Lesson 4, Week 1: Programming is applied formal logic

This is the only theory lesson in the whole course. As such, it is very, very important!

AIM

— to understand the difference between logic and formal logic

— to understand how formal logic makes computer language possible

After this lesson, you will be able to

* explain what we mean by the word logic
* explain the difference between logic and formal logic
* explain why any computer must embody formal logic
* give a small example of a language that applies formal logic
* explain why writing computer programs is expressive, in the same way as writing poems or music

Why this lesson?

Programming often feels very strange to beginners, and if they haven't worked much with mathematical formulae, it can feel utterly alien.

This lesson cannot make the initial experience any less weird. Weird is part of the territory. But by explaining that it isn't arbitrary, nor aimed at keeping people out, but instead an essential aspect of any computer language, I hope to encourage you to persist until the Julia way of writing programs starts to feel natural. At that point, you will have learnt a new languge.

That is entirely serious: Julia, as I hope you will see below, is as rich and expressive in its own right as any natural language, such as Spanish, Mandarin and !Kung. Apart from implementing formal logic, there is no real difference with other languages. You could use it to write advertisements, love letters, novels, whatever! But mostly, since the formal logic can be implemented on a machine, you use it to write programs that make machines do things you want them to do.

Logic: the study of correct forms of reasoning

Reasoning is part of everyday life and language. It is simply the process of trying to make sense of things and expressing that as clearly as possible.

Unfortunately, there are many ways people use to make sense that actually are mistakes in reasoning. For example, it makes sense that raining wets grass, and this may lead people to reason as follows: if the grass is wet, there has been rain. Although this may often be true, it isn't always true: rain is not the only way grass can get wet. Strictly speaking, it is not correct. Reasoning correctly is quite hard!

As far we know, the first people who became aware of this *and* made a study of correct forms of reasoning lived approximately 100 generations ago. This occurred separately, in at least three places: in ancient China, ancient Greece and ancient India rules were formulated for reasoning correctly.

The word logic is used in many different ways in ordinary language. In this course, we use it in only one sense: the study of correct forms of reasoning.

Formal logic: using symbolic formulae to study and apply logic

Formal logic is simple to define: it uses symbols and formulae to study reasoning. This is fairly new discipline, it started only about 6 generations ago[1].

The aim of making logic formal was absolute precision in reasoning and absolute certainty about its correctness. It aims at this by using rules similar to the rules of algebra[2]. By doing so, it reduces logical analysis to calculations which can be independently checked. In other words, it removes the need for intuition and insight from logical analysis.

Formal logic has the strange property that something obvious and something baffling can sit side by side. For an example of the obvious, see the NOT, AND and OR truth tables below. Unfortunately, we you'll have to take our word for for the ease with which baffling things come up quite unexpectedly (or read *Logicomix*, in particular pages 164 to 168, about paradoxes).

Truth tables for `NOT`, `AND`, `OR`

In two-valued logic, which is by far the most common type of formal logic and also what Julia uses, there are two constant values, we denote them with `true` and `false` [3]. A statement is then something that is either `true` or `false`. As noted above, for a formal logic we need a symbol to stand for a statement, for example we can use P to stand for the statement "The earth spins on its axis" and Q for the statement "The earth is round"[4].

[1] See *Logicomix* by Doxiadis and Papadimitrou for an excellent and very human account of some of the most important episodes in the early history of formal logic.

[2] This goal has been considerably refined, because it was shown that a system cannot be consistent as well as complete and applicable to all of mathematics. For computers, it is not necessary to produce *all* correct reasoning—it is enough to ensure that all the reasoning in your computer is completely correct.

[3] In Julia code they are the values `true` and `false`.

[4] You may disagree with either of these; the first is much clearer than the second. Of course the earth is not perfectly round, but for practical purposes the imperfections are negligible, so it is perfectly reasonable to say that Q is true. This

So now we have two symbols P and Q. Here are three more: $\neg$, $\wedge$, $\vee$. With these, lets look at four valid formulae: $\neg P$, $\neg Q$, $P \wedge Q$, $P \vee Q$.

In words, these are "The earth does not spin on its axis", "The earth is not round", "The earth spins on its axis and the earth is round", and "The earth spins on its axis or the earth is round". That is, $\neg$ means `NOT`, $\wedge$ means `AND` and $\vee$ means `OR`.

But what do these mean, in terms of formal logic? Well, they (may) change the truth values. For example, if P is true then $\neg P$ must be false. A good way to summarise exactly what they mean is by means of truth tables. In a truth table, we specify the truth value of all valid combinations.

So the truth table for $\neg$ (equivalently, `NOT`) is as follows (note that *two* lines are needed to get both possibilities):

P	$\neg P$
`true`	`false`
`false`	`true`

The truth table for $\wedge$ (equivalently, `AND`) requires four lines:

P	Q	$P \wedge Q$
`true`	`true`	`true`
`true`	`false`	`false`
`false`	`true`	`false`
`false`	`false`	`false`

In other words, the combined forumula $P \wedge Q$ is `true` only when both are `true`, and `false` otherwise.

The truth table for $\vee$ (equivalently, `OR`) is:

P	Q	$P \vee Q$
`true`	`true`	`true`
`true`	`false`	`true`
`false`	`true`	`true`
`false`	`false`	`false`

In other words, the combined forumula $P \vee Q$ is `false` only when both are `false`, and `true` otherwise.

Because they may change truth values, `NOT`, `AND` and `OR` are called operators. To be more specific, they are logical operators.

The symbols we've seen are standard for research papers in formal logic, but not in progamming. In Julia, the symbol for `NOT` is `!`, `AND` is `&&` and `OR` is `||` . We'll see these again later; the fact that we need two characters make one symbol is of no significance.

Why formal logic needs a formal language

For formal logic to work, every expression must consist of symbols. In fact, it must consist of symbols that formal logic recognises—in exactly the same way that every expression in Julia must consist of symbols that Julia recognises[5].

sort of thing is why ordinary language and everyday reasoning are very hard to represent in a formal language.

[5]Introductions to formal logic have to solve a bootstrap problem: until you know the symbols, you can't do formal logic. They tend to start as we did here: by translating some very simple sentences into symbols. This has the effect of making the very simple seem complicated. Weird, but very hard to avoid.

A formal language makes it possible to say exactly what symbols and expressions are valid in that language. It has to go beyond the three operators we've seen above. That is, in order to express something, one has to use symbols that can carry truth values. Above, we used P and Q for that, but that was for convenience only. There are much better ways, and Julia is one of those.

Let's repeat that: Julia is a formal language[6]. It uses characters to form values and names, and then it uses delimiters and operators to form the names and values into expressions. The rules of the formal logic that governs Julia then determine whether the expression is valid or not.

In other words, you can write an unbelievably large number of Julia programs, but in order for them to consist of valid expressions, you have to follow the rules of Julia's formal language and formal logic.

Formal languages can generative and make computers possible

By "generative" I mean that one can make new expressions from old ones. A useful language with only a few valid expressions can certainly exist, but Julia and all major computer languages allow a great many valid expressions, and also set no limit on how many such expressions are used to write a program.

Writing a poem is also the result of a generative process: you add one word after another, one line after another, and there is no set limit on how long your poem may be.

But a formal language is not just generative: because the rules are completely fixed and explicit, they can be mechanised. They can be made to be part of a machine[7]; in such a machine you have a way to represent the values `true` and `false`. Then when you apply `not` to a `true` you get the value `false`, and so on. Such a machine can produce and evaluate all the expressions of a formal language and its the formal logic.

At the level of a microchip, all modern electronic digital computers contain many millions of tiny electrical circuits that are nothing but electronic versions of `NOT` , `AND`, `OR`[8]. Formal logic is what makes a computer possible; all modern computers are applications of formal logic.

A formal language with just a few letters and rules

Let us look at brief example of a formal language, let's call it AdHoc1. It will be very simple, just a few letters and rules. They allow the formation of some sentences in English, but also of expressions that are very far from being English. That is, the valid expressions in this language include some English but not all of it, and includes non-English also. To specify AdHoc1, we give a full list of symbols and a full list of how rules for building expressions.

The symbols are a few characters: the letters `a c e i f h l n s t` and the space character.

The rules are

[6] In fact, every computer language is a formal language.

[7] The earliest programmable computer design was by Charles Babbage over 150 years ago, for a machine to be driven by steam.

[8] Of course, they contain much more than just these operators, but this is not a course in computer design.

1. Some letters are also vowels, they are: `a e i`.
2. Some letters are also consonants, they are `c f h l n s t`.
3. An expression must start with a consonant.
4. An expression may not contain two vowels next to each other.
5. Any expression can have a character added to it, except that
   (a) No expression can have more than 100 characters
   (b) No expression can end with the space character

Note that some expressions made with AdHoc1 look like sentences in English, for example "that is it" and "that is it i think". But strangely enough, "i think that is it" is *not* a valid expression in AdHoc1, because it violates rule 3. Some amusing expressions are possible, such as "the think that i think that i think that i that think thinks". Also the famous paradox "this sentence is false" is valid AdHoc1.

However, by reading that last sentence as a paradox we are reading it as English. This is not part of AdHoc1, and requires an extension that applies AdHoc1 expressions to English. But most AdHoc1 expressions are not English, for example "ticillllll eee", so this extension is very informal[9] and relies a good deal on us as people who can read English.

There are some other patterns to note (or oddities, if you prefer): you cannot form the empty expression (because of rule 3), you can't write more than one line (there is no newline character), there are no delimiters so you cannot do punctuation. All one has so far with with AdHoc1 is a a set of rules for making

Let us also note that AdHoc1 cannot be a computer language, because it has no way of making the computer do anything. For that, there needs to be a number of expressions in the language that cause something to happen inside the computer. An example in Julia is the `println` function. In fact both lines of code in your first computer program makes something happen in or via the computer. Most of the details of this is hidden from the user of Julia—and this is one of the reasons for choosing Julia as a first programming language to learn.

Programming, poetry and truth

> 'Beauty is truth, truth beauty,—that is all
> Ye know on earth, and all ye need to know.' (Keats)

As we have repeatedly seen, a programming language like Julia allows you make many, many expressions. When you choose what names your variables are to have, which functions to employ, how to structure the logical operators in your program, and so on, you are using the language to express yourself. You are doing much the same as a poet.

Moreover, you cause things to happen: certain numbers are calculated, pictures are drawn, documents are printed. A poet aims at different effects: reactions from readers/listeners, memorable descriptions, catching the mood. But like a poet, the programmer is using their skill to achieve definite purposes.

[9] Actually, no formal language exists that produces all the valid sentences in English and only those sentences. This is partly because people cannot agree on what all the valid sentences in English are.

Let us not forget truth: it is central to logic, as we saw above. Computer programs utterly rely on the programmer keeping a very strict eye on whether their logic maintains truth. Paraphrasing Keats, one might say: "Code is truth, truth code,—that is all you need at the keyboard".

The kind of truth that is embodied in fine pottery is not same as the truth in beautiful poetry, and the truth that is captured in a computer program is far from both of these. But in the end, all three these forms use the expressive power of the medium to get at truth.

Programmers are, in that sense, like artists and poets[10].

Review and summary

* Logic (in this course) is the study of correct forms of reasoning
* Formal logic is the use of formulae to do logic
* Truth tables spell out exactly how logical operators work
* To be useful, a formal logic must be part of a formal language
* A formal language is a set of symbols and a set of rules for combining them
* Julia is itself a formal language
* A programming language is capable of endless variety of expression
* Writing code is a creative as much as writing poetry or music.

[10] But also like storytellers, stand-up comedians, composers and basket-weavers. All of these aim at certain effects which in some sense set a standard of truth.