6 Logic for programming

6.1 Propositional logic

Propositional logic

The aim of logic in computer science is to develop languages to model the situations we encounter as computer science professionals, in such a way that we can reason about them formally. Reasoning about situations means constructing arguments about them; we want to do this formally, so that the arguments are valid and can be defended rigorously, or executed on a machine. Consider the following argument:

Example 1.1
If the train arrives late and there are no taxis at the station, then John is late for his meeting. John is not late for his meeting. The train did arrive late. Therefore, there were taxis at the station.

Intuitively, the argument is valid, since if we put the first sentence and the third sentence together, they tell us that if there are no taxis, then John will be late. The second sentence tells us that he was not late, so it must be the case that there were taxis.

Much of this book will be concerned with arguments that have this structure, namely, that consist of a number of sentences followed by the word 'therefore' and then another sentence. The argument is valid if the sentence after the 'therefore' logically follows from the sentences before it. Exactly what we mean by 'follows from' is the subject of this chapter and the next one. Consider another example:

Example 1.2 If it is raining and Jane does not have her umbrella with her, then she will get wet. Jane is not wet. It is raining. Therefore, Jane has her umbrella with her.

This is also a valid argument. Closer examination reveals that it actually has the same structure as the argument of the previous example! All we have done is substituted some sentence fragments for others:

Propositions

Example 1.1	Example 1.2
the train is late	it is raining
there are taxis at the station	Jane has her umbrella with her
John is late for his meeting	Jane gets wet.

The argument in each example could be stated without talking about trains and rain, as follows:

If p **and not** q, **then** r. **Not** r. p. **Therefore**, q.

In developing logics, we are not concerned with what the sentences really mean, but only in their logical structure. Of course, when we apply such reasoning, as done above, such meaning will be of great interest.

6.2 Declarative sentences

In order to make arguments rigorous, we need to develop a language in which we can express sentences in such a way that brings out their logical structure. The language we begin with is the language of propositional logic. It is based on propositions, or declarative sentences which one can, in principle, argue as being true or false. Examples of declarative sentences are:

1. The sum of the numbers 3 and 5 equals 8.
2. Jane reacted violently to Jack's accusations.
3. Every even natural number >2 is the sum of two prime numbers.
4. All Martians like pepperoni on their pizza.
5. Albert Camus´etait un´ecrivain fran¸cais.
6. Die W¨ urde des Menschen ist unantastbar.

These sentences are all declarative, because they are in principle capable of being declared 'true', or 'false'. Sentence (1) can be tested by appealing to basic facts about arithmetic (and by tacitly assuming an Arabic, decimal representation of natural numbers). Sentence (2) is a bit more problematic. In order to give it a truth value, we need to know who Jane and Jack are and perhaps to have a reliable account from someone who witnessed the situation described. In principle, e.g., if we had been at the scene, we feel that we would have been able to detect Jane's violent reaction, provided that it indeed occurred in that way. Sentence (3), known as Goldbach's conjecture, seems straightforward on the face of it. Clearly, a fact about all even numbers >2 is either true or false. But to this day nobody knows whether sentence (3) expresses a truth or not. It is even not clear whether this could be shown by some finite means, even if it were true. However, in this text we will be content with sentences as soon as they can, in principle, attain some truth value regardless of whether this truth value reflects the actual state of affairs suggested by the sentence in question. Sentence (4) seems a bit silly, although we could say that if Martians exist and eat pizza, then all of them will either like pepperoni on it or not. (We have to introduce predicate logic in Chapter 2 to see that this sentence is also declarative if no Martians exist; it is then true.) Again, for the purposes of this text sentence (4) will do. Et alors, qu'est-ce qu'on pense des phrases (5) et (6)? Sentences (5) and (6) are fine if you happen to read French and German a bit. Thus, declarative statements can be made in any natural, or artificial, language.

NOT Declarative sentences

The kind of sentences we won't consider here are non-declarative ones, like

- Could you please pass me the salt?
- Ready, steady, go!
- May fortune come your way.

Primarily, we are interested in precise declarative sentences, or statements about the behaviour of computer systems, or programs. Not only do we want to specify such statements but we also want to check whether a given program, or system, fulfils a specification at hand. Thus, we need to develop a calculus of reasoning which allows us to draw conclusions from given assumptions, like initialised variables, which are reliable in the sense that they preserve truth: if all our assumptions are true, then our conclusion ought to be true as well. A much more difficult question is whether, given any true property of a computer program, we can find an argument in our calculus that has this property as its conclusion. The declarative sentence (3) above might illuminate the problematic aspect of such questions in the context of number theory. The logics we intend to design are symbolic in nature. We translate a certain sufficiently large subset of all English declarative sentences into strings of symbols. This gives us a compressed but still complete encoding of declarative sentences and allows us to concentrate on the mere mechanics of our argumentation. This is important since specifications of systems or software are sequences of such declarative sentences. It further opens up the possibility of automatic manipulation of such specifications, a job that computers just love to do[1].

Atomic sentences

Our strategy is to consider certain declarative sentences as being *atomic*, or *indecomposable*, like the sentence

'The number 5 is even.'

We assign certain distinct symbols p, q, r, . . ., or sometimes p1, p2, p3, . . . to each of these atomic sentences and we can then code up more complex sentences in a compositional way. For example, given the atomic sentences

p: 'I won the lottery last week.'

q: 'I purchased a lottery ticket.'

r: 'I won last week's sweepstakes.'

we can form more complex sentences according to the rules below:

[1]1 There is a certain, slightly bitter, circularity in such endeavours: in proving that a certain computer program P satisfies a given property, we might let some other computer program Q try to find a proof that P satisfies the property; but who guarantees us that Q satisfies the property of producing only correct proofs? We seem to run into an infinite regress.

$\neg$ The negation of p is denoted by $\neg$p and expresses 'I did not win the lottery last week,' or equivalently 'It is not true that I won the lottery last week.'

$\bigvee$ Given p and r we may wish to state that at least one of them is true: 'I won the lottery last week, or I won last week's sweepstakes;' we denote this declarative sentence by p $\bigvee$ r and call it the disjunction of p and r 2.

$\bigwedge$ Dually, the formula p $\bigwedge$ r denotes the rather fortunate conjunction of p and r: 'Last week I won the lottery and the sweepstakes.'

$\rightarrow$ Last, but definitely not least, the sentence 'If I won the lottery last week, then I purchased a lottery ticket.' expresses an implication between p and q, suggesting that q is a logical consequence of p. We write p $\rightarrow$ q for that3 . We call p the assumption of p $\rightarrow$ q and q its conclusion.

$\leftrightarrow$ 'If and only if it is Sunday, I'm happy' expresses an equivalence between p and q, suggesting that q is a logical consequence of p and also p is a logical consequence of q. We write p $\longleftrightarrow$ q for that.

Of course, we are entitled to use these rules of constructing propositions repeatedly. For example, we are now in a position to form the proposition p $\bigwedge$ q $\rightarrow$ $\neg$r $\bigvee$ q which means that 'if p and q then not r or q'. You might have noticed a potential ambiguity in this reading. One could have argued that this sentence has the structure 'p is the case and if q then . . . ' A computer would require the insertion of brackets, as in (p $\bigwedge$ q) $\rightarrow$ (($\neg$r) $\bigvee$ q) to disambiguate this assertion. However, we humans get annoyed by a proliferation of such brackets which is why we adopt certain conventions about the binding priorities of these symbols. Convention 1.3 $\neg$ binds more tightly than $\bigvee$ and $\bigwedge$, and the latter two bind more tightly than $\rightarrow$. Implication $\rightarrow$ is right-associative: expressions of the form p $\rightarrow$ q $\rightarrow$ r denote p $\rightarrow$ (q $\rightarrow$ r).

Differenza tra $\rightarrow$ e $\leftrightarrow$

$\longleftrightarrow$ 'If and only if it is Sunday, I'm happy' expresses an equivalence between p and q, suggesting that q is a logical consequence of p and also p is a logical consequence of q. We write p $\longleftrightarrow$ q for that.

Nota che alcune volte usare $\leftrightarrow$ è più completo che usare la semplice implicazione $\rightarrow$.

Ad esempio se ho un metodo isPositive(x) che restituisce vero se x e positivo e false se non lo è, allora la completa carratterizzazione del metodo è $x > 0 \leftrightarrow$ isPositive(x). Se scrivessi solo $x > 0 \rightarrow isPositive(x)$ allora con x negativo il metodo sarebbe libero di ritornare quello che vuole (true o false).

6.3 Natural deduction

How do we go about constructing a calculus for reasoning about propositions, so that we can establish the validity of Examples 1.1 and 1.2? Clearly, we would like to have a set of rules each of which allows us to draw a conclusion given a certain arrangement of premises. In natural deduction, we have such a collection of proof rules. They allow us to infer formulas from other formulas. By applying these rules in succession, we may infer a conclusion from a set of premises. Let's see how this works. Suppose we have a set of formulas φ1 , φ2 , φ3, . . . , φn , which we will call premises, and another formula, ψ, which we will call a conclusion. By applying proof rules to the premises, we hope to get some more formulas, and by applying more proof rules to those, to eventually obtain the conclusion. This intention we denote by

$$φ1, φ2, ..., φn⊢ψ$$

This expression is called a *sequent*; it is valid if a proof for it can be found. The sequent for Examples 1.1 and 1.2 is p ⋀ ¬q → r, ¬r, p q. Constructing such a proof is a creative exercise, a bit like programming. It is not necessarily obvious which rules to apply, and in what order, to obtain the desired conclusion. Additionally, our proof rules should be carefully chosen; otherwise, we might be able to 'prove' invalid patterns of argumentation. For example, we expect that we won't be able to show the sequent p, q p ⋀ ¬q. For example, if p stands for 'Gold is a metal.' and q for 'Silver is a metal,' then knowing these two facts should not allow us to infer that 'Gold is a metal whereas silver isn't.' Let's now look at our proof rules. We present about fifteen of them in total; we will go through them in turn and then summarise at the end of this section.

6.3.1 Rules for natural deduction

The rules for conjunction Our first rule is called the rule for conjunction (⋀): and-introduction. It allows us to conclude φ ⋀ ψ, given that we have already concluded φ and ψ separately. We write this rule as

$$\frac{φ \quad ψ}{φ⋀ψ}⋀i.$$

Above the line are the two premises of the rule. Below the line goes the conclusion. (It might not yet be the final conclusion of our argument; we might have to apply more rules to get there.) To the right of the line, we write the name of the rule; ⋀i is read 'and-introduction'. Notice that we have introduced a ⋀ (in the conclusion) where there was none before (in the premises). For each of the connectives, there is one or more rules to introduce it and one or more rules to eliminate it. The rules for and-elimination are these two:

$$\frac{φ⋀ψ}{φ}⋀e1. \qquad \frac{φ⋀ψ}{\psi}⋀e2.(1.1)$$

The rule ⋀e1 says: if you have a proof of φ ⋀ ψ, then by applying this rule you can get a proof of φ. The rule ⋀e2 says the same thing, but allows you to conclude ψ instead. Observe the dependencies of these rules: in the first rule of (1.1), the conclusion φ has to match the first conjunct of the premise, whereas the exact nature of the second conjunct ψ is irrelevant. In the second rule it is just the other way around: the conclusion ψ has to match the second conjunct ψ and φ can be any formula. It is important to engage in this kind of pattern matching before the application of proof rules.

Example 1.4 Let's use these rules to prove that p ⋀ q, r |− q ⋀ r is valid. We start by writing down the premises; then we leave a gap and write the conclusion:

p ⋀ q
r
...
q ⋀ r

The task of constructing the proof is to fill the gap between the premises and the conclusion by applying a suitable sequence of proof rules. In this case, we apply ⋀e2 to the first premise, giving us q. Then we apply ⋀i to this q and to the second premise, r, giving us q ⋀ r. That's it! We also usually number all the lines, and write in the justification for each line, producing this:

1	p ⋀ q	premise
2	r	premise
3	q	⋀e2 1
4	q ⋀ r	⋀i 3, 2

Demonstrate to yourself that you've understood this by trying to show on your own that (p ⋀ q) ⋀ r, s ⋀ t |− q ⋀ s is valid. Notice that the φ and ψ can be instantiated not just to atomic sentences, like p and q in the example we just gave, but also to compound sentences. Thus, from (p ⋀ q) ⋀ r we can deduce p ⋀ q by applying ⋀e1, instantiating φ to p ⋀ q and ψ to r. If we applied these proof rules literally, then the proof above would actually be a tree with root q ⋀ r and leaves p ⋀ q and r, like this:

$$\dfrac{\dfrac{p\bigwedge q}{q}\bigwedge e2 \qquad r}{q\bigwedge r}\bigwedge i$$

However, we flattened this tree into a linear presentation which necessitates the use of pointers as seen in lines 3 and 4 above. These pointers allow us to recreate the actual proof tree. Throughout this text, we will use the flattened version of presenting proofs. That way you have to concentrate only on finding a proof, not on how to fit a growing tree onto a sheet of paper. If a sequent is valid, there may be many different ways of proving it. So if you compare your solution to these exercises with those of others, they need not coincide. The important thing to realise, though, is that any putative proof can be checked for correctness by checking each individual line, starting at the top, for the valid application of its proof rule.

The rules of double negation Intuitively, there is no difference between a formula φ and its double negation ¬¬φ, which expresses no more and nothing less than φ itself. The sentence 'It is not true that it does not rain.' is just a more contrived way of saying 'It rains.' Conversely, knowing 'It rains,' we are free to state this fact in this more complicated manner if we wish. Thus, we obtain rules of elimination and introduction for double negation:

¬¬φ φ ¬¬e ¬¬i. φ ¬¬φ (There are rules for single negation on its own, too, which we will see later.)

Example 1.5 The proof of the sequent p, ¬¬(q ⋀ r) most of the proof rules discussed so far: ¬¬p ⋀ r below uses 1 2 3 4 5 6 p ¬¬(q ⋀ r)¬¬p q ⋀ r r ¬¬p ⋀ r premise premise ¬¬i 1 ¬¬e 2 ⋀e2 4 ⋀i 3, 5 Example 1.6 We now prove the sequent (p ⋀ q) ⋀ r, s ⋀ t |− q ⋀ s which you were invited to prove by yourself in the last section. Please compare the proof below with your solution: 1 2 3 4 5 6 (p ⋀ q) ⋀ r s ⋀ t p ⋀ q q s q ⋀ s premise premise ⋀e1 1 ⋀e2 3 ⋀e1 2 ⋀i 4, 5