's strengths are its wide variety of virtual instruments and its strong wired-circuit metaphor, and it is unlikely that compound conditionals play great part in the typical LabView world. Still, it would have been nice to have an option to show more than one arm at a time when juxtaposed comparisons were needed.

The spreadsheet solution is to pack up the formula into a single cell, normally hidden; this, too, sacrifices visibility and juxtaposability. In contrast, formatted text (Figure 23), ladder diagrams (Figure 27) and logic diagrams (Figure 28) just accept the space requirement, and make it an easy matter to compare two or more conditionals when necessary.

Hard Mental Operations

Conditionals are not easy to think about, and they can readily create hard mental operations in upstream reasoning, even when downstream working is easy. Working downstream in a sequential notation such as Figure 17, one starts at the first line; if 'raining' is true, one goes on to the next line, otherwise one skips to the next `else` that is at the same level of indentation, and so on until an action has been performed. In well-formatted text where the indenting is accurate this will not be a hard task. Now consider working upstream, say from 'take raincoat'. Travelling left, first one meets an `if pessimist`. Keeping 'pessimist' in mind, in a sort of mental shopping basket, and continuing to travel left and up, one meets an `else`. That indicates that the next condition must be negated; keep that in mind. Next one moves upward to the matching `if`, and add 'raining to the shopping basket – remembering to negate it first. Final answer 'not raining; pessimist'. Clearly there is more work here than in the downstream direction.

But the problems rack up when working upstream in the logic diagram notation of Figure 28. The problem is keeping one's place. MORE HERE using the circumstantial notation of Figure 34 was extremely difficult, even for experienced users.n

Second, what about writing the code or drawing the diagram? Does that too involve hard mental operations? To the authors' surprise, creating the decision tree took a bit of thinking through, even for such a simple example as Figure 22; unfortunately there is not much empirical data on the difficulty of creating graphical programs, as opposed to understanding them once created. Even if there were, it would be hard to know how much difficulty might be caused by the tedious business of selecting and laying out the component parts (and-gates and wires, etc) and how much would be irreducible cognitive difficulty.

But the textual notations, too, have their problems. goto ... Luke's record ...

Viscosity

Viscosity always depends on the editing environment as well as the notation. Advanced editors for text-based notations are available, but editors for graphical notations are a much harder problem because even something so simple as adding another component (pure incrementation) can mean that the layout has to be painstakingly reworked – unless an unusually clever editing program is capable of organising the diagram sensibly. Some comments on this in the graphical programming essay? The logic diagram (Figure 28) is an example of such a notation. However, the ladder diagram has very low viscosity for simple incrementation, ie adding another rung to deal with another combination of circumstances. I'd better find a user to check that with

As usual text notations are less viscous than most graphical ones. Injecting another `IF` into the code of Figure 42 is fairly easy, and the same goes for Figure 37's spreadsheet code – although if the editing environment automatically provided a bested layout it would be easier.

Closeness of mapping

Notational conditionals resemble everyday thought reasonably closely but not perfectly. For a start, notational conditionals need to be precise and unambiguous, as noted at the start of this essay. In principle a conditional should cover every possible combination of input conditions – indeed, some notations indeed make it impossible to leave some outcomes undefined – but everyday expressions can readily leave some options unspecified, as in Figure 17.

A deeper problem sometimes surfaces, the 'paradoxes of material implication'. Natural language can express conditionals in many ways, not just 'if ... then' but also 'whenever ...', 'in the event that ..', and so on, and the connotations differ – for example, 'in the event that X ...' is usually used when X is a relatively unusual occurrence, as in 'In the event of fire, do not use this lift'. Formal logic uses much more strictly defined expressions. Those logic systems that express 'if P then Q' as 'P implies Q' and define 'P implies Q' to be a true statement unless P is true and Q false (which looks a reasonable definition at first blush) run into these paradoxes: if P is not in fact true, then 'P implies Q' is true by definition, whatever Q might be. (Eg, 'if the week had 8 days then the earth would have 2 moons' must be a true assertion, since the week does not have 8 days.) Anyone who is reasoning backwards through conditionals needs to avoid drawing false inferences do I need to spell this out? or maybe too much already?

Abstractions

None of these conditional notations welcomes abstractions.

Summary

These are the main points emerging from this discussion of conditional structures.

Observations

First, some conditional notations are strongly directional, making it much easier to traverse then downstream than upstream; others are more weakly directional; the 'formatted text' notation of Figure 23 seems hardly directional at all (but remember, we have actual data on that). The concept of directionality has not previously been introduced in the the study of notations, as far as we aware, though it is hardly arcane.

Second, among the directional notations, some are sequential in nature while others are circumstantial. As a result, there is a cross-over in difficulty of traversal, as seen above FIGURE TO COME In some cases upstream traversal is so difficult, thanks to a combination of hard mental operations and the need o search, that it is nearly impossible except for small cases.

Third, for the visual notations, managing the trade-off between amount of space used and juxtaposability / visibility still appears to need better solutions than the ones we have found.

Lastly, we draw attention to the development of the ladder diagram, which started as a fairly direct representation of relay switch-work, then became more abstract, with the same notation being used to refer to more modern circuitry, and then became the logic diagram, with the ladder rungs replaced by data flow between logic gates – only for that in turn to become a network of higher-level components. The essay on 'notational lifecycles' ETC

Suggestions

Where does that leave us? We would suggest that notation designers contemplating the problems of introducing conditionals should perhaps strive for a notation that was not too strongly directional. One way is to introduce cues to the other direction. In the highly sequential if-then-else notation, we found evidence that introducing cues to circumstance (the 'Nest-INE' notation, Figure 31) improved upstream traversal; something similar could perhaps be devised to improve upstream traversal of circumstantial notations like the logic diagram, Figure 34. That could well be a feature provided by the editing environment rather than the notation proper.

We would also suggest that notation designers should pay more attention to notational reasoning, by providing mechanisms for perceptual parsing, where relevant, and for secondary notation.

Finally, since requirements change with time, we suggest that notation designers should pay more attention to problems of viscosity, especially in the graphical notations.