Using ItL Jape with X

(*Third X edition, Jape version 3.2*)

Richard Bornat, Department of Computer Science, QMW

October 1998

Contents

0. Preface to X edition

This document was prepared in the first instance for the MacOS implementation of Jape. I should like to produce a version which explained exactly how to use Jape under X, but X is so variable – from choice of window manager to configuration of options – that it would be pointless to do so. Instead I have had to produce a simplified version of the MacOS document, without any mention of how windows are manipulated and managed.

The illustrations are from the MacOS version as well: I haven't yet found out how, in the X world, to make a picture which can be copied into a word-processed document. Apart from some minor differences in the way that the Konstanz font is displayed on the Mac and under X, they should be recognisable to any X user of Jape.

1. What is ItL Jape?

When I first came across Natural Deduction I found its rules seductively persuasive, but I found it hard to construct a proof. Partly that was because I was trying to do it in the tree style of Reeves and Clarke, trying to build a proof tree with assumptions at the top – some of them crossed out because they had bee 'discharged' – and a conclusion at the bottom. I found it hard to guess where and when and how many times to write down an assumption, when to cross it out and when not. It wasn't until I was introduced to a mechanical proof assistant that I began to be able to make proofs for myself, and not until I got used to the box presentation of proofs in Jape that I began to be totally relaxed about 'natural' deduction.

So I've been there, I know how it feels, and I would like to save you from the confusions that I suffered from. As part of the QMW Calculator Project that brought MiraCalc to the world and introduced the department to Tarski's World, I offer you *Jape*; the name is an acronym for 'just another proof editor'. Jape handles the details of the application of rules in a formal proof, leaving you to think about which rules to apply and to what and in what order. I hope that practice with ItL Jape will help you to understand the rules of Natural Deduction and to become so familiar with them that you can then make proofs on your own account without ItL Jape to help you.

When you are using ItL Jape I hope that you will quickly come to realise that Natural Deduction is mainly about the *simplification* of formulas by the application of rules. To find a Natural Deduction proof using ItL Jape you use rules to consume operators in the premises, assumptions premises and the conclusions of a proof in progress. You have to choose the assumption, premise or conclusion to work on, you have to choose the rule to apply, but ItL Jape does all the book-keeping, making sure that you have chosen a rule that applies to the things you have selected and showing you a nice tidy version of the proof as you build it.

Jape is a general proof editor, capable of handling many different logics. If you already know how to run Jape, you know how to run ItL Jape and, to a certain extent, vice-versa – but I expect that most readers of this document will be starting from scratch.

Forward or backward proof?

Just as I originally found it difficult to construct a Natural Deduction proof as a tree in the Reeves and Clarke style, so I also found it difficult to decide, in a box-style proof, how and when to 'introduce an assumption'. I learnt, by using Jape and other mechanical proof aids, that it is *hardly ever necessary* to do such a thing. In practice, as I hope you will discover, assumptions are introduced automatically and naturally, as a consequence of the use of rules on formulæ that are already present in the proof. You don't have to introduce arbitrary assumptions ever: all that is necessary is that sometimes you are prepared to work backwards – from conclusion towards assumption – rather than always forwards. The gain is that when you introduce an assumption in this way you *never*have to guess what it might be.

It would be as wrong to say that backward proof is better than forward as it would be to say the opposite. One of the things that I hope you will learn from ItL Jape is that you can't choose in advance whether to make a proof by working forwards from the assumptions or backwards from the conclusions. The problem – the collection of premises, assumptions and conclusion and the theorems and rules you have available – should dictate what you must do, not some orthodoxy.

Forward proof works well when you have a step which you can make entirely from the premises and assumptions which you already have available. It is attractive, to the novice, because it makes it seem that the proof is being developed line-by-line in the same order as the completed proof can be read.

Forward proof stops working well, and backward proof starts working better, when you run out of things to do with the premises. When this happens, rigid-thinking forward provers will want to introduce an assumption to relieve their difficulties, and will find that Jape doesn't let them. That is because a proof is really a *structure* of lines, not a sequence. Jape shows you that assumptions are introduced as the result of using rules – in Natural Deduction these are the ∨-E and ∃-E rules going forward, and the →-I and ¬-I rules going backward. When you use one of those rules you are, in effect, saying "I don't know what the final proof will be, but I know what its *structure* will be". Jape helps you, in those cases, by calculating the assumption that you need.

You will also find that when making a proof with ItL Jape, sometimes the first move you make must be a step of backward reasoning. This isn't 'wrong' or in any sense cheating: it is the way that proofs actually get developed. In reality it is the line-by-line 'forward' reading of a completed proof that is deceitful, because it hides from the novice the fact that a proof is a structure, not a sequence, of steps, and that its construction sometimes needed forward and sometimes backward reasoning.

There is no best way nor even a 'right way' to develop a proof; nor is there a 'wrong way' – although of course we may decide, once a proof is completed, that it might have been done better, and then ItL Jape will let you try again. Forward reasoning isn't better, or righter – or wronger – than backward reasoning: each has its place, and in order to make *natural* Natural Deduction proofs it is often necessary to employ both styles. ItL Jape was developed in order to help you learn by experience that most Natural Deduction proofs are so natural as to be almost automatic, with a structure dictated by the structure of the premises, assumptions and conclusion. In order to learn that lesson it is sometimes necessary to make use of backward reasoning. I hope that you will find ItL Jape easy to use, and I hope that it will stimulate you to think about the structures of proofs, about why the rules are the way they are, and about how Natural Deduction works.

Premise, assumption, hypothesis: what's in a word?

In the QMW course, your lecturer writes 'premise' or 'premises' next to any formulæ which are assumed at the outermost level of a proof, and 'assumption' or 'assumptions' next to any which are assumed inside the proof. ItL Jape does the same.

The distinction between 'premise' and 'assumption' is to my mind very fine, a nice distinction indeed. In this manual I blur it in order to make my sentences shorter. I've used the word assumption where I would otherwise have to write "premise or assumption".

The word 'hypothesis' is, to my ear, synonymous with 'assumption'.

Differences from the course notes

Jape isn't really a Natural Deduction engine, and its internal mechanism is based on inference trees, not the sort of line-and-box proofs that ItL Jape displays. Most of the time ItL Jape does a really good job of pretending that it is a Natural Deduction engine and that it really does work with lines and boxes, but sometimes the internal workings show through. You may notice the following:

- Jape doesn't always allow you to appeal to an earlier line of the proof, even when the box structure would let you;
- Jape's implementation of 'scope boxes' associated with the ∃-E and ∀-I rules uses pseudo-predicates 'var *x*' and 'inscope *x*', rather than annotated boxes;
- Jape's syntax for quantification is ∀*variable*.*formula* and ∃*variable*.*formula* – in each case the full stop (dot, period) is essential, and *formula* has to be bracketed unless it's just a predicate application.

So far as I know, those are the only significant differences between Jape's treatment and the course notes.

Any comments?

If you have any comments about ItL Jape that you would like me or anybody else to hear, there's a dcs.jape newsgroup at QMW, or you can email me (richard@dcs.qmw.ac.uk). I'm especially interested to hear from people who find any difficulty, no matter how slight, in using the program to do their work: Jape will only improve if I get to hear of their problems. So if you see something wrong, tell me, put a message in the newsgroup or email me. If you think Jape is wonderful, please let me know!

2. Getting started

Type the following to an xterm window:

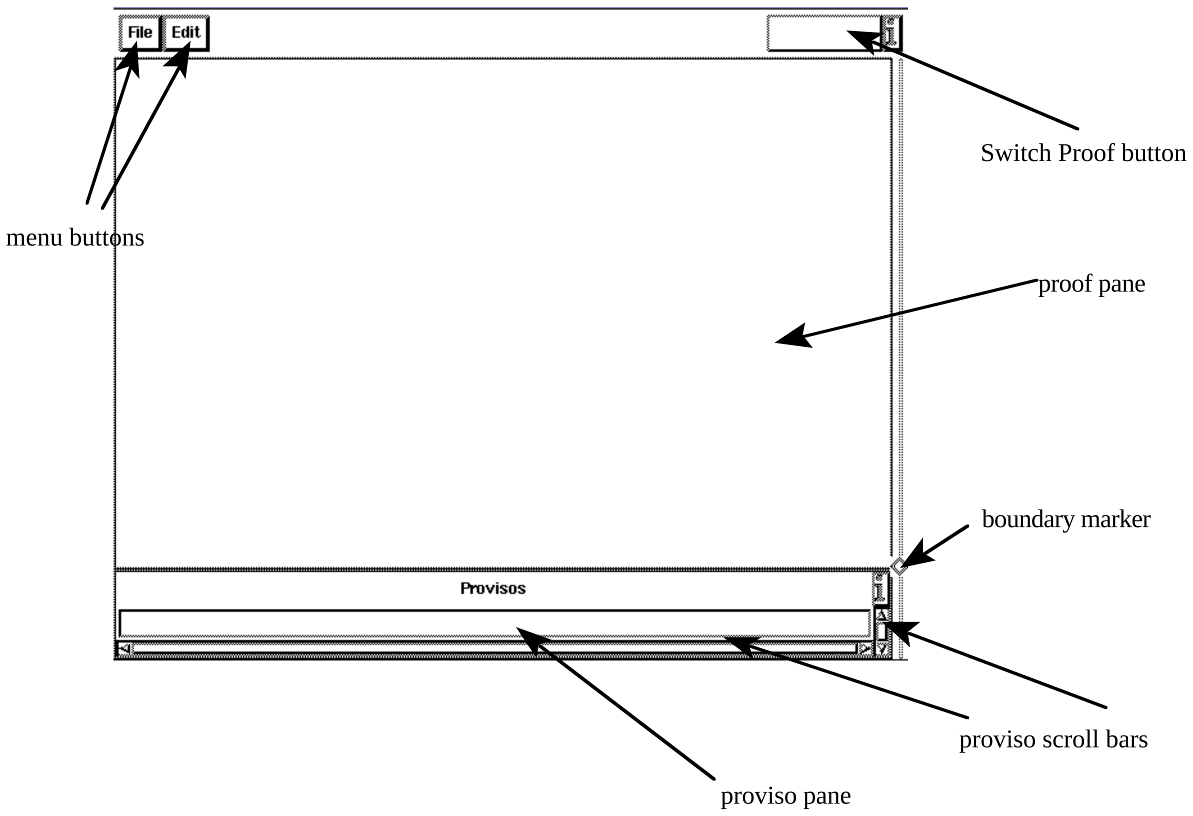
```
itljape&
```

(if that doesn't work try `/import/jape/itljape&`). Jape then starts and creates two windows:

- Jape conjectures;
- Jape session for theory "ItL".

(you may also see a dialogue box quacking on about fonts: just click on the button marked with a red tick, and click on the red tick in the box which follows). You may have to use your window manager to arrange the windows conveniently on the desktop.

The session window looks something like this:



Most of the work goes on in the proof pane. The proviso pane shows provisos (sometimes called side-conditions on a proof) which can arise as a result of using rules which deal with ∀ and ∃ formulæ: most of the time it's blank. You can alter the boundary between the panes by dragging the boundary marker up and down. You scroll around in the proviso pane with the proviso scroll bars; you can drag a proof in the proof pane by pressing on a blank part of the pane and dragging the proof to the position you want.

If you have some saved proofs, now is the time to load them (use 'Load a file of proofs' from the File menu).

If you are running Jape for the first time, read on.

Buttons in the Conjectures panel

The Conjectures panel has various parts (which I'd love to illustrate with a screen snapshot, but I don't know how). Most important is the list of conjectures – a collection of proof problems which we think you might like to experiment with. The list is scrollable, using the scroll bar to its right. If you want to see more, or less, of the list you can alter the size of the window using whatever gesture or gestures your window manager provides.

The Conjectures window is described in this document as a *panel*, because it carries buttons and a list of things to choose from. The buttons in the panel each have their particular use, discussed below. For the moment we shall concentrate on the Prove button, which is used to start a proof. Select the first entry in the panel (click on it) and then press the Prove button. A display describing the conjecture will appear in the proof pane.

Doing things in the session window

You make your proofs in the proof pane: any provisos which might arise during the proof are displayed in the proviso pane. You can move a proof around in its pane by moving the cursor to a blank part of the background, holding down the middle button of the mouse[1] and dragging the proof around. For some reason X Jape always puts the proof too low down in the pane, and the first thing I do is always to lift it a little. The motion is jerky and the proof moves faster than the mouse does, but with practice you can position it where you like. If you lose the proof by dragging too fast or too far, press the Home key on the keyboard, and it should pop up somewhere visible.

You can have more than one proof on the go at the same time. Only one of them will be *active*, and shown in the session window, but each of them will be listed in the menu under Switch Proof (top right of the session window). The active one has a highlighted check-box in that menu. By clicking on the entry of an inactive proof in the Switch Proof menu, you make it active and it takes over the session window. You can abandon the active proof at any time by clicking Abandon Proof in the Switch Proof menu.

Selecting and choosing

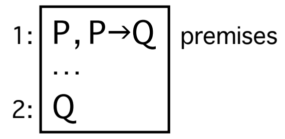
X Jape has various different ways of ‘gesturing’ at a proof with the mouse, introduced in the discussion below (see appendix A for a complete list). When this manual uses one of the words ‘choose’, ‘select’ or ‘click’, you are supposed to click something with the left mouse button unless the context makes it clear that something else is intended.

1 If your mouse doesn’t have a middle button you are perhaps running on an Apple Mac, and using Jape via an X server program. Don’t bother: there’s a MacOS implementation which you can get from the Web.

3. Two proofs by forward reasoning using →- elimination

If you haven't already started the proof of the first conjecture in the Conjectures panel, do it now: select the first entry in the list of conjectures and press the Prove button. Jape will display the problem to you in the proof pane, as illustrated above. (If you already have a proof of that conjecture on the go, Jape will tell you and you can then choose whether to start another. If you have already proved and registered the proof of that conjecture, Jape will tell you and you can decide whether you want to attempt a new proof.)

If I showed a picture of a session window every time I wanted to include an example, this manual would be far too large. From now on I shall save space by showing only the contents of the proof pane – and the proviso pane, if it isn't empty – like this:



```
1: P, P→Q   premises
   ...
2: Q
```

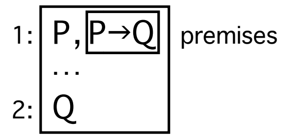
You are asked to prove from premises P and P→Q the conclusion Q. The premises are listed on a single line, separated by a comma. Jape uses one line rather than two so as to save screen space, which is a precious resource, as you will see if ever the proof becomes so long that it exceeds the height of the proof pane[1]. The line of dots between lines 1 and 2 indicates that there is something to prove – a 'gap' in the proof so far. The lines of the proof are already numbered, but those numbers can be expected to change as extra lines are added to the proof.

Making a proof step

You should already have an idea how to solve this problem: you use the →-E rule to derive Q from premises P and P→Q (if you don't know, you are about to learn, and if you've forgotten about the →E rule, see the appendix on the → rules). In Jape's language you simplify the P→Q premise using the →-E rule: you don't have to tell it about the P premise because it will find it automatically.

The rules of Natural Deduction are all in the Rules menu. As well as the rules that you already know, there's an extra entry 'hyp' at the bottom of the menu, whose use is discussed later.

To make the proof step you must 'select' the P→Q premise and then 'apply' the →-E rule. You select the premise by clicking once on it with the left mouse button (move the mouse pointer over the premise, press the left mouse button and immediately release[2]). A box will appear round the premise you have selected, like this[3]



$$\begin{array}{rll} 1: & P, \boxed{P\rightarrow Q} & \text{premises} \\ & \dots & \\ 2: & Q & \end{array}$$

If you have selected the wrong premise, don't panic: left-click on the one you really wanted to choose and Jape will obey. If you clicked on the conclusion by accident, just left-click on some blank part of the proof pane and Jape will cancel all your selections; then you can start again.

1 You can reduce the size of a proof somewhat by setting the font size, using the Set Font sizes command from the Files menu. X fonts are always a bit unreadable at small sizes, so it's a good idea to experiment.

2 Don't double-click, or you will get an error message about 'direct manipulation'. You can ignore that error message, but you will have to select the assumption again once it has gone away.

3 From now I shall only show the contents of the proof window (and the proviso window, if there is anything in it). You can imagine the rest of the screen.

Now pull down the 'Rules' menu and click →-E. Jape makes the step, and shows a justification for it. In this case the result of the step is the conclusion we are trying to prove, so the proof is completed and the line of dots disappears:

1:	P, P→Q	premises
2:	Q	→-E 1.1,1.2

I hope you did get this effect: if not, use Undo (in the Edit menu, or use the keyboard Undo key) and try again.

Notice the way that ItL Jape numbers justifications which refer to premises or assumptions: the step which leads to line 2 depends on the first formula on line 1 (1.1 means P) and the second formula on line 1 (1.2 means P→Q).

That's not the only way to make this proof, but it will do for now.

Registering your proof with Jape

Now that you have made a proof of a conjecture you can save it : pull down the Edit menu and select Done. The proof disappears from the session window, and Jape records the fact that the conjecture is now proved, marking its entry in the conjectures panel with 'THM' in the margin.

Proof of the conjecture has made it a theorem. You can use theorems in your proofs as if they were additional Natural Deduction rules by using the Conjectures panel's Apply button, and you can review their proofs using Show Proof. But more of that later.

Three different ways to make the same proof

There are often several equally attractive ways to make the same proof – and just as often there are other less desirable ways as well.

Select the second entry in the Conjectures panel and click Prove. You should see a proof pane containing

1:	P→Q, Q→R, P	premises
	...	
2:	R	

I shall show you three different ways to make the same proof (and there are still other ways in which it can be done). The first starts by forward proof from P and P→Q, which are both premises, to make the intermediate conclusion Q. Left-click to select P→Q and apply →-E from the Rules menu. The proof becomes

1:	P→Q, Q→R, P	premises
2:	Q	→-E 1.3,1.1
	...	
3:	R	

Notice how the line numbering has changed: the conclusion is now line 3, and line 2 is the result of the →-E step. Notice also how the justification of line 2 identifies the premises it has used: P (1.3) and P→Q (1.1). Now on line 2 we have Q and on line 1 we have Q→R: clearly we can derive R using the →-E rule.

Select Q→R and apply →-E once again, and the proof is completed:

1:	P→Q, Q→R, P	premises
2:	Q	→-E 1.3,1.1
3:	R	→-E 2,1.2

That’s one way of doing it. What are the others? To find out, apply Undo (from the Edit menu) twice, to get back to the starting point. This time select Q→R and apply →-E. You will see

1:	P→Q, Q→R, P	premises
	…	
2:	Q	
3:	R	→-E 2,1.2

This state of affairs has been reached by logical reasoning. If you want to break down Q→R by →-E, you need a proof of Q and it leads to a proof of R; R was the conclusion you were working towards, so you have already found out to reach R, but now you have a new problem: how do you reach Q, which is on the way to R?.

In effect you know the *structure* of the proof, but you don’t yet know all the steps nor their order. So ItL Jape shows you that the conclusion on line 3 has been derived from line 2 (Q) and premise 1.2 (Q→R). Since there isn’t a premise Q you will have to provide a proof of it, and therefore line 2 is a new conclusion to be proved. (This isn’t backward reasoning: it’s incomplete forward reasoning from the premises, even though the reasoning reaches the conclusion. I show some examples of true backward reasoning backwards below, using a different example.)

Now we’ve got an premise P→Q and an premise P, so we can proceed as in the first proof: select P→Q and apply →-E, and the proof is once more completed:

1:	P→Q, Q→R, P	premises
2:	Q	→-E 1.3,1.1
3:	R	→-E 2,1.2

The same proof, a different way of deriving it with ItL Jape. That’s one of Jape’s strengths: it doesn’t force you to apply rules in just one order. At least not always!

I have shown two ways of deriving the same proof. What’s the third? Undo twice to get back to

1:	P→Q, Q→R, P	assumptions
	…	
2:	R	

You have already proved a theorem P→Q, P ⊢ Q. That theorem applies in this situation: we do have an premise P and an premise P→Q, and, just as when you apply a rule, it doesn’t matter what order they appear in, how many times they appear, or what other assumptions we have.

We want to apply the theorem to the premises P and P→Q. Select either of them (it doesn’t matter which); then select the theorem (it’s the first entry in the Conjectures panel) and press Apply[1]. The proof pane changes to show the effect of applying the theorem:

1:	P→Q, Q→R, P	premises
2:	Q	Theorem P, P→Q ⊢ Q 1.3,1.1
	…	
3:	R	

[1] If you press Prove by mistake, you will be offered the opportunity to start a new proof of P→Q, P ⊢ Q: just click No in the dialogue box and then press Apply. If you press Show by mistake the proof you have already made of P→Q, P ⊢ Q will be displayed in the session window: just choose Abandon Proof from the Switch Proof menu, and then press Apply.

Now the proof can be completed as before, using the →-E rule, or it can be completed using the same theorem again. We now have Q (line 2) and Q→R (line 1, premise 2), and we want to prove R. The theorem covers that possibility. Apply it once more[1] and the proof is completed once again:

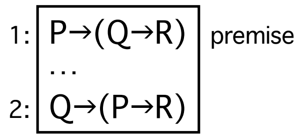
1:	P→Q, Q→R, P	premises
2:	Q	Theorem P, P→Q ⊢ Q 1.3,1.1
3:	R	Theorem P, P→Q ⊢ Q 2,1.2

Choose Done (Edit menu) to register the proof with Jape.

1 You don't need to select a hypothesis, because Jape can work out which hypotheses are to be used from the theorem itself and the conclusion Q.

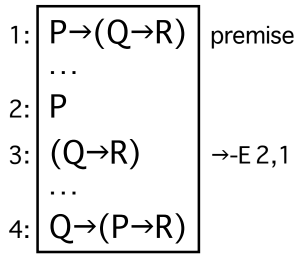
4. A proof by backward reasoning using →-introduction

Choose P→(Q→R) ⊢ Q→(P→R) from the Conjectures panel (it's the fifth entry). You should see a proof pane containing



```
1: P→(Q→R)   premise
   ...
2: Q→(P→R)
```

This time you can't use forward reasoning to simplify the premise,. If you think about it, you can see why: to use the premise P→(Q→R) you need a proof of P, but there isn't one available. If you do select the premise and apply →-E as in the first two proofs, ItL Jape will show you that you need a proof of P to go any further:

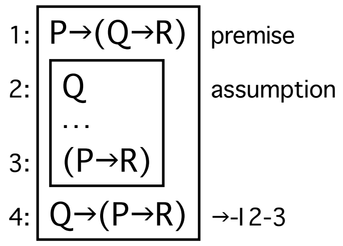


```
1: P→(Q→R)   premise
   ...
2: P
3: (Q→R)     →-E 2,1
   ...
4: Q→(P→R)
```

Line 2, with the three dots above it, is an invitation to prove P. You can't make a proof of P out of thin air, even with the help of the premise P→(Q→R), and not even with the help of the theorems you have already proved. Undo back to the beginning.

It's at this point that a rigid-thinking forward reasoner would begin to bleat about the need to introduce additional assumptions. ItL Jape can help, but it does so very straightforwardly, by *backward* reasoning from the conclusion.

Observe that the → in the conclusion will almost certainly have to be introduced by the →I rule. Apply the →I rule from the Rules menu (you can select the conclusion before applying the rule, but because there is only one unproved conclusion in the proof you don't have to select it) and you should see

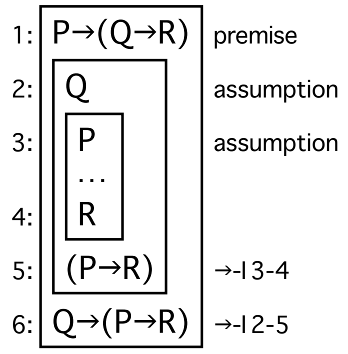


```
1: P→(Q→R)   premise
2: Q         assumption
   ...
3: (P→R)
4: Q→(P→R)   →-I 2-3
```

What ItL Jape is done is to proceed *as if* the last line was produced by using the →I rule: that would have to be because there is a proof of P→R from Q, and ItL Jape has drawn the corresponding box structure with a gap which shows that the proof of P→R isn't completed. Once again, you can see the *structure* of the proof, even if it doesn't have all its steps. Notice that you didn't explicitly have to decide to 'introduce an assumption', because the formula Q is part of the conclusion Q→(P→R), and it is just exactly the assumption you need to prove that conclusion – no other formula would fit. You don't have to decide what conclusion to put in the box – that, too, can be calculated from the rule and the conclusion you applied it to. This is an example of the way that assumptions are *naturally* introduced by backward reasoning in Jape, and I believe that this kind of backward reasoning makes 'Natural' Deduction far more natural. (This kind of helpfulness with the introduction of assumptions isn't restricted to backward reasoning steps, as some of the examples in the appendices on the ∨ and ∃ rules show).

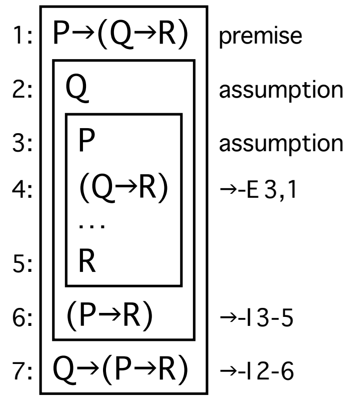
Notice that the last line in the proof is now marked with a justification and doesn't have three dots above it any more – that is, it is marked as 'proved', although we don't yet have the complete proof of the lines on which it depends. The justification for that line quotes the box on lines 2-3. That quotation will expand, of course, as the proof develops and the box expands.

Now just because we know Q we don't necessarily know P, so we still can't use →E on line 1. But we can see that line 3 is almost certainly produced by the →I rule. If you apply the →I rule again, you will see:



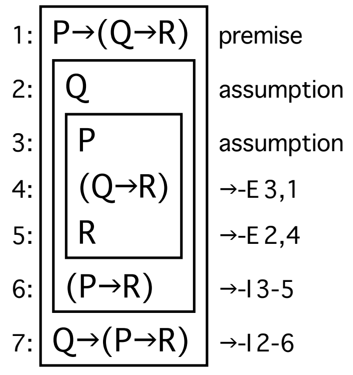
```
1: P→(Q→R)   premise
2: Q         assumption
3: P         assumption
   ...
4: R
5: (P→R)     →-I 3-4
6: Q→(P→R)   →-I 2-5
```

We have to prove R from the three assumptions P→(Q→R), Q and P: that's pretty easy using forward reasoning and the →-E rule. Choose the premise on line 1 and apply →-E. You will see



```
1: P→(Q→R)   premise
2: Q         assumption
3: P         assumption
4: (Q→R)     →-E 3,1
   ...
5: R
6: (P→R)     →-I 3-5
7: Q→(P→R)   →-I 2-6
```

Now choose line 4 and apply →-E again:



```
1: P→(Q→R)   premise
2: Q         assumption
3: P         assumption
4: (Q→R)     →-E 3,1
5: R         →-E 2,4
6: (P→R)     →-I 3-5
7: Q→(P→R)   →-I 2-6
```

The proof is complete.

Notice that by using backward reasoning Jape has been able to *calculate* what assumptions to introduce. A forward reasoner, looking at line 1, might suppose that we need to introduce an assumption P to help simplify P→(Q→R): but that is wrong: the proof doesn't go through in that way. The assumption we need is dictated by the shape of the *conclusion*, not the premise.

This example has also illustrated the general principle that *backward* reasoning with an introduction (-I) rule essentially eliminates connectives from conclusions, just as earlier examples showed that *forward* reasoning with an elimination (-E) rule eliminates connectives from premises and assumptions. In fact the forward/backward process, and the way that it applies to logical connectives, is so nearly mechanical that you can set Jape up so that it chooses the rule itself when you double-click on a conclusion, a premise or an assumption. Because we want you to learn about Natural Deduction and not just the use of the mouse, we've been heartless and ItL Jape is set up so that you have to choose the rules for yourself.

5. Unknowns, unification and the *hyp* rule

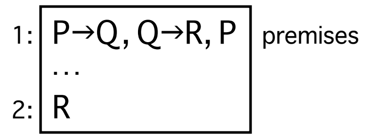
Above I showed three different ways to prove P→Q, Q→R, P ⊢ R. There is a fourth, which illustrates one of the ways that Jape does its work behind the scenes.

The →-E rule is

$$\begin{array}{lll} i: & A & \ldots \\ & \ldots & \\ j: & A \rightarrow B & \ldots \\ & \ldots & \\ k: & B & \rightarrow -E\ i,j \end{array} \qquad \text{or} \qquad \dfrac{\begin{matrix}\vdots & \vdots \\ A & A \rightarrow B\end{matrix}}{B}\rightarrow -E$$

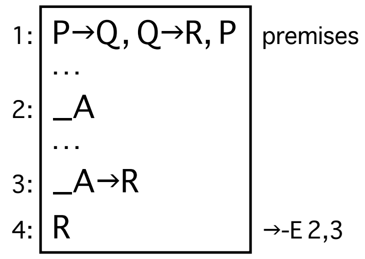
– given a proof of *A* and a proof of *A*→*B*, make a proof of *B*. If you point to an assumption formula which matches *A*→*B*, Jape can make the step and then, if you have a conclusion *B* or an assumption *A*, it can fit the new step into the proof. If you point to a conclusion but not an assumption, Jape can match the conclusion with *B*, but that doesn't tell it what to use in place of *A*. In those circumstances, you have to decide. But it would be oppressive to force you to decide before you are ready, so Jape makes up an 'unknown' name, which stands for some yet-to-be-decided formula *A*, and that allows you to defer your decision .

To show you an example of what happens I will go through the proof of P→Q, Q→R, P ⊢ R again. You may have proved this already, but Jape lets you make a new proof if you want to: select its entry in the Conjectures panel, press Prove[1], and you will see



```
1: P→Q, Q→R, P    premises
   ...
2: R
```

Apply the →-E rule *without selecting a premise* (you can select the conclusion on line 2 if you wish). You will see[2]



```
1: P→Q, Q→R, P    premises
   ...
2: _A
   ...
3: _A→R
4: R              →-E 2,3
```

What's happened is *backward* reasoning: I've guessed that the last step in the completed proof will be an application of →-E, and Jape has done its best to oblige. Because it doesn't know all of the information it needs, it has introduced an unknown _A[3] to stand for the bit that's missing: you have to find a proof of _A and a proof of _A→R, and obviously that requires you to decide just what _A is.

You may find this display surprising, perhaps even confusing, and then you might ask: why is it necessary? The answer is that the display tells you not only that the rule is applicable, but also that more information is needed to make it clear just how it applies. Another way to look at it is that you must decide, whenever you apply the →-E rule, just what formula to use in place of *A*, but Jape has allowed you to defer that decision by introducing an unknown, a placeholder, _A. It can be very useful to defer such decisions when exploring for possible proofs or trying out the effect of rules that you may not quite understand.

1 If this conjecture is already a theorem, Jape will put up a dialogue box asking you if you want to prove it again. Yes, you do.

2 On the X Jape implementation at the time of writing, the underscores on lines 2 and 3 are rather difficult to see. This is regrettable.

3 All unknowns, and only unknowns, start with an underscore.

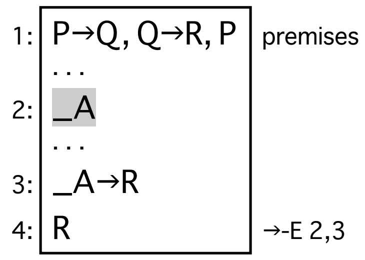
Now it is immediately obvious in this example that the unknown _A must stand for Q, so we must make it do so. When we have done this, Jape will replace every occurrence of _A, which we don't want to see, with Q, which is quite acceptable. There are two basic ways to show Jape the correspondence.

Unification with the Unify command

Up to now we've selected whole formulæ by clicking on them. Now we want to tell Jape that _A, which appears as a whole formula on line 2, and Q, which only appears as a sub-formula on line 1, are the same. What you have to do is to 'text-select' a number of sub-formulæ and then use the Unify command from the Edit menu to say that all those sub-formulæ are to be 'unified' – that is, to be made the same.

Text-selection uses the middle button with a press-and-drag action. You move the mouse cursor to one end of the text you want to select, hold the middle button down, and wipe the cursor across the text you are selecting.

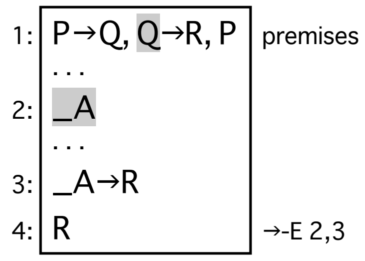
Move the mouse to the beginning or the end of line 2; hold the middle button down and press-and-drag with the mouse over _A: you should be able to make it light up like this[1]



```
1: P→Q, Q→R, P   premises
   . . .
2: _A
   . . .
3: _A→R
4: R             →-E 2,3
```

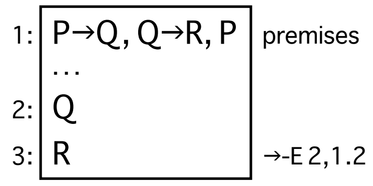
It's quite a simple action, but it takes a bit of practice. If you get it wrong you can double-middle-click the bad selection with the middle button and start again. If you double-middle-click on a blank part of the screen you cancel all the text selections, wherever they are in the proof pane.

Once you've successfully selected _A you can move on to Q. Just middle-clicking over one of the Qs in line 1 does the job, and the proof should now look something like this:



```
1: P→Q, Q→R, P   premises
   . . .
2: _A
   . . .
3: _A→R
4: R             →-E 2,3
```

Now you can use the Unify command from the Edit menu to make the two sub-formulæ the same, and the proof will change to



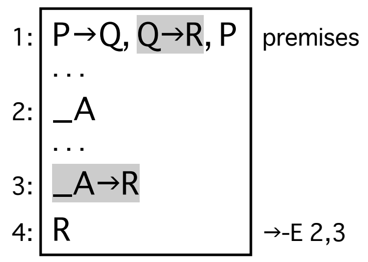
```
1: P→Q, Q→R, P   premises
   . . .
2: Q
3: R             →-E 2,1.2
```

Rather a lot happens in one step! Jape is told that _A is the same as Q, so line 2 changes from _A to Q, as you might expect. Line 3 used to be _A→R, but Jape first made it Q→R (because _A is the same as Q), and there is already an assumption Q→R: so references to that line can be re-directed to premise 1.2 instead; then we don't need the conclusion Q→R at all, and the last line can be numbered 3.

1 Text selection is indicated on different machines in different ways. On a colour screen it is usually shown by changing the colour of the background. On a grey-scale screen, and in these notes, it can be shown by a grey background. On a black-and-white screen it is shown by a black background with white text.

No matter that quite a lot happens in one step, and some of it is rather subtle – the total effect is just what is needed, and we are back to something recognisable. What happened was that an unknown was introduced by backward reasoning, and then eliminated by unification. It was all quite unnecessary in this case, because forward reasoning from the assumption can produce the same effect in one step and without any need for text selection, but it was worth the illustration because unification of an unknown with a known formula is a very useful technique – as you will find when it comes to proofs involving negation, if not before.

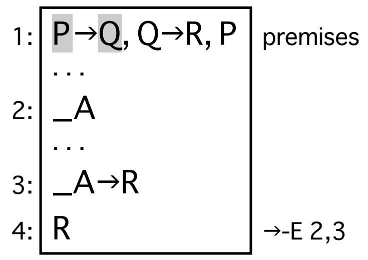
Now in the step above I made _A the same as Q, and that made _A→R the same as Q→R. If I had used text-selection to choose those two sub-formulæ instead of _A and Q:



```
1: P→Q, Q→R, P    premises
   ...
2: _A
   ...
3: _A→R
4: R              →-E 2,3
```

then the effect of the Unify command would have been just the same. In unifying _A→R with Q→R, Jape notes that the → and the R parts are the same, and unifies only _A with Q.

Unification can change only unknowns. If you text-select P and Q, say:

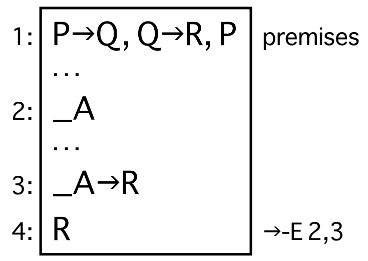


```
1: P→Q, Q→R, P    premises
   ...
2: _A
   ...
3: _A→R
4: R              →-E 2,3
```

then the Unify command will fail – Jape can't unify P with Q, and it will say so.

Unification with the hyp *rule*

There's another way – sometimes it's a simpler way – to force Jape to unify _A with Q. In the display with unknowns



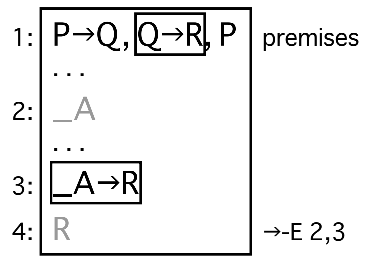
```
1: P→Q, Q→R, P    premises
   ...
2: _A
   ...
3: _A→R
4: R              →-E 2,3
```

the formula _A→R on line 3 is the same as the premise Q→R on line 1. The hyp rule, which means nothing more nor less than 'an assumption proves this conclusion', is the extra entry at the bottom of the Rules menu. In tree form the rule is

$$\frac{\quad}{A}\,hyp$$

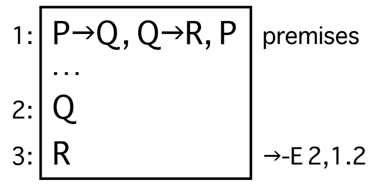
and it has no equivalent in box form, because it is equivalent to pointing to a particular assumption line.

Before you use the hyp rule, you select the relevant assumption and conclusion – sometimes you can use the rule without selection if it's obvious what you mean, but selection is essential if there is more than one conclusion and/or more than one assumption which would fit the conclusion. In this case it would be enough to select the conclusion on line 3 (because there is only one assumption which fits it), but it’s good mental hygiene to select the matching second premise on line 1. When you’ve done that, the proof pane should show:



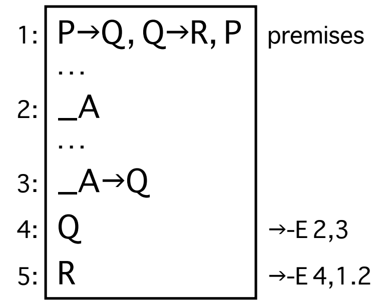
```
1: P→Q, Q→R, P   premises
   ...
2: _A
   ...
3: _A→R
4: R             →-E 2,3
```

(Notice how the parts of the proof that aren’t relevant to proving line 3 are ‘greyed out[1]’ by the act of selecting that conclusion). Now you can apply the hyp rule, and just as with the Unify command, all the _As are replaced in a single step, and the _A→R line vanishes entirely:



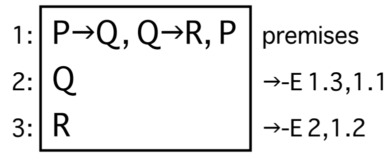
```
1: P→Q, Q→R, P   premises
   ...
2: Q
3: R             →-E 2,1.2
```

You can do the same thing again in the next step: apply →E to line 2



```
1: P→Q, Q→R, P   premises
   ...
2: _A
   ...
3: _A→Q
4: Q             →-E 2,3
5: R             →-E 4,1.2
```

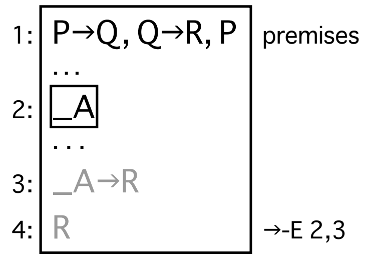
Now whether you say that _A is the same as P or that _A→Q is the same as P→Q, and whether you use Unify or hyp to say it, the proof is completed just as before.



```
1: P→Q, Q→R, P   premises
2: Q             →-E 1.3,1.1
3: R             →-E 2,1.2
```

More unknowns than you can shake a stick at

To see a slightly more complicated use of unknowns, go back a couple of steps to the first introduction of _A and choose the conclusion on line 2:



```
1: P→Q, Q→R, P   premises
   ...
2: _A
   ...
3: _A→R
4: R             →-E 2,3
```

1 In some X Jape versions ‘greying out’ is implemented by a colour change. In this manual I can only use shades of grey and black.

Now apply →-E:

```
1: │ P→Q, Q→R, P  │ premises
   │ ...          │
2: │ _A1          │
   │ ...          │
3: │ _A1→_A       │
4: │ _A           │ →-E 2,3
   │ ...          │
5: │ _A→R         │
6: │ R            │ →-E 4,5
```

If _A is to be proved by →-E, which it eventually will be, then yet another unknown is needed. Jape generates unknown names very simplistically, and comes up with _A1. To make the proof go through, _A1 must be P, and _A must be Q. As soon as Jape knows both those things the proof will be over. We can maintain Jape's confusion as long as possible by telling it first that _A1 is P , either by Unify or hyp:

```
1: │ P→Q, Q→R, P  │ premises
   │ ...          │
2: │ P→_A         │
3: │ _A           │ →-E 1.3,2
   │ ...          │
4: │ _A→R         │
5: │ R            │ →-E 3,4
```

Then you can tell it, again using either Unify or hyp, that P→Q is the same as P→_A, or that Q→R is the same as _A→R, or that Q is the same as _A. Whichever way you choose, all the _As change into Qs and the proof is completed.

```
1: │ P→Q, Q→R, P  │ premises
2: │ Q            │ →-E 1.3,1.1
3: │ R            │ →-E 2,1.2
```

6. Scope boxing in Jape

The ∀-I rule of the QMW Introduction to Logic course looks something like this:

$$\begin{array}{rll}
i: & {}^{c}\begin{array}{|l|}\hline \ldots \\ A(c) \\ \hline \end{array} & \ldots \\
 & \ldots & \\
j: & \forall x.A(x) & \to -\forall - I\ i
\end{array}$$

in which the little *c* in the margin next to the shows that it's a 'scope box'; scope boxing imposes a condition on the proof that the name *c* doesn't appear outside the box. That condition stops you proving nonsense like ∀x.P(x) ⊢ ∃x.P(x) – it's not a theorem because an premise that *P*(*x*) holds universally doesn't prove that it holds at any particular position (the universe of possible values of *x* might be empty, and then the left-hand side would be trivially true while the right-hand side would be false).

Jape can't use scope-boxing directly, but it can imitate the mechanism reasonably well. It uses ordinary boxing with a pseudo-predicate 'var *c*' that means '*c* is in scope', and a side condition 'inscope *A*' that demands that all the free names in formula *A* are in scope. To use it you have to be a bit agile with the mouse, which is a pity but there it is.

Jape's versions of the ∀-I and ∃-E rules are[1]

$$\begin{array}{rll}
i: & \begin{array}{|l|}\hline \text{var } c \\ \ldots \\ A(c) \\ \hline \end{array} & \ldots \\
j: & & \\
 & \ldots & \\
k: & \forall x.A(x) & \to -\forall - I\ i\text{–}j
\end{array}
\qquad
\begin{array}{rll}
i: & \exists x.A(x) & \ldots \\
 & \ldots & \\
j: & \begin{array}{|l|}\hline \text{var } c, A(c) \\ \ldots \\ B \\ \hline \end{array} & \text{assumptions} \\
k: & & \ldots \\
 & \ldots & \\
l: & B & \exists - E\ i, j..k
\end{array}$$

The box in these diagrams are ordinary boxes. The "var *c*" pseudo-predicate is part of Jape's treatment of scope boxing: you can't appeal to var *c* outside the box in which it is introduced. Associated with each of these rules is a proviso that *c* is a 'fresh individual', and that is the other part of Jape's treatment. The proviso stops you using *c* in most places outside the box.

When you apply the ∀-E and ∃-I rules you have to make use of the variables that are introduced by the rules above. ∀-E and ∃-I use the "inscope *c*" pseudo-predicate to check that the name you use is in scope – that is, that you are inside a scope box for it:

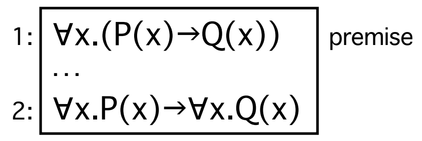
$$\begin{array}{rll}
i: & \forall x.A(x) & \ldots \\
 & \ldots & \\
j: & inscope\ c & \ldots \\
 & \ldots & \\
k: & A(c) & \forall - E\ i, j
\end{array}
\qquad
\begin{array}{rll}
i: & A(c) & \ldots \\
 & \ldots & \\
j: & inscope\ c & \ldots \\
 & \ldots & \\
k: & \exists x.A(x) & \exists\text{–}I\ i, j
\end{array}$$

"Inscope *c*" is a side-condition, which is hidden when it is proved and displayed when it is not proved. It's a bit like an unknown formula, unwelcome in a proof. You can avoid seeing it at all if you select the right argument formula when you apply the rule.

I recommend that you ***always give an argument*** to the ∀-E and ∃-I rules.

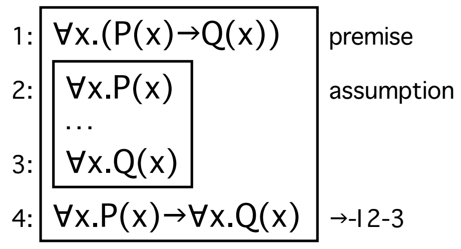
1 Behind the scenes Jape doesn't use scope boxing and it doesn't even use predicate notation. Those wo are interested may read more of the gory details in appendix I. With luck, you will never notice the difference.

Here’s an example to illustrate Jape’s treatment of scope boxing. The starting point is:



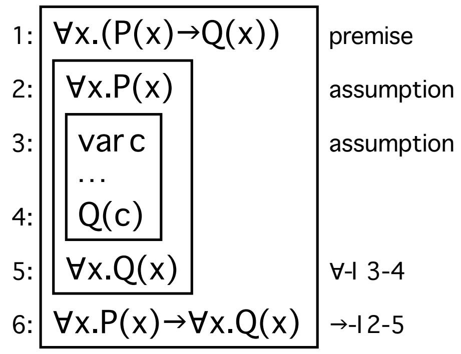
```
1: ∀x.(P(x)→Q(x))        premise
   ...
2: ∀x.P(x)→∀x.Q(x)
```

First I break down the conclusion with →-I:



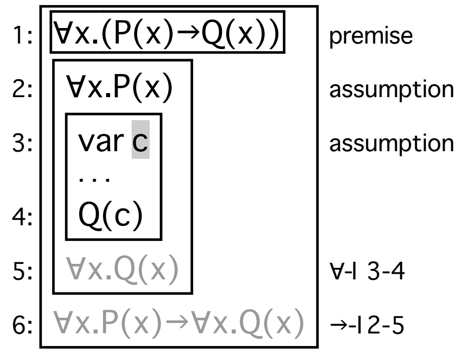
```
1: ∀x.(P(x)→Q(x))        premise
2:   ∀x.P(x)             assumption
     ...
3:   ∀x.Q(x)
4: ∀x.P(x)→∀x.Q(x)       →-I 2-3
```

Next I use ∀-I on line 3:



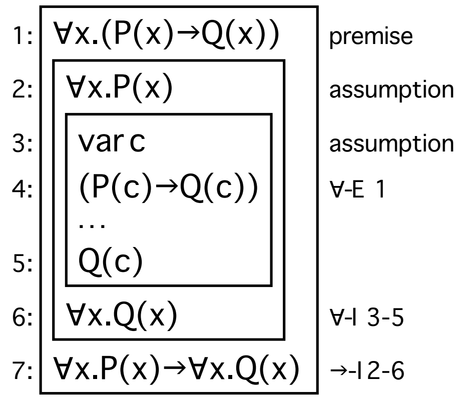
```
1: ∀x.(P(x)→Q(x))        premise
2:   ∀x.P(x)             assumption
3:     var c             assumption
       ...
4:     Q(c)
5:   ∀x.Q(x)             ∀-I 3-4
6: ∀x.P(x)→∀x.Q(x)       →-I 2-5
```

Line 3 is the first line of a scope box (it’s labelled ‘assumption’ but never mind). Next I can use ∀-E on line 1 or line 2. It doesn’t matter which I do first , so I choose line 1. Before I make the step I text-select *c* on line 3, because that’s the variable I want to use:



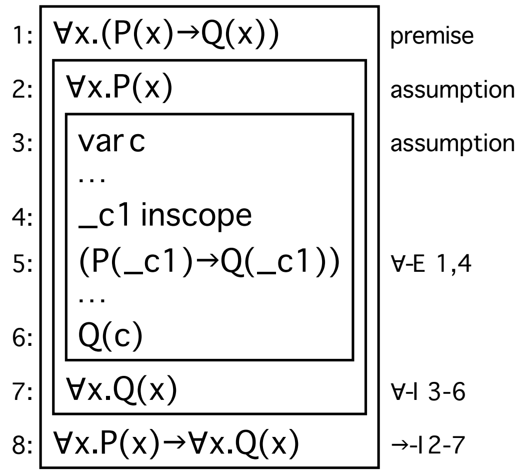
```
1: ∀x.(P(x)→Q(x))        premise
2:   ∀x.P(x)             assumption
3:     var c             assumption
       ...
4:     Q(c)
5:   ∀x.Q(x)             ∀-I 3-4
6: ∀x.P(x)→∀x.Q(x)       →-I 2-5
```

and then I apply the rule



```
1: ∀x.(P(x)→Q(x))        premise
2:   ∀x.P(x)             assumption
3:     var c             assumption
4:     (P(c)→Q(c))       ∀-E 1
       ...
5:     Q(c)
6:   ∀x.Q(x)             ∀-I 3-5
7: ∀x.P(x)→∀x.Q(x)       →-I 2-6
```

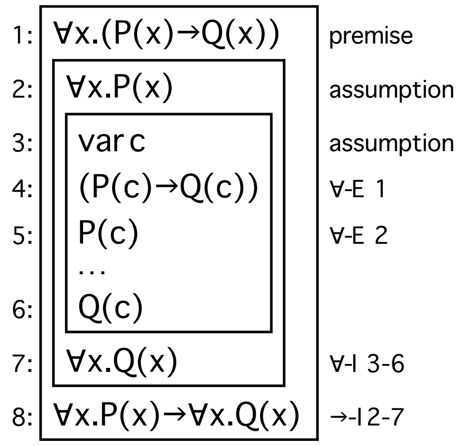
If I hadn't text-selected *c*, Jape would have used an unknown, and it would have had to show me the side-condition



```
1: ∀x.(P(x)→Q(x))          premise
2: ∀x.P(x)                 assumption
3: var c                   assumption
   ...
4: _c1 inscope
5: (P(_c1)→Q(_c1))         ∀-E 1,4
   ...
6: Q(c)
7: ∀x.Q(x)                 ∀-I 3-6
8: ∀x.P(x)→∀x.Q(x)         →-I 2-7
```

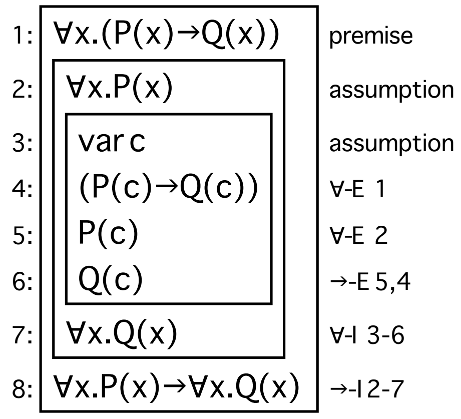
In this proof there is only one variable that is ever in scope, so it's clear to me that I would like *_c1* to be *c*. But it isn't clear to Jape (and there are lots of proofs where that's a good thing, so Jape never tries to guess what you mean). I can either unify *_c1* and *c* (by text-selecting them and choosing Unify from the Edit menu), or I can undo the last step, text-select *c* on line 3 and do the ∀-E step again. Either way, the inscope condition disappears.

If I do the same thing again with line 2 (text-select *c* somewhere, select line 2, apply ∀-E) then the proof moves one step farther forward:



```
1: ∀x.(P(x)→Q(x))          premise
2: ∀x.P(x)                 assumption
3: var c                   assumption
4: (P(c)→Q(c))             ∀-E 1
5: P(c)                    ∀-E 2
   ...
6: Q(c)
7: ∀x.Q(x)                 ∀-I 3-6
8: ∀x.P(x)→∀x.Q(x)         →-I 2-7
```

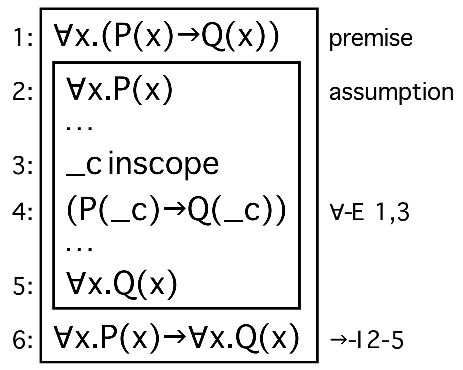
And finally a step of →-E finishes it off:



```
1: ∀x.(P(x)→Q(x))          premise
2: ∀x.P(x)                 assumption
3: var c                   assumption
4: (P(c)→Q(c))             ∀-E 1
5: P(c)                    ∀-E 2
6: Q(c)                    →-E 5,4
7: ∀x.Q(x)                 ∀-I 3-6
8: ∀x.P(x)→∀x.Q(x)         →-I 2-7
```

Enforcing the scope box condition

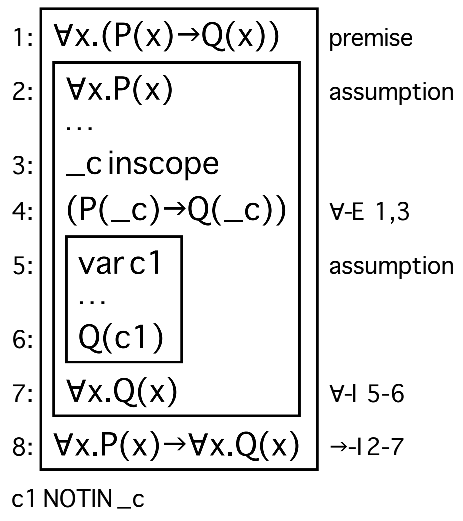
Suppose that I had done things in a slightly different order. Suppose that I had make a ∀-E step before the ∀-I step:



```
1: ∀x.(P(x)→Q(x))        premise
2:   ∀x.P(x)             assumption
     ...
3:   _c inscope
4:   (P(_c)→Q(_c))       ∀-E 1,3
     ...
5:   ∀x.Q(x)
6: ∀x.P(x)→∀x.Q(x)       →-I 2-5
```

There isn't a variable that I can select which is in scope, so I'm bound to provoke a display of the side condition.

Now I'm stuck. There is nothing I can do that will get rid of that side-condition. Even if I do the ∀-I step next, it doesn't help:



```
1: ∀x.(P(x)→Q(x))        premise
2:   ∀x.P(x)             assumption
     ...
3:   _c inscope
4:   (P(_c)→Q(_c))       ∀-E 1,3
5:     var c1            assumption
       ...
6:     Q(c1)
7:   ∀x.Q(x)             ∀-I 5-6
8: ∀x.P(x)→∀x.Q(x)       →-I 2-7
c1 NOTIN _c
```

That "var *c1*" is in the wrong place, and it's inside a scope box. Jape won't let you unify *_c* with *c1*, because a condition on the ∀-I rule is that the name *c1* is a 'fresh individual' in the proof: it has noticed the unknown *_c* and imposed a proviso (bottom left) that *c1* mustn't appear free in *_c*. Even if it would let you make the unification step, the box structure would still stop you completing the proof.

Some of the entries in the Conjectures panel – those which end with the word NOT – are included just so that you can see how scope-boxing stops some kinds of nonsense proofs. Many other entries, particularly those which are variations on De Morgan's laws, require considerable care in the use of the ∀-I and ∃-E rules, just to make things happen in the right order.

In general it is a good idea to use the ∀-I and ∃-E rules ***early***, and the ∀-E and ∃-I rules ***late***, to avoid falling foul of scope-boxing restrictions.

7. Some useful rules of thumb[1]

Rules of thumb don't always work, but you never know your luck. Try these, and if you don't understand them, look in the appendices for explanation:

- Get rid of → in conclusions as early as possible (using →-I)
- Get rid of ∨ in premises and assumptions as early as possible.
- Use ∀-I and ∃-E early, and ∀-E or ∃-I late, to avoid scope-boxing restrictions..
- As a last resort, use proof by contradiction: ¬-E backwards, followed by ¬-I backwards, followed by ∧-I and then perhaps by hyp – see the appendix on the ¬ rules for more explanation.

1 Some people believe that 'rule of thumb' refers to an ancient English law which governed the thickness of sticks which could be used by a husband in chastising a wife, and they say that we shouldn't use the phrase for that reason. That isn't where 'rule of thumb' comes from. The phrase derives from the fact that an inch is, originally by legal definition, approximately the width of a human adult male thumb. The French word for 'inch' – *pouce* actually means 'thumb'. So you can use your *thumb* as an *rule* –nowadays we would say *ruler* – if you are prepared to put up with a bit of inaccuracy: hence, by punning the meanings of 'rule', we get 'rule of thumb' for any procedure which sometimes works and sometimes doesn't.

Whilst I'm on the topic of measurement and its relation to body size, a foot was supposed to be the length of an adult male person's foot, and a yard the distance from his outstretched fingers to the tip of his nose. People were bigger when the measures were invented - really they were, it is a matter of diet. I'm approximately Dark Age Saxon standard size, and because of improved nutrition the average size of humans is rapidly increasing so that, in a generation or two, feet my size may once again be the male norm. A cubit, by the way, in case you are thinking of building any Arks, is the distance from elbow to finger end.

Belief in the existence of a law which allowed husbands to beat their wives with sticks of less than a particular thickness is very ancient (E.P. Thompson, in "Customs in Common", has a reference from the time of the enclosures in England) and it was talked about as 'the rule of thumb'. The name was, I suppose, a folk pun.

8. Using theorems and defining conjectures

The Apply button in the Conjectures panel lets you use theorems in proofs – ‘sequent substitution’ is the phrase that is used in the Introduction to Logic course to describe this action. I illustrated it above, in the last of the three alternative proofs of $P\to Q,\ Q\to R,\ P \vdash R$. In that proof I used the theorem $P,\ P\to Q \vdash Q$ twice: once to prove $P,\ P\to Q \vdash Q$, and once to prove $Q,\ Q\to R \vdash R$. Each of those sequents is a ‘substitution instance’ of the theorem; only the first happens to be identical to the theorem.

ItL Jape makes substitution instances of a theorem by replacing some or all of the names in its sequent. But only some names are replaceable, and it uses a simple lexical convention to help it to decide. Names in sequents fall into one of three classes:

- *variables*: any name starting with *x*, *y*, *z* or *c*;
- *formulas*: any name starting with *A*, *B*, *C*, *P*, *Q*, *R* or *S*.

Variable names in a theorem sequent can be replaced by any variable, formula names by any formula, constant names by any constant. This means, for example, that $P\to(Q\to R),\ P \vdash Q\to R$, and $(P1\wedge P2)\to(Q1\vee Q2), P1\wedge P1 \vdash Q1\vee Q2$ are each substitution instances of $P\to Q, P \vdash Q$.

I use formula names *A*, *B* and *C* in rules, *P*, *Q*, *R* and *S* in theorems.

Stating your own conjectures

The (New) button at the bottom of the Conjectures panel lets you define your own conjectures. It brings up a dialogue box (no illustration, I’m sorry to have to say again). You type the conjectured sequent into the box labelled CONJECTURE. (Usually that is all you need to do, but if the conjecture has any provisos you type them into the box labelled PROVISOS; if it has any parameter names you type them into the box labelled (...).) The keypad buttons at the bottom of the box are there so that you don’t have to learn how to type the special symbols that appear in logical formulæ and sequents. When you have finished, press the State Conjecture button (you need also to press Cancel in the Conjecture Entry dialogue box, because it doesn’t automatically disappear as it should): Jape adds your conjecture to the Conjectures panel and selects it, so that you can press Prove to start a proof of it.

When you are stating a conjecture, don’t forget to distinguish between the various classes of name (see above). If you use a name which is outside either class, Jape will refuse to accept the conjecture. If you state a conjecture which has the wrong kind of name in it – one which contains a formula like $\forall P.Q$, for example, which has a formula name where a variable name is required – Jape may accept it but you will find that the rules don’t actually apply.

Never mind all those complications! It’s actually very easy to define a conjecture and then to start a proof of it. For example, I typed $P\wedge\neg P\to Q$ into the CONJECTURE box, pressed State Conjecture in the dialogue window and then Prove in the Conjectures panel. Here’s the first step in the proof of that wild conjecture – it’s provable, but I’m not going to tell you how.

```
1: | P∧¬P |    assumption
   | ...  |
2: | Q    |
3: P∧¬P→Q   →-I 1-2
```

9. Reviewing and altering proofs of theorems

If you select a theorem in the Conjectures panel and press the Show button, you will be shown the last-registered proof of that theorem – only its final stage, not the intermediate steps. If you don't like what you see you can change it if you like, and register the changed proof with Done. Because Jape doesn't register the intermediate steps of a proof, you can't use Backtrack or Undo to trace through its development, but you can 'prune' parts of it and do them differently.

For example, consider the following proof of $P\land(Q\land R)\vdash(P\land Q)\land R$:

```
1: P∧(Q∧R)   premise
2: P         ∧-E(L) 1
3: (Q∧R)     ∧-E(R) 1
4: Q         ∧-E(L) 3
5: (P∧Q)     ∧-I 2,4
6: (Q∧R)     ∧-E(R) 1
7: R         ∧-E(R) 6
8: (P∧Q)∧R   ∧-I 5,7
```

Because of the order in which I developed the proof (first several ∧-Is, backwards, to reduce the conclusion, then ∧-Es , forwards, to simplify the premise), I've used ∧-E twice to get two lines which state $Q\land R$ – lines 3 and 6. I could have done things in another order. To show what I mean, I click on the conclusion in line 8 and choose Prune from the Edit menu, to see

```
1: P∧(Q∧R)   assumption
   ...
2: (P∧Q)∧R
```

Then I can use ∧-E four times to reduce the premise:

```
1: P∧(Q∧R)   assumption
2: P         ∧-E(L) 1
3: Q∧R       ∧-E(R) 1
4: Q         ∧-E(L) 3
5: R         ∧-E(R) 3
   ...
6: (P∧Q)∧R
```

and then ∧-I twice to complete the proof:

```
1: P∧(Q∧R)   premise
2: P         ∧-E(L) 1
3: (Q∧R)     ∧-E(R) 1
4: Q         ∧-E(L) 3
5: R         ∧-E(R) 3
6: (P∧Q)     ∧-I 2,4
7: (P∧Q)∧R   ∧-I 6,5
```

This version is one line shorter. It's not much of a difference, but I think it's an improvement. In other cases pruning and reproving can be much more effective.

Prove starts a new proof

If you select a theorem – a conjecture which you've already proved and registered with Jape – in the Conjectures panel and press Prove, Jape will start a new proof of that theorem, after first checking that that is what you really want to do.

Beware circular arguments!

Jape checks your proofs for circularity. It's possible to make a proof of theorem A, then prove theorem B using theorem A, then go back and re-'prove' theorem A using theorem B. The two proofs are now nonsense– A depends on B depends on A depends on B[1] ... – because you have made a 'circular argument'.

Jape doesn't check your proof as you make it, but when you think you've finished and try to register the new proof, it will refuse, telling you just how the proof is circular. Change the proof, or use Abandon Proof to forget all about it!

1 Well, sometimes complicated meta-argument using induction on the structure of your proofs can make this kind of circularity acceptable. It's simplest to say that the proofs are nonsense.

10. Printing a proof (File menu)

If you choose Print from the File menu, you will be presented with a dialogue box with some strange graphics on it, asking you to choose how your proof should be printed. Just press the Write File button if – as normally – you don't want to rotate or resize your proof before printing[1]. The proof will be sent to your default printer (the one named in your PRINTER environment variable, the same one that output is sent to when you use lpr to print a file from an xterm window). At the time of writing the proof that is printed doesn't have a title, but perhaps we will have fixed that before you read this sentence.

You may want to do more complicated things with your proof, and it's possible to save your proof as an EPS (Encapsulated PostScript) file (if you don't know what that is, relax and skip to the next section). This is what you do. Choose the "Output Proof as Scaled Postscript" entry in the File menu. A dialogue box is shown to you, called "select a file for the postscript proof", which for reasons cited above I unfortunately can't illustrate here. In that dialogue box you can enter a file name (next to File: at the bottom), or you can just press OK and it will choose a file name for you.

Next you will see another dialogue box – the same one as in the Print dialogue – which invites you to resize and/or rotate the printe. You can do wonderful things with this dialogue box – shrinking, growing, rotating the image – but almost certainly you will just need to press Write File.

Now you have to go to a terminal window, and type a UNIX command. If necessary you will have to create one – you're on your own here. Type

```
jlpr filename
```

or, if you know how to choose a printer to send your output to, type

```
jlpr -Pprinter filename
```

The filenames that Jape uses will all be of the form `ItLjape`*`nn`*`.jps`. Printers in use in the ITL at QMW are called itlaser1, itlaser3.

Good luck. It's been so long since I used UNIX much that I'd forgotten how ghastly it all is.

1 If you do want to rotate or resize, I think the controls in the dialogue box are straightforward. Anyway, you can experiment.

11. Saving and restoring your proofs (the File menu)

On the File menu there are a collection of commands to open, close and save files of logic descriptions and proofs. You won't need, and shouldn't use, the first, second and fourth commands (Select a top-level theory file, Add a theory file to the current theory, load the file ./J): if you do use them by accident, be sure to press Cancel in the file selection window or I shan't be liable for the consequences.

Save Proofs and Save As… will save your completed proofs, and any proofs in progress in the session window, in a file of your choice, using a file-selection dialogue (it's best to save them in files called *somethingorother*.jp). Load a file of proofs will load them back again (and it will automatically look for files called *whatsit*.jp, which is why it's best to save proofs in a file with that sort of name). If you have any unsaved proofs when you quit, or any proofs in progress in the session window, Jape will ask you if you want to save them.

If you want to build up a collection of proofs in a file, make sure you reload your old proofs before you start adding to the collection. If you start a new collection and then save it to the same file, X Jape will – without checking that it is what you want to do – overwrite the old collection.

Close seems to have the same effect as Quit Jape. Why do you need two identical commands on the same window? Search me!

The "Select a top-level theory file" command

The "Select a top-level theory file" command on the File menu lets you erase the logic loaded into Jape – both its rules and any proofs which you might have made – and load the rules of another logic. The dialogue is a little confusing and, in particular, once you have pressed Ok in the file selection window, pressing Cancel in later windows may not have the effect you expect.

There are some interesting logics encoded in Japeish, which can be found in the `/import/jape/examples` directory. If you look at them then you are on your own: I take no responsibility for any confusion which any of those logics may cause!

Appendix A: Using the mouse

X Jape understands various mouse gestures in the proof pane:

- a single *left-button click* selects a conclusion or an assumption to work on;
- a *double left-button click* can automatically apply a rule to a conclusion or assumption (***not available in ItL Jape***);
- a single *middle-button click* to select a character as an argument to a rule;
- *press and drag* with the *middle button* down to select a sequence of characters as argument to a rule;
- *press and drag*with the *middle button* down over a blank portion of the pane to move the proof within the pane.

In addition, various mouse gestures cancel the effect of some of the above:

- a single *left-button click* in a blank area of the screen cancels all conclusion, assumption, character and text selections;
- a *double middle-button click* over a text selection cancels it (and, currently, any other text selections in the same formula, but not any in other formulæ);
- a *double middle-button click* in a blank area of the screen cancels all text selections.

Single-left-click

The effect of the single-click gesture is illustrated in the examples above. A square box is drawn round the conclusion or assumption you clicked on, and it becomes a focus of attention for the command you apply. You can select both a conclusion and an assumption, but no more than one of each kind.

When you select a conclusion, all the other conclusions are greyed-out to show that you can't use them. Only the assumptions appropriate to that conclusion are still shown, and some conclusions proved by forward reasoning are still shown – you can use them, as you would expect, just as if they were assumptions.

When you select an assumption, the conclusions which aren't associated with it are greyed-out.

If you select only an assumption, there may be many conclusions with which it is associated. In order to make a proof step, Jape needs to know which conclusion you are trying to prove. It will ask you to select a conclusion if necessary.

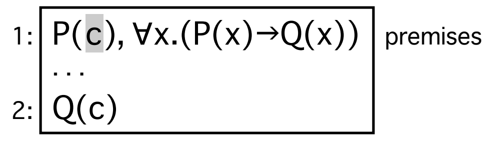
Left-clicking in a blank area of the pane cancels all formula and text selections.

Double-left-click

Jape allows the logic designer to define the effect of a double-click on an assumption or conclusion, so that rules can be 'automatically' applied. We have decided not to use that facility in ItL Jape, because we want you to think about, and learn about, the choice of rules.

Middle-click and middle-press-and-drag

If you click the middle button over a character it is text-selected; if you press the middle button over a character and move the mouse left or right, that character and the others you drag over are text-selected; in either case the selected text is highlighted. It looks like this:



```
1: P(c), ∀x.(P(x)→Q(x))   premises
   . . .
2: Q(c)
```

When you next apply a rule, the highlighted text is given as an argument to that rule: that's mainly useful in the rules which have to do with ∀ and ∃ (see below). If the text won't do – for example because it isn't a proper logical formula, or because it is the wrong kind of formula – Jape will complain.

You can cancel a text selection that you have already made either by double-middle clicking it, or by double-middle clicking in a blank area of the proof pane.

Jape is sometimes a bit fussy in its reading of the mouse position, especially if the text you are selecting is at the very beginning or the very end of an assumption or conclusion.

Appendix B – Troubleshooting

I didn't write the graphical interface to X Jape, so I can't say much about what to do if it won't start, or if the windows don't perform as described. Mail me (richard@dcs.qmw.ac.uk) if you have real difficulty, and I'll do my best to help.

If you lose your Jape windows, clicking on the screen background with the middle button gives you a list of all your windows; if only one Jape window is shown, choose it and the others should appear.

What if a proof step goes wrong?

When you try to apply a rule one of two things can happen. Either the rule applies, and the step goes through, or it doesn't, and Jape shows you an error message. At first the error messages may seem very complicated and rather confusing – indeed, they could do with some simplification[1] – and often the best thing is to read them as if they said "the rule doesn't apply, so you will have to try something else".

As always when using machinery, there can be more than one explanation for a rule's failure. It may be that the rule *does* apply, but you didn't select the right formula for it to work on. Or it may be that you have chosen the wrong rule. All you can do is to try to see why the rule didn't work, and then go back and try to fix the problem somehow, either by applying the same rule differently, or by applying a different rule.

And then, even though a rule does apply and the proof step does go through, it may not turn out to be the right thing to do. Sometimes a successful step now can lead to a dead end later. Sometimes a step works, but not in the way that you expected – perhaps lots of unknowns suddenly appear in the proof, or there are lots of extra lines that you didn't expect, or lots of lines suddenly disappear.

Whenever something happens that isn't what you intended, the first stage of a cure is to use the Undo command (choose it from the Edit menu). Undo takes you back one step in the proof, two Undos take you back two steps, and so on. Using Undo can reverse an unintended step; using several can move you back from a dead end to an earlier position from which you can move forward in a different direction.

You can even recover from Undo! The Redo command (in the Edit menu) reverses the effect of Undo, two Redos reverse two Undos, and so on. So if you decide, after Undoing, that you really did want to make that surprising step after all, Redo will make it again. (If you Undo and make a new proof step, then the one you Undid is gone for ever, like it or not.)

The Undo command allows you to explore if you don't know what rule or theorem to apply in a proof: you can experiment with different rules and theorems from the menus until you find one that works. That can be a bad thing, if you just try things at random until you find one that happens to work, and indeed such a search usually turns out to be a very slow way to make proofs. But sometimes we all need to search for a proof, and then Undo and Redo are invaluable. I hope that when you do search and find a surprising avenue that happens to work,. you will pause to ask yourself *why* it works. Jape is designed to support that kind of 'reflective exploration' – it helps with the exploring part, and you learn by reflecting on the results.

1 Even though they have improved a lot recently, Jape's error messages are still awful and most of the time they are of very little use at all. If you get a really bad one and I'm around you can show the problem to me, because there might be something in Jape that I can fix. But quite often I may say that I can't do anything about it this millenium. The messages have even got a little *worse* over the summer. Ho hum!

Appendix C: The → rules

Normally you will use the → *elimination* rule to make a forward step from an assumption (or from a conclusion already proved by a forward step) and you will use the → *introduction* rule to make a backward step from a conclusion. In either case the assumption or conclusion should be of the form *something* → *something else*, and in either case the effect is to *simplify* the assumption or conclusion you selected by splitting it into two parts and eliminating the → connective.

You can use the → elimination rule backwards, and in that case it can make sense to give it an argument – see below.

Forward reasoning with →-E

The →-E rule in box form is

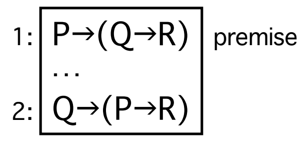
$$\begin{array}{lll} i\text{:} & A & \ldots \\ & \ldots & \\ j\text{:} & A \rightarrow B & \ldots \\ & \ldots & \\ k\text{:} & B & \rightarrow\!-E\ i,j \end{array}$$

and in tree form

$$\dfrac{\begin{array}{cc} \vdots & \vdots \\ A & A \rightarrow B \end{array}}{B} \rightarrow -E$$

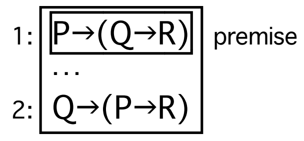
– from a proof of *A* and a proof of *A*→*B*, make a proof of *B*.

Select an assumption (or a conclusion already proved by a forward step) which has an arrow you wish to eliminate, select the conclusion if necessary, and apply the rule. For example in



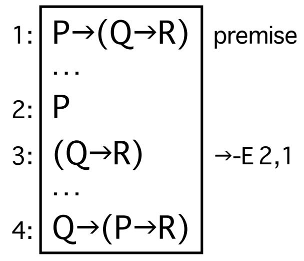
```
1: P→(Q→R)   premise
   ...
2: Q→(P→R)
```

select the premise:



```
1: P→(Q→R)   premise
   ...
2: Q→(P→R)
```

and apply →-E:



```
1: P→(Q→R)   premise
   ...
2: P
3: (Q→R)     →-E 2,1
   ...
4: Q→(P→R)
```

The selected assumption provides one of the antecedents of the rule. If there is an assumption (or a conclusion already proved by forward reasoning) which corresponds to the other antecedent, Jape uses that. If not, it introduces a new unproved conclusion, as in this example.

Don't give an argument when using →-E for forward reasoning – it only confuses Jape.

Backward reasoning with →-I

The →-I rule in box form is

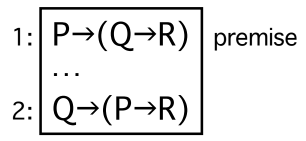
$$\begin{array}{lll} i: & \begin{array}{|l|}\hline A \\ \ldots \\ B \\ \hline\end{array} & \begin{array}{l}\text{assumption}\\ \\ \ldots\end{array} \\ j: & & \\ & \ldots & \\ k: & A\rightarrow B & \rightarrow -I\ i..j \end{array}$$

and in tree form

$$\dfrac{\begin{array}{c}[A]\\ \vdots \\ B\end{array}}{A\rightarrow B}\rightarrow -I$$

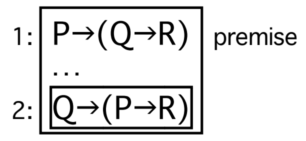
– from a proof of *B*, within which you are allowed to assume a proof of *A*, construct a proof of *A*→*B*.

Select an unproved conclusion which includes an arrow you want to eliminate, and apply →-I. For example in



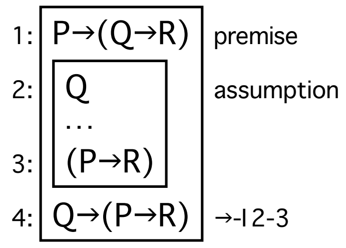
```
1: P→(Q→R)   premise
   ...
2: Q→(P→R)
```

select the conclusion



```
1: P→(Q→R)   premise
   ...
2: Q→(P→R)
```

and apply →-I:



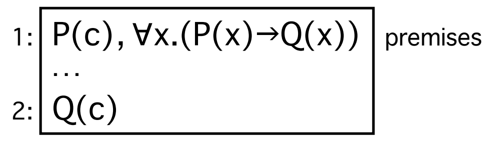
```
1: P→(Q→R)   premise
2:   Q       assumption
     ...
3:   (P→R)
4: Q→(P→R)   →-I 2-3
```

The new assumption is automatically introduced, and the necessary enclosing box drawn. The application would have the same effect even if you hadn't selected the conclusion, because there is only one open conclusion.

Notice how this step has automatically introduced the correct assumption in the new box. Jape relieves you of the burden of inventing assumptions, like it or not.

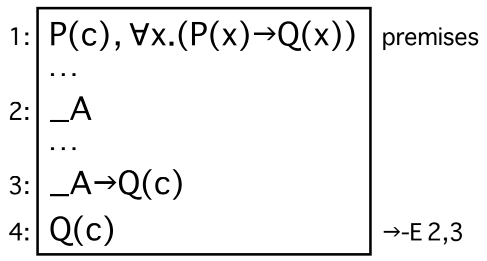
Backward reasoning with →-E

It is possible to use the →-E rule to reason backwards from *any* conclusion *B*, whatever shape it is; Jape will supply an unknown for the antecedent *A*. For example, in



```
1: P(c), ∀x.(P(x)→Q(x))   premises
   ...
2: Q(c)
```

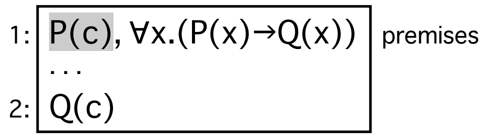
you can apply →-E immediately:



```
1: P(c), ∀x.(P(x)→Q(x))   premises
   ...
2: _A
   ...
3: _A→Q(c)
4: Q(c)                   →-E 2,3
```

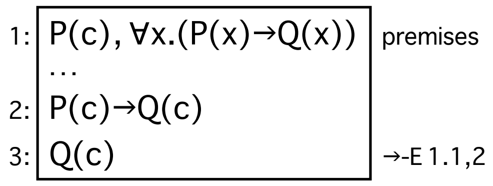
Jape will introduce an unknown _A (or some name like that) and demand that you provide a proof both of _A and _A→*the conclusion.*

If you give an argument to the rule, by selecting text, Jape will use that argument instead of _A. For example, in the same problem as above, it might be reasonable to use P(c) as the argument. Text-select (middle-press-and-drag) that formula (but *don't* formula-select it with a left-click!!!):



```
1: P(c), ∀x.(P(x)→Q(x))   premises
   ...
2: Q(c)
```

and apply →-E



```
1: P(c), ∀x.(P(x)→Q(x))   premises
   ...
2: P(c)→Q(c)
3: Q(c)                   →-E 1.1,2
```

Whichever way you do it, it doesn't matter: this proof is doomed (see the discussion of scope boxing above and of the ∀ rules below.

Appendix D: the ∧ rules

Normally you will use one of the ∧ *elimination* rules to make a forward step from an assumption (or from a conclusion already proved by a forward step) and you will use the ∧ *introduction* rule to make a backward step from a conclusion. In either case the assumption or conclusion should be of the form *something* ∧ *somethingelse*, and in either case the effect is to *simplify* the assumption or conclusion you selected by breaking it into two parts (and perhaps discarding one part) and eliminating the ∧ connective.

You can use the ∧ elimination rule backwards, and in that case it can make sense to give it an argument – see below.

Forward reasoning with ∧-E (L or R)

The ∧-E rules in box form are:

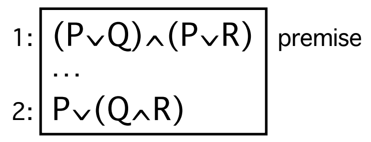
$$\begin{array}{lll} i: & A \wedge B & \ldots \\ & \ldots & \\ j: & A & \wedge - E(L)\, i \end{array} \qquad \begin{array}{lll} i: & A \wedge B & \ldots \\ & \ldots & \\ j: & B & \wedge - E(R)\, i \end{array}$$

and in tree form

$$\dfrac{\begin{array}{c}\vdots \\ A \wedge B\end{array}}{B}\ \wedge -E(L) \qquad \dfrac{\begin{array}{c}\vdots \\ A \wedge B\end{array}}{B}\ \wedge -E(R)$$

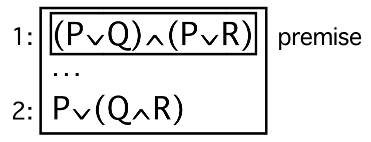
– from a proof of *A*∧*B* you can make a proof of *A* or a proof of *B*, as you wish. In lectures these may have been presented as a single rule, or perhaps as two rules with a single name. Strictly speaking (and machines have to be instructed very strictly) there are two similar but distinct rules, and therefore Jape must be told about two rules.

Suppose there is an assumption, or a conclusion already proved by a forward step, of the form *A*∧*B* and you want to use either *A* or *B* in the proof. You select the assumption and apply the relevant rule. For example, in



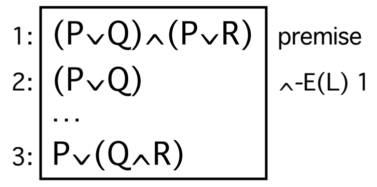
```
1: (P∨Q)∧(P∨R)   premise
   ...
2: P∨(Q∧R)
```

select the premise



```
1: (P∨Q)∧(P∨R)   premise
   ...
2: P∨(Q∧R)
```

and apply ∧-E(L):



```
1: (P∨Q)∧(P∨R)   premise
2: (P∨Q)         ∧-E(L) 1
   ...
3: P∨(Q∧R)
```

Backward reasoning using ∧-I

The ∧-I rule in box form is

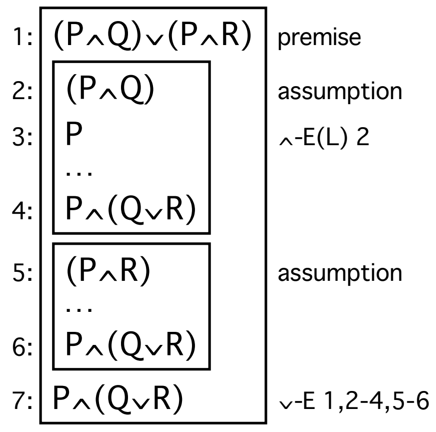
$$\begin{array}{rll} i: & A & \ldots \\ & \ldots & \\ j: & B & \ldots \\ & \ldots & \\ k: & A \wedge B & \wedge - I\ i, j \end{array}$$

and in tree form:

$$\dfrac{\begin{array}{cc} \vdots & \vdots \\ A & B \end{array}}{A \wedge B} \wedge - I$$

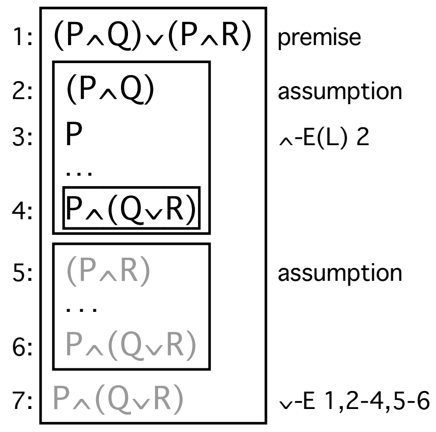
– from a proof of *A* and a proof of *B* make a proof of *A*∧*B*.

Select an unproved conclusion which contains an ∧ you want to get rid of, and apply the rule. Jape introduces two new unproved conclusions, or uses assumptions if they are available. For example, in



```
1: (P∧Q)∨(P∧R)   premise
2:   (P∧Q)       assumption
3:   P           ∧-E(L) 2
     ...
4:   P∧(Q∨R)
5:   (P∧R)       assumption
     ...
6:   P∧(Q∨R)
7: P∧(Q∨R)       ∨-E 1,2-4,5-6
```

select the conclusion on line 4



```
1: (P∧Q)∨(P∧R)   premise
2:   (P∧Q)       assumption
3:   P           ∧-E(L) 2
     ...
4:   P∧(Q∨R)
5:   (P∧R)       assumption
     ...
6:   P∧(Q∨R)
7: P∧(Q∨R)       ∨-E 1,2-4,5-6
```

and apply ∧-I:

```
1: (P∧Q)∨(P∧R)      premise
2:   (P∧Q)          assumption
3:   P              ∧-E(L) 2
     ...
4:   (Q∨R)
5:   P∧(Q∨R)        ∧-I 3,4
6:   (P∧R)          assumption
     ...
7:   P∧(Q∨R)
8: P∧(Q∨R)          ∨-E 1,2-5,6-7
```

Line 3 proves P, but you still have to make a proof of Q∨R.

Backward reasoning with ∧-E

Occasionally ∧ elimination might be the *last* step in a proof. You can then use the ∧ elimination rules backwards to prove any conclusion *A*∧*something* or *something*∧*B*. Jape will normally introduce an unknown for *B* in the (L) rule or *A* in the (R) rule, but you can provide an argument if you wish.

For example, consider

```
1: P→(Q∧R), P    premises
   ...
2: Q
```

The obvious first step in this contrived example is to use → elimination on the first premise. But the last step in the proof is bound to be ∧ elimination, to make a proof of Q from a proof of Q∧R. If you apply ∧-E(L) backwards – no need to select the conclusion, because it is the only one – Jape will show you

```
1: P→(Q∧R), P    premises
   ...
2: Q∧_B
3: Q             ∧-E(L) 2
```

If you find that unknown offensive, or confusing, or messy, you can text-select R before you apply the rule, and Jape will construct

```
1: P→(Q∧R), P    premises
   ...
2: Q∧R
3: Q             ∧-E(L) 2
```

Appendix E: the ∨ rules

Normally you will use the ∨ *elimination* rule to make a forward step from an assumption (or from a conclusion already proved by a forward step) and you will use one of the ∨ *introduction* rules to make a backward step from a conclusion. In either case the assumption or conclusion should be of the form *something* ∨ *somethingelse*, and in either case the effect is to *simplify* the assumption or conclusion you selected.

You can use the ∨ introduction rules to reason forwards, and in that case it usually makes sense to give the rule an argument – see below. You can use the ∨ elimination rule backwards, but it is quite tricky and you will normally have to use *hyp* or other even more arcane tricks to tidy up afterwards.

Forward reasoning using ∨-E

The ∨-E rule in box form is

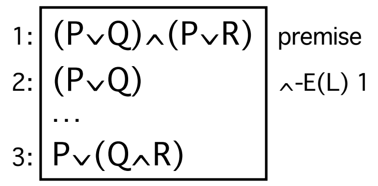
$$\begin{array}{lll} i: & A \vee B & \ldots \\ & \ldots & \\ j: & \boxed{\begin{array}{l} A \\ \ldots \\ C \end{array}} & \begin{array}{l} \text{assumption} \\ \\ \ldots \end{array} \\ k: & & \\ & \ldots & \\ l: & \boxed{\begin{array}{l} B \\ \ldots \\ C \end{array}} & \begin{array}{l} \text{assumption} \\ \\ \ldots \end{array} \\ m: & & \\ & \ldots & \\ n: & C & \vee-E\ i, j..k, l..m \end{array}$$

and in tree form

$$\dfrac{\begin{array}{c} \\ \vdots \\ A \vee B \end{array} \quad \begin{array}{c} [A] \\ \vdots \\ C \end{array} \quad \begin{array}{c} [B] \\ \vdots \\ C \end{array}}{C} \vee E$$

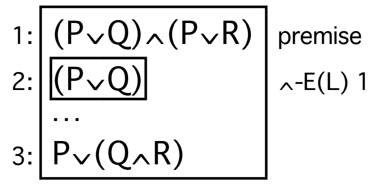
– from a proof of $A{\vee}B$, a proof of C (which can assume a proof of A) and another proof of the same C (which this time can can assume a proof of B), make a proof of C.

To use the rule, select an assumption, or a conclusion already proved by forward step, which includes a ∨ that you want to get rid of. For example, in



```
1: (P∨Q)∧(P∨R)   premise
2: (P∨Q)         ∧-E(L) 1
   ...
3: P∨(Q∧R)
```

select the conclusion on line 2



```
1: (P∨Q)∧(P∨R)   premise
2: (P∨Q)         ∧-E(L) 1
   ...
3: P∨(Q∧R)
```

and apply ∨-E:

```
1: │(P∨Q)∧(P∨R)     │ premise
2: │(P∨Q)           │ ∧-E(L) 1
3: │ ┌───────────┐  │ assumption
   │ │P          │  │
   │ │...        │  │
4: │ │P∨(Q∧R)    │  │
   │ └───────────┘  │
5: │ ┌───────────┐  │ assumption
   │ │Q          │  │
   │ │...        │  │
6: │ │P∨(Q∧R)    │  │
   │ └───────────┘  │
7: │P∨(Q∧R)         │ ∨-E 2,3-4,5-6
```

Jape copies the conclusion you are trying to prove (in this case it's on line 3 in the original box) and makes it the conclusion of each of two new boxes, introducing the two parts of the original formula as additional assumptions in the new boxes.

Backward reasoning using ∨-I

The ∨ introduction rules in box form are

$$\begin{array}{llllll} i: & A & \ldots & i: & B & \ldots \\ & \ldots & & & \ldots & \\ j: & A \vee B & \vee-I(L)\,i & j: & A \vee B & \vee-I(R)\,i \end{array}$$

and in tree form

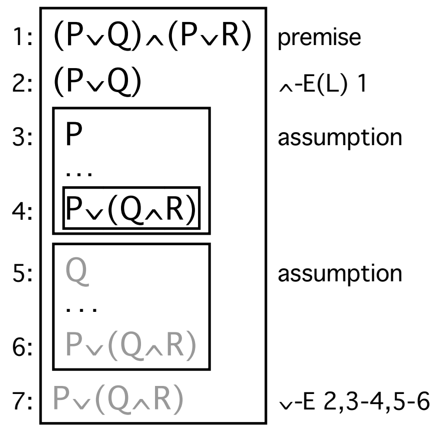
$$\dfrac{\begin{matrix}\vdots\\ A\end{matrix}}{A \vee B}\ \vee-I(L) \qquad\qquad \dfrac{\begin{matrix}\vdots\\ B\end{matrix}}{A \vee B}\ \vee-I(R)$$

– from a proof of *A* or a proof of *B* you can make a proof of *A*∨*B*. As with the ∧ elimination rules, there really are two similar but distinct rules and therefore they need distinct Jape names.

Select an unproved conclusion which contains a ∨ which you want to get rid of, and apply one of these rules. For example, in

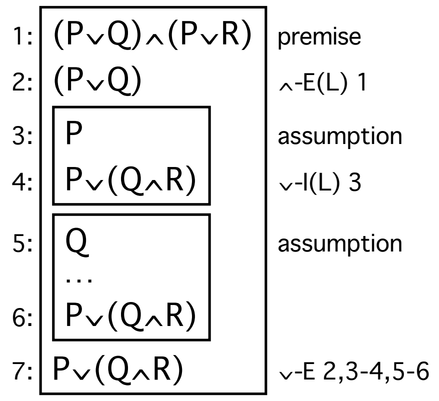
```
1: │(P∨Q)∧(P∨R)     │ premise
2: │(P∨Q)           │ ∧-E(L) 1
3: │ ┌───────────┐  │ assumption
   │ │P          │  │
   │ │...        │  │
4: │ │P∨(Q∧R)    │  │
   │ └───────────┘  │
5: │ ┌───────────┐  │ assumption
   │ │Q          │  │
   │ │...        │  │
6: │ │P∨(Q∧R)    │  │
   │ └───────────┘  │
7: │P∨(Q∧R)         │ ∨-E 2,3-4,5-6
```

select the conclusion on line 4



```
1: (P∨Q)∧(P∨R)   premise
2: (P∨Q)         ∧-E(L) 1
3:   P           assumption
     ...
4:   P∨(Q∧R)
5:   Q           assumption
     ...
6:   P∨(Q∧R)
7: P∨(Q∧R)       ∨-E 2,3-4,5-6
```

and apply ∨-I(L):

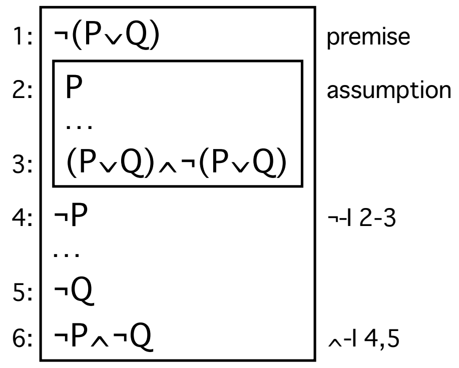


```
1: (P∨Q)∧(P∨R)   premise
2: (P∨Q)         ∧-E(L) 1
3:   P           assumption
4:   P∨(Q∧R)     ∨-I(L) 3
5:   Q           assumption
     ...
6:   P∨(Q∧R)
7: P∨(Q∧R)       ∨-E 2,3-4,5-6
```

Forward reasoning using ∨-I

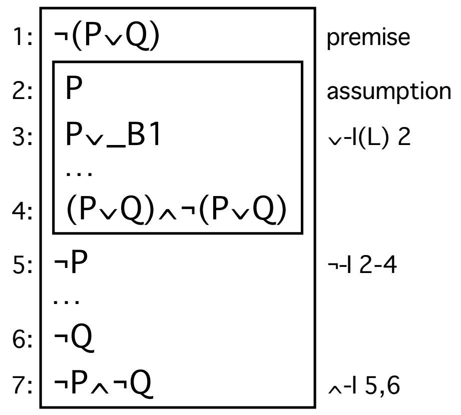
If you have an assumption P, or a conclusion P proved by a forward step, then you can conclude either P∨B or A∨P for any formula A or B. If you don't provide an argument by text selection then Jape will use an unknown instead.

For example, in



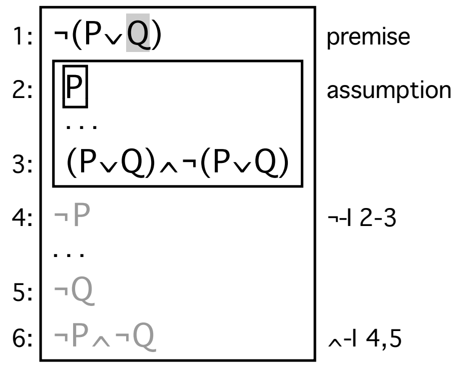
```
1: ¬(P∨Q)            premise
2:   P               assumption
     ...
3:   (P∨Q)∧¬(P∨Q)
4: ¬P                ¬-I 2-3
   ...
5: ¬Q
6: ¬P∧¬Q             ∧-I 4,5
```

you can select line 2 and use ∨-I (L):



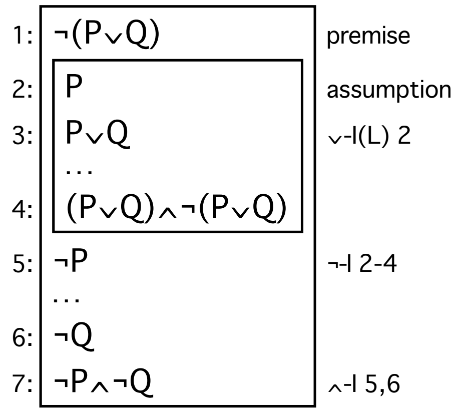
```
1: ¬(P∨Q)                    premise
2:   P                       assumption
3:   P∨_B1                   ∨-I(L) 2
     ...
4:   (P∨Q)∧¬(P∨Q)
5: ¬P                        ¬-I 2-4
   ...
6: ¬Q
7: ¬P∧¬Q                     ∧-I 5,6
```

In this case it is clear that the unknown should be unified with Q, and you can achieve that effect in a number of ways. If you select Q somewhere in the proof before you apply the rule:



```
1: ¬(P∨Q)                    premise
2:   P                       assumption
     ...
3:   (P∨Q)∧¬(P∨Q)
4: ¬P                        ¬-I 2-3
   ...
5: ¬Q
6: ¬P∧¬Q                     ∧-I 4,5
```

then the ∨-I (L) rule will do all the work itself:



```
1: ¬(P∨Q)                    premise
2:   P                       assumption
3:   P∨Q                     ∨-I(L) 2
     ...
4:   (P∨Q)∧¬(P∨Q)
5: ¬P                        ¬-I 2-4
   ...
6: ¬Q
7: ¬P∧¬Q                     ∧-I 5,6
```

Similar tricks work with the ∨-I (L) rule.

Backward reasoning using ∨-E

Consider the following conjecture:

```
   ...
1: (P→Q)∨(Q→P)
```

It is provable in ItL Jape, but it's tricky. One way to prove it is to use ∨ elimination from a proof of P∨¬P. Since that isn't an assumption or a rule (and why should it be? it's provable itself!) you have to describe what you are doing to Jape.

The first step is to apply ∨-E:

```
     ...
  1: _A∨_B
  2: | _A                  |  assumption
     | ...                 |
  3: | (P→Q)∨(Q→P)         |
  4: | _B                  |  assumption
     | ...                 |
  5: | (P→Q)∨(Q→P)         |
  6: (P→Q)∨(Q→P)              ∨-E 1,2-3,4-5
```

Jape constructs a proof with two unknowns in it. You can soldier on with the proof, leaving the unknowns unknown, keeping in mind that you actually meant P∨¬P, or you can tell it what is going on. Jape's argument mechanism isn't much help here, because the rule needs two arguments, and Jape has no mechanism which you can use to tell it what the two arguments are.

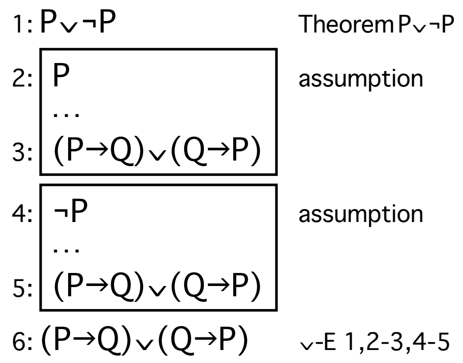
The neatest solution to this problem is to make a proof of P∨¬P, complete it with the Done command (so that the proof makes an entry in the Theorems menu), go back to the the proof of (P→Q)∨(Q→P) and prove line 1 by that theorem. Since you are proving _A∨_B by P∨¬P, _B must be ¬_A:

```
  1: _A∨¬_A                   Theorem P∨¬P
  2: | _A                  |  assumption
     | ...                 |
  3: | (P→Q)∨(Q→P)         |
  4: | ¬_A                 |  assumption
     | ...                 |
  5: | (P→Q)∨(Q→P)         |
  6: (P→Q)∨(Q→P)              ∨-E 1,2-3,4-5
```

Sort out the confusion about unknowns – text-select both _A and P (see 'Unknowns, unification and the hyp rule' above if you don't know how to do this):

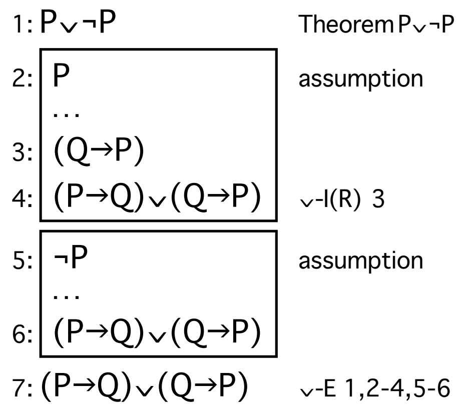
```
  1: _A∨¬_A                   Theorem P∨¬P
  2: | _A                  |  assumption
     | ...                 |
  3: | (P→Q)∨(Q→P)         |
  4: | ¬_A                 |  assumption
     | ...                 |
  5: | (P→Q)∨(Q→P)         |
  6: (P→Q)∨(Q→P)              ∨-E 1,2-3,4-5
```

and then use the Unify command to get to the screen you really wanted:



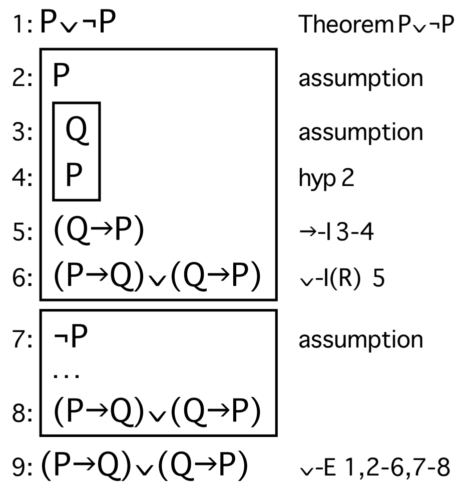
```
1: P∨¬P                  Theorem P∨¬P
2: │ P                   assumption
   │ ...
3: │ (P→Q)∨(Q→P)
4: │ ¬P                  assumption
   │ ...
5: │ (P→Q)∨(Q→P)
6: (P→Q)∨(Q→P)           ∨-E 1,2-3,4-5
```

Then pick out the second half of line 3 by ∨-I(R):



```
1: P∨¬P                  Theorem P∨¬P
2: │ P                   assumption
   │ ...
3: │ (Q→P)
4: │ (P→Q)∨(Q→P)         ∨-I(R) 3
5: │ ¬P                  assumption
   │ ...
6: │ (P→Q)∨(Q→P)
7: (P→Q)∨(Q→P)           ∨-E 1,2-4,5-6
```

prove that implication by →-I:



```
1: P∨¬P                  Theorem P∨¬P
2: │ P                   assumption
3: │ │ Q                 assumption
4: │ │ P                 hyp 2
5: │ (Q→P)               →-I 3-4
6: │ (P→Q)∨(Q→P)         ∨-I(R) 5
7: │ ¬P                  assumption
   │ ...
8: │ (P→Q)∨(Q→P)
9: (P→Q)∨(Q→P)           ∨-E 1,2-6,7-8
```

– and so on. Notice the justification quoted for line 4: this is one of the cases where an explicit hyp line appears in a proof. It’s hard to see how it could be avoided.

I leave the second half of the proof to you: it involves a contradiction and, of course, uses the other side of the conclusion. Of course, the proof would be almost the same shape if you took _A to be Q, because the conclusion is symmetrical ...

Appendix F: the ¬ rules

I find the rules for ¬, as given in the ItL course, very *un*natural. But you have to be able to use them just as they are described in the course, and therefore ItL Jape has to include them just like that, so here goes.

You can use the ¬ elimination rule forward, and the ¬ introduction rule backward, like any of the other rules. But often you use them as a pair, and then as a last resort, to construct a ‘proof by contradiction’. I’ll explain what I mean below.

Forward reasoning using ¬-E

Sometimes, very occasionally, you may want to use ¬-E in this way. The rule in box form is

$$\begin{array}{lll} i: & \neg\neg A & \ldots \\ & \ldots & \\ j: & A & \neg-E\ i \end{array}$$

and in tree form

$$\dfrac{\begin{array}{c}\vdots \\ \neg\neg A\end{array}}{A}\ \neg-E$$

– from a proof of $\neg\neg A$ you can conclude A.

If you have an assumption of the form $\neg\neg A$ then you can select it and apply the ¬-E rule. For example, in

$$\begin{array}{r|l|l} \cline{2-2} 1: & \neg\neg P & \text{premise} \\ & \ldots & \\ 2: & Q\rightarrow P & \\ \cline{2-2} \end{array}$$

you can select the premise

$$\begin{array}{r|l|l} \cline{2-2} 1: & \boxed{\neg\neg P} & \text{premise} \\ & \ldots & \\ 2: & Q\rightarrow P & \\ \cline{2-2} \end{array}$$

and then apply ¬-E:

$$\begin{array}{r|l|l} \cline{2-2} 1: & \neg\neg P & \text{premise} \\ 2: & P & \neg\text{-E } 1 \\ & \ldots & \\ 3: & Q\rightarrow P & \\ \cline{2-2} \end{array}$$

You can take it from there ...

Backward reasoning using ¬-I

The ¬-I rule is

```
i : | A          |  assumption
    | ...        |
j : | B ∧ ¬B     |  ...
    ...
k : ¬A              ¬ – I i..j
```

and in tree form

$$\dfrac{\begin{matrix}[A]\\ \vdots \\ B\wedge\neg B\end{matrix}}{\neg A}\;\neg - I$$

– if from assumption *A* you can prove the contradiction *B*∧¬*B*, then *A* can't be valid and you have made a proof of ¬*A*. The formula *B* is arbitrary, and if you apply this rule without providing an argument Jape will introduce an unknown. In the case of the ¬-I rule you will often find that it is not helpful to try to provide an argument: better to explore the proof a little farther to see what turns up before you decide what *B* should be. In some cases *A* and *B* turn out to be the same formula, or one turns out to be the negation of the other, which can be surprising but which makes things very simple indeed when it happens!

Suppose that you have to prove

```
1: | P→Q       |  premise
   | ...       |
2: | ¬(P∧¬Q)   |
```

The conclusion is of the form ¬*something*, so the ¬-I rule is applicable. Apply it and you see

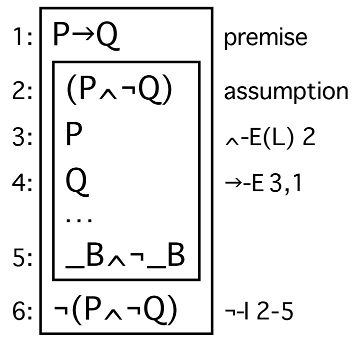
```
1: | P→Q           |  premise
2: | | (P∧¬Q)   | |  assumption
   | | ...       | |
3: | | _B∧¬_B   | |
4: | ¬(P∧¬Q)       |  ¬-I 2-3
```

We can let the unknown _B ride along while we explore to see what the contradiction is going to be.

Now on line 1 there is an implication, and on line 2 the antecedent of that implication is part of a conjunction, so it might be a good idea to try to simplify line 1. Use →-E on line 1:

```
1: | P→Q           |  premise
2: | | (P∧¬Q)   | |  assumption
   | | ...       | |
3: | | P         | |
4: | | Q         | |  →-E 3,1
   | | ...       | |
5: | | _B∧¬_B   | |
6: | ¬(P∧¬Q)       |  ¬-I 2-5
```

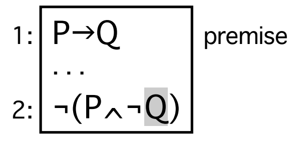
and then select lines 2 and 3 and use ∧-E(L):



```
1: P→Q              premise
2:   (P∧¬Q)         assumption
3:   P              ∧-E(L) 2
4:   Q              →-E 3,1
     ...
5:   _B∧¬_B
6: ¬(P∧¬Q)          ¬-I 2-5
```

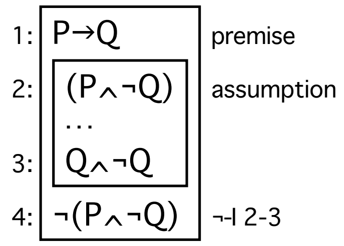
By now it should be clear: the contradiction is going to be Q∧¬Q, and the proof can be completed by a few steps of ∧ elimination and introduction.

It is possible to provide an argument to the ¬-I rule, which Jape uses instead of the unknown _B. If I had known what the contradiction was going to be at the beginning of the proof, I could have text-selected Q:



```
1: P→Q              premise
   ...
2: ¬(P∧¬Q)
```

and then applied ¬-I:



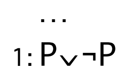
```
1: P→Q              premise
2:   (P∧¬Q)         assumption
     ...
3:   Q∧¬Q
4: ¬(P∧¬Q)          ¬-I 2-3
```

If you aren't exploring, using an argument keeps the proof tidier, which reduces confusion and the mistakes that confusion can cause. But I'm usually exploring when I use the ¬-I rule.

'Proof by contradiction': backward reasoning using ¬-E followed by ¬-I, then ∧-I and hyp

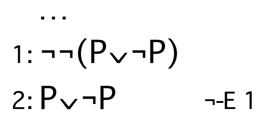
Sometimes you will be confronted with a problem which you can't solve using the rules just as simplification tools. Then you can – then you must! – fall back on a proof technique called 'proof by contradiction': if ¬*C* is ridiculous – if it leads to an absurd conclusion, to a contradiction – then you can say that *C* must hold. In the rules used in the ItL course, you have to use a slight variation on this strategy: if ¬*C* leads to a contradiction, then ¬¬*C* holds (by ¬-I), and then *C* holds (by ¬-E).

You normally realise that you must try proof by contradiction because nothing else will work. For example, you might have to prove



```
   ...
1: P∨¬P
```

Using ∨ introduction – throwing away either P or ¬P – gets you nowhere. The proof by contradiction technique works backwards. You introduce the doubly-negated form to see if it is easier to prove, by applying the ¬-E rule backwards:



```
   ...
1: ¬¬(P∨¬P)
2: P∨¬P            ¬-E 1
```

Line 1 might not seem to be easier to prove than line 2 – it looks more complicated – but using ¬-E has opened up a new world of possibilities because now the ¬-I rule has something to chew on. Line 1 is of the form ¬*something*, where *something* is ¬(P∨¬P). Apply ¬-I:

```
1: ┌¬(P∨¬P)    ┐  assumption
   │...        │
2: │_B∧¬_B     │
   └───────────┘
3: ¬¬(P∨¬P)       ¬-I 1-2
4: P∨¬P           ¬-E 3
```

Already the proof has the shape of a proof by contradiction. From the negation of the conclusion we want to prove, show that a contradiction follows. Conclude that the negation of the negation of the conclusion must hold, and thus the negation of the conclusion.

Now the last step in the boxed proof must surely have been ∧-I. Apply that rule

```
1: ┌¬(P∨¬P)    ┐  assumption
   │...        │
2: │_B         │
   │...        │
3: │¬_B        │
4: │_B∧¬_B     │  ∧-I 2,3
   └───────────┘
5: ¬¬(P∨¬P)       ¬-I 1-4
6: P∨¬P           ¬-E 5
```

Now you have to make a proof of _B and a proof of ¬_B. At this point you usually look at the assumptions you have available – and lo and behold, we actually have ¬*something* as an assumption. (That always happens if you apply ¬-E backwards and then ¬-I backwards, if you think about it!)

So at this point I make a guess: I guess that ¬_B is the same as ¬(P∨¬P). To make the proof follow the guess, I select the conclusion on line 3 and the assumption on line 1

```
1: ┌¬(P∨¬P)    ┐  assumption
   │...        │
2: │_B         │
   │...        │
3: │¬_B        │
4: │_B∧¬_B     │  ∧-I 2,3
   └───────────┘
5: ¬¬(P∨¬P)       ¬-I 1-4
6: P∨¬P           ¬-E 5
```

and unify them using hyp:

```
1: ┌¬(P∨¬P)           ┐  assumption
   │...               │
2: │P∨¬P              │
3: │P∨¬P∧¬(P∨¬P)      │  ∧-I 2,1
   └──────────────────┘
4: ¬¬(P∨¬P)              ¬-I 1-3
5: P∨¬P                  ¬-E 4
```

Magic! I had to prove P∨¬P from no assumptions: by using ¬-E and ¬-I, then ∧-I and hyp I have transformed that into the problem of proving P∨¬P assuming its own negation! I leave the rest of the proof to you, with the hint that now the assumptions have changed it is worthwhile trying ∨ introduction backwards.[1]

Those four steps – ¬-E backwards, ¬-I backwards, ∧-I backwards, finally hyp – are a very useful rule of thumb. If you are stuck, they sometimes help (but not always).

If you know what _B should be, you may be able to provide it as an argument to ¬-I, which makes things a bit tidier and misses out the hyp step. Usually, though, I like to leave the question open by allowing Jape to invent an unknown for me.

(The proof which I have commenced above is the ‘direct’ proof, using just the rules of the logic. There is a much nicer proof of P∨¬P, which uses the theorem ¬(P∨Q) ⊢ ¬P∧¬Q – it’s a few lines lower down in the Conjectures panel. Proof of the theorem itself is direct, and fairly easy. Proofs by contradiction are rarely straightforward, even though they may be called ‘direct’.)

1 This approach leads to what I think is the ‘standard’ proof of P∨¬P. I think that proof stinks, but then I don’t really believe in the ¬¬-E rule. There is a simpler proof which uses the theorem ¬(P∨Q)⊢¬P∧¬Q.

Appendix G: the ∀ rules

Jape can't use precisely the syntax of the ItL notes for quantification: it requires a dot (full stop, period) after the bound variable, and the body of the formula doesn't need to be bracketed. Otherwise the treatment is identical. You need to bracket the body of the formula if it isn't a predicate or a negation, so in Jape you write ∀x.(P(x)∧Q(x)), still with the extra dot.

Normally you will use the ∀ *elimination* rule to make a forward step from an assumption (or from a conclusion already proved by a forward step) and you will use the ∀ *introduction* rule to make a backward step from a conclusion. In either case the assumption or conclusion should be of the form ∀ *variable . something,* and in either case the effect is to *simplify* the assumption or conclusion you selected.

You usually give an argument to the elimination rule by text-selecting some variable which has been introduced by ∀ introduction or ∃ elimination; it isn't necessary or helpful to give an argument to the introduction rule.

The order you use the quantification rules is important. The ∀ introduction rule, like the ∃ elimination rule, introduces variables into the proof; the ∀ elimination rule and the ∃ introduction rule make use of those variables. So you usually use ∀ introduction, and the ∃ elimination, as early as possible. See the discussion of scope boxing above, if you haven't already read it.

Because the introduction rule, used backwards, introduces a variable into the proof and you have to use such a variable in the elimination rule, in this appendix we discuss the introduction rule first.

Backward reasoning using ∀-I

The ∀-I rule in box form is

$$\begin{array}{lll} i: & \boxed{\begin{array}{l}\text{var } c \\ \ldots \\ A(c)\end{array}} & \\ j: & & \ldots \\ & \ldots & \\ k: & \forall x.A(x) & \rightarrow -\forall - I\ i\text{–}j \end{array}$$

and in tree form

$$\dfrac{\begin{array}{c}[\text{var } c] \\ \vdots \\ A(c)\end{array}}{\forall x.A(x)}\ (\text{FRESH } c)\ \ \forall - I$$

where *A* is some predicate, and *c* is a 'fresh individual' – a name you haven't made any assumptions about outside the box or outside the tree which proves *A*(*c*). (The proviso that *c* should be fresh actually appears in the box form of the rule as well, but there isn't any space on the diagram to put it.)

The name *c* is arbitrary, and in general you don't care what name you use, provided it isn't one that pops up in an assumption. Jape will invent such a name for you – not an unknown, but a proper variable name. If you really want to use a name of your own choice, you can give the rule an argument and Jape will check that it doesn't appear free in any of the assumptions – if it does, then it won't let you apply the rule.

Because of the scope boxing, which restricts the area of the proof in which the variable *c* can be used, you normally ***use the ∀ introduction rule as early as possible in a proof***, so that the variable can be used in later steps. If you try to break the scope rules, Jape will certainly stop you.

For example, in

```
1: P(c)                 premise
   ...
2: ∀x.(P(x)→Q(x))
```

you can simply apply ∀-I, and Jape will introduce the name c1 – not c because it is already in use – without being prompted:

```
1: P(c)                 premise
2:   var c1             assumption
     ...
3:   (P(c1)→Q(c1))
4: ∀x.(P(x)→Q(x))       ∀-I 2-3
```

It looks as if Jape has made a bad choice: if it had chosen c instead, you would have been able to go ahead and complete the proof. But it won't work: the name introduced in the scope box has to be new to the proof, or at least it mustn't appear in the assumptions which apply in the ∀-I step. If you backtrack, text-select c in the top line

```
1: P(c)                 premise
   ...
2: ∀x.(P(x)→Q(x))
```

and then try to apply the ∀-I rule, Jape won't let you. It says that the ∀-I rule is not applicable, because a proviso has been violated: the argument you have chosen *does* appear free in the assumption on line 1, and so you can't use the rule with that argument. For that reason, you can't prove this conjecture in the Natural Deduction system.

A trick with bound variables and scope boxing

Jape's scope box restriction, and its FRESH proviso, apply to *free occurrences* of the variable *c*. You can trick it, sometimes, by supplying an argument which is the bound variable itself. \For example, in proving

```
1: ∃x.∀y.P(x,y)   premise
   ...
2: ∀y.∃x.P(x,y)
```

you can text-select y

```
1: ∃x.∀y.P(x,y)   premise
   ...
2: ∀y.∃x.P(x,y)
```

and then apply ∀-I:

```
1: ∃x.∀y.P(x,y)   premise
2:   var y        assumption
     ...
3:   ∃x.P(x,y)
4: ∀y.∃x.P(x,y)   ∀-I 2-3
```

Variable y does appear in the premise – twice – but it doesn't appear *free* – one is a binding occurrence and the other is a bound occurrence.

The trick is little more than an interesting curiosity – it's just as easy to prove the theorem without it.

Forward reasoning using ∀-E

Be careful when you use this rule that you have already done all the ∀ introductions that you want to: otherwise your proof can be blocked.

The ∀-E rule in box form is

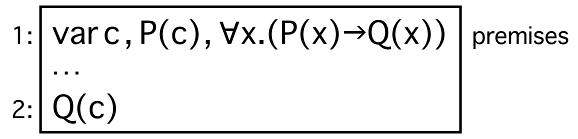
$$\begin{array}{lll} i: & \forall x.A(x) & \ldots \\ & \ldots & \\ j: & \mathit{inscope}\ c & \ldots \\ & \ldots & \\ k: & A(c) & \forall - E\ i,j \end{array}$$

and in tree form

$$\dfrac{\begin{array}{cc} \vdots & \vdots \\ \forall x.A(x) & \text{inscope } c \end{array}}{A(c)}\ \forall - E$$

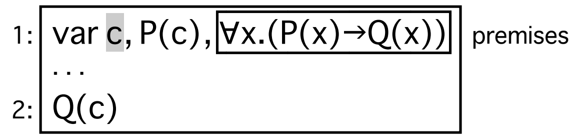
From a proof of $\forall x.Ax$), make a proof of $A(c)$, where A is some predicate and c is a variable which is in scope – that is, var c must appear in the assumption line of a box enclosing lines j to k. (In the tree version, var c must appear in the hypotheses used to prove $A(c)$.) The inscope line is a 'side condition', which is hidden if it is proved and displayed if it is not. If you use the rule as recommended, you won't see the side condition.

Forward reasoning with the rule is straightforward: choose an assumption (or a conclusion proved on a forward step) which is a universal quantification, ***give an argument by text-selection***, and apply the ∀-E rule. For example, in



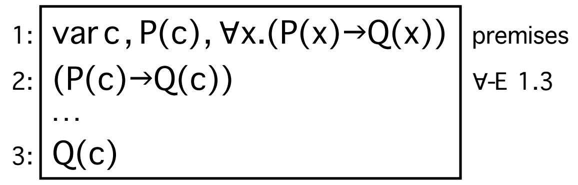
```
1: var c, P(c), ∀x.(P(x)→Q(x))   premises
   ...
2: Q(c)
```

select the third premise, text-select one of the *c*s:



```
1: var c, P(c), ∀x.(P(x)→Q(x))   premises
   ...
2: Q(c)
```

and apply ∀-E:



```
1: var c, P(c), ∀x.(P(x)→Q(x))   premises
2: (P(c)→Q(c))                   ∀-E 1.3
   ...
3: Q(c)
```

If you make the step without supplying an argument, Jape will use an unknown and show you the side condition of the rule:

```
1: var c, P(c), ∀x.(P(x)→Q(x))   premises
   ...
2: _c1 inscope
3: (P(_c1)→Q(_c1))               ∀-E 1.3,2
   ...
4: Q(c)
```

It’s pretty obvious that in this proof that the unknown `_c1` should be `c`, both because that is the only variable available and because `Q(_c1)` will eventually have to unify with `Q(c)`. In this case, then, the problem will resolve itself eventually and the side condition will disappear, but I recommend that you always provide an argument when applying the rule, just to avoid ugly and confusing displays like this one. Perhaps Jape should insist that you supply an argument ...

Backward reasoning with ∀-E

It is possible to use the ∀-E rule backwards to prove an arbitrary conclusion. If you provide the argument *B* then the result may be comprehensible. If not, maybe not, and arbitrary amounts of nonsense revealing Jape’s internal workings may appear in the session window. I give no examples here.

Appendix H: the ∃ rules

Jape can't use precisely the syntax of the ItL notes for quantification: it requires a dot (full stop, period) after the bound variable. Otherwise the treatment is identical. You need to bracket the body of the formula if it isn't a predicate or a negation, so in Jape you write ∃x.(P(x)∧Q(x)), still with the extra dot.

Normally you will use the ∃ *elimination* rule to make a forward step from an assumption (or from a conclusion already proved by a forward step) and you will use the ∃ *introduction* rule to make a backward step from a conclusion. In either case the assumption or conclusion should be of the form ∃ *variable* . *something,* and in either case the effect is to *simplify* the assumption or conclusion you selected.

You usually give an argument to the introduction rule by text-selecting some variable which has been introduced by ∃ elimination or ∀ introduction; it isn't necessary or helpful to give an argument to the elimination rule.

The order you use the quantification rules is important. The ∀ introduction rule, like the ∃ elimination rule, introduces variables into the proof; the ∀ elimination rule and the ∃ introduction rule make use of those variables. So you usually use ∀ introduction, and the ∃ elimination, as early as possible. See the discussion of scope boxing above, if you haven't already read it.

Forward reasoning using ∃-E

The ∃-E rule is in box form

$$
\begin{array}{rll}
i: & \exists x.A(x) & \ldots \\
 & \ldots & \\
j: & \boxed{\begin{array}{l} \text{var } c, A(c) \\ \ldots \\ B \end{array}} & \begin{array}{l} \text{assumptions} \\ \\ \ldots \end{array} \\
k: & & \\
 & \ldots & \\
l: & B & \exists - E\ i, j..k
\end{array}
$$

and in tree form

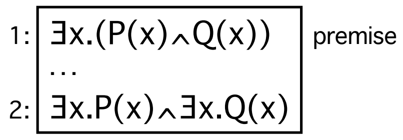
$$
\dfrac{\begin{array}{c} \\ \vdots \\ \exists x.F(x) \end{array} \qquad \begin{array}{c} [\text{var } c, F(c)] \\ \vdots \\ A \end{array}}{A}\ (\text{FRESH } c)\ \ \exists - E
$$

where A is some predicate, and c is a 'fresh individual' – a name you haven't made any assumptions about outside the box or outside the tree which proves $A(c)$. (The proviso that c should be fresh actually appears in the box form of the rule as well, but there isn't any space on the diagram to put it.)

The name c is arbitrary, and in general you don't care what name you use, provided it isn't one that pops up in an assumption. Jape will invent such a name for you – not an unknown, but a proper variable name. If you really want to use a name of your own choice, you can give the rule an argument and Jape will check that it doesn't appear free in any of the assumptions – if it does, then it won't let you apply the rule.

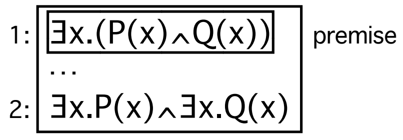
Because of the scope boxing, which restricts the area of the proof in which the variable c can be used, you normally ***use the ∃ introduction rule as early as possible in a proof***, so that the variable can be used in later steps. If you try to break the scope rules, Jape will certainly stop you.

For example, in



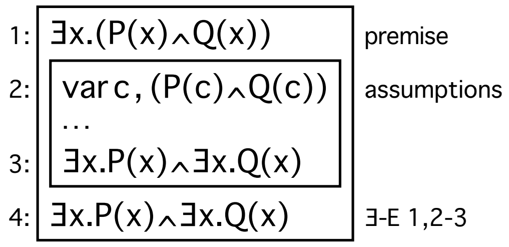
```
1: ∃x.(P(x)∧Q(x))     premise
   ...
2: ∃x.P(x)∧∃x.Q(x)
```

select the premise



```
1: ∃x.(P(x)∧Q(x))     premise
   ...
2: ∃x.P(x)∧∃x.Q(x)
```

and apply ∃-E:



```
1: ∃x.(P(x)∧Q(x))         premise
2:   var c, (P(c)∧Q(c))   assumptions
     ...
3:   ∃x.P(x)∧∃x.Q(x)
4: ∃x.P(x)∧∃x.Q(x)        ∃-E 1,2-3
```

Jape introduces the name c, substitutes it appropriately into the new assumptions, and shows the structure of the new proof with a box.

It isn't usually useful to give an argument to the rule, though occasionally it can produce results (see 'A trick with bound variables and scope boxing' above).

Backward reasoning with ∃-I

Once you have introduced all the variables you need, it is safe to use ∃ introduction backwards (and, likewise, use ∀ elimination forwards). You normally provide an argument to say just what instance you want to use: otherwise Jape will introduce an unknown and show you a nasty side condition.

The rule, where *A* is a predicate and *c* is a variable, is in box form

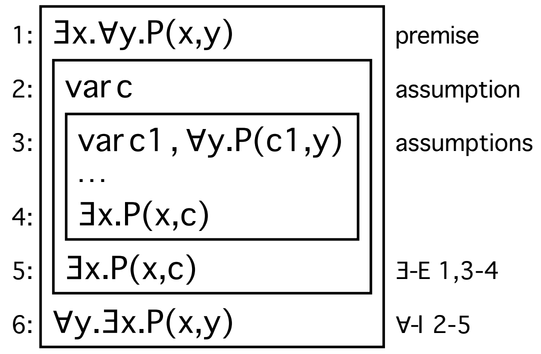
$$\begin{array}{lll} i: & A(c) & \ldots \\ & \ldots & \\ j: & \textit{inscope } c & \ldots \\ & \ldots & \\ k: & \exists x.A(x) & \exists\!-\!I\ i,j \end{array}$$

and in tree form

$$\frac{\begin{array}{cc} \vdots & \vdots \\ A(B) & \text{inscope } B \end{array}}{\exists x.A(x)}\ \exists - I$$

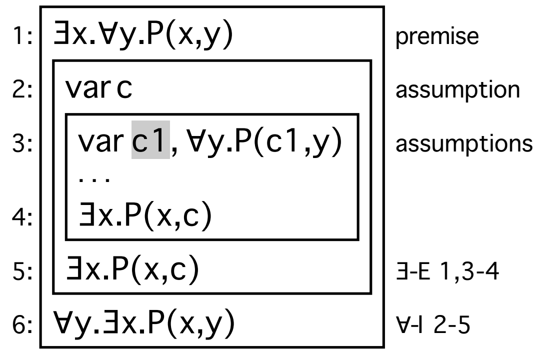
From a proof of $A(c)$ for some formula c you can make a proof of $\exists x.A(x)$, provided that c is in scope – that is, there is a var c in the assumption line of a box enclosing lines j to k. (In the tree version, var c must appear in the hypotheses available to prove $\exists x.A(x)$.) The inscope line (or tree) is a 'side condition', which is hidden if it is proved and displayed if it is not. If you use the rule as recommended, you won't see the side condition.

Forward reasoning with the rule is straightforward: choose a unproved conclusion which is an existential quantification, ***give an argument by text-selection***, and apply the ∀-E rule. For example, in



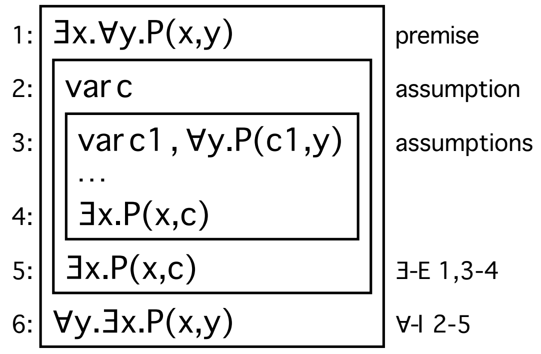
```
1: ∃x.∀y.P(x,y)              premise
2:   var c                    assumption
3:     var c1 , ∀y.P(c1,y)    assumptions
       ...
4:     ∃x.P(x,c)
5:   ∃x.P(x,c)                ∃-E 1,3-4
6: ∀y.∃x.P(x,y)              ∀-I 2-5
```

text-select *c1*



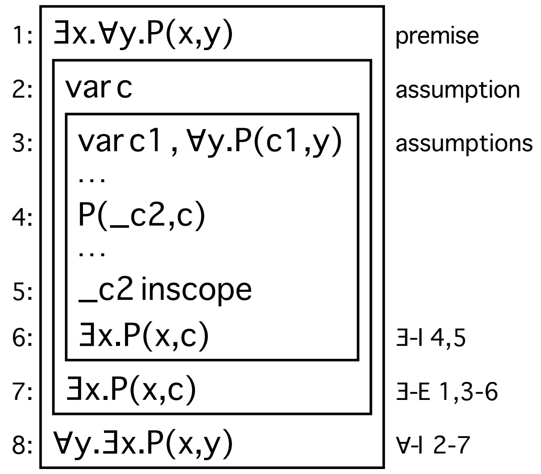
```
1: ∃x.∀y.P(x,y)              premise
2:   var c                    assumption
3:     var c1, ∀y.P(c1,y)     assumptions
       ...
4:     ∃x.P(x,c)
5:   ∃x.P(x,c)                ∃-E 1,3-4
6: ∀y.∃x.P(x,y)              ∀-I 2-5
```

and apply ∃-I:



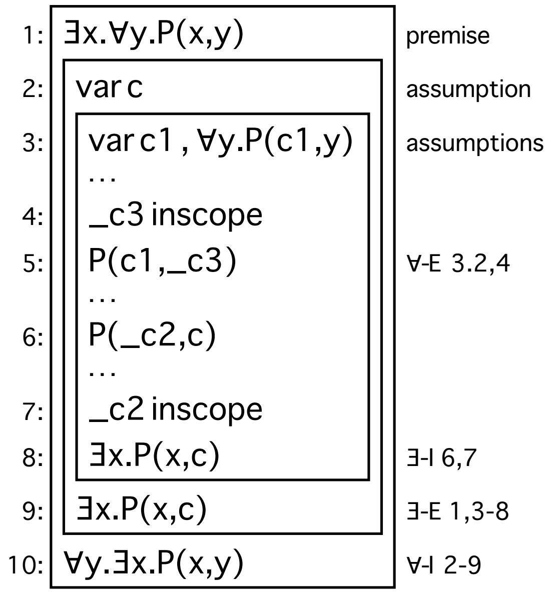
```
1: ∃x.∀y.P(x,y)              premise
2:   var c                    assumption
3:     var c1 , ∀y.P(c1,y)    assumptions
       ...
4:     ∃x.P(x,c)
5:   ∃x.P(x,c)                ∃-E 1,3-4
6: ∀y.∃x.P(x,y)              ∀-I 2-5
```

If you can't see which variable to choose, just apply the rule without an argument. Jape will use an unknown and show you the side condition:



```
1: ∃x.∀y.P(x,y)              premise
2:   var c                    assumption
3:     var c1 , ∀y.P(c1,y)    assumptions
       ...
4:     P(_c2,c)
       ...
5:     _c2 inscope
6:     ∃x.P(x,c)              ∃-I 4,5
7:   ∃x.P(x,c)                ∃-E 1,3-6
8: ∀y.∃x.P(x,y)              ∀-I 2-7
```

It’s not too bad to see that side condition if you really don’t know where the proof is heading. The next step of ∀-E will introduce another unknown and side condition, if you don’t give it an argument (you first have to select line 4 as well as the second assumption on line 3):



```
 1: ∃x.∀y.P(x,y)                premise
 2:   var c                     assumption
 3:     var c1 , ∀y.P(c1,y)     assumptions
        ...
 4:     _c3 inscope
 5:     P(c1,_c3)               ∀-E 3.2,4
        ...
 6:     P(_c2,c)
        ...
 7:     _c2 inscope
 8:     ∃x.P(x,c)               ∃-I 6,7
 9:   ∃x.P(x,c)                 ∃-E 1,3-8
10: ∀y.∃x.P(x,y)                ∀-I 2-9
```

When you unify lines 5 and 6, either with hyp or with Unify, the side conditions are satisfied and the proof is complete.

Backward reasoning with ∃-E

It is possible to use the ∃-E rule backwards to prove an arbitrary conclusion, but it’s very tricky and I wouldn’t bother if I were you.

Appendix I: Internal secrets of Jape

Most users of ItL Jape won’t want to read this appendix. It’s included for those who are curious about the internal workings of Jape, and both knowledgeable and strong-minded enough to take the truth.

There’s an extensive manual of over 100 pages available from the Jape ftp site (ftp://ftp.dcs.qmw.ac.uk/programming/jape/index.html). This appendix gives just a taste, and it only covers those matters that may actually affect your use of Jape..

Forward reasoning and trees

In box-and-line proofs a step on line *i* can make use of a step on any previous line provided that it isn’t inside a box which line *i* is outside. Jape actually works with a tree, and makes ingenious use of the *cut* rule of the sequent calculus to imitate forward reasoning.

When a rule has two antecedents, as for example the ∧-I rule does

$$\frac{\begin{array}{cc}\vdots & \vdots \\ A & B\end{array}}{A \wedge B}\ \wedge\!-\!I$$

Jape’s trickery is exposed. In the box form of the proof

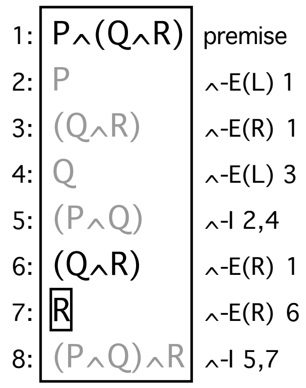
$$\begin{array}{lll} i: & A & \ldots \\ & \ldots & \\ j: & B & \ldots \\ & \ldots & \\ k: & A \wedge B & \wedge - I\ i,j \end{array}$$

the lines which are generated as part of the proof of *A* – that is, the lines of the first antecedent tree – can’t be used in the proof of *B*.

For example, in

1:	P∧(Q∧R)	premise
2:	P	∧-E(L) 1
3:	(Q∧R)	∧-E(R) 1
4:	Q	∧-E(L) 3
5:	(P∧Q)	∧-I 2,4
6:	(Q∧R)	∧-E(R) 1
7:	R	∧-E(R) 6
8:	(P∧Q)∧R	∧-I 5,7

it is impossible to use line 3 to prove line 7. You can see that this is so if you select line 7 in that proof:



```
1: P∧(Q∧R)     premise
2: P           ∧-E(L) 1
3: (Q∧R)       ∧-E(R) 1
4: Q           ∧-E(L) 3
5: (P∧Q)       ∧-I 2,4
6: (Q∧R)       ∧-E(R) 1
7: R           ∧-E(R) 6
8: (P∧Q)∧R     ∧-I 5,7
```

This is a real defect, not just a simple bug. Accessibility of lines depends on the way that the proof was made, the order that its lines are developed. One day I hope to make a Jape which is better at imitating box-and-line proofs like the ones that you make, and then I can delete this apology from the manual.

Predicate notation and substitution

Internally, Jape doesn't really understand predicate notation like $P(c)$ or $Q(x,y)$. It uses instead a 'substitution notation' which is somewhat more general and more powerful. Most of the time you won't see the difference, but very occasionally it may surface in surprising ways.

For example, Jape translates the ∀-I rule into something like this:

$$\begin{array}{lll} i: & \text{var } c & \\ & \ldots & \\ j: & A[x\backslash c] & \ldots \\ & \ldots & \\ k: & \forall x.A & \rightarrow -\forall - I\ i\text{–}j \end{array}$$

– from a proof of formula A in which every free occurrence of x is replaced by c, conclude $\forall x.A$ (with a proviso that c is a fresh individual). If formula A is actually something like $(P(x) \rightarrow Q(x))$, in which everything is either a predicate or the bound variable, then the substitution $A[x\backslash c]$ becomes $(P(c) \rightarrow Q(c))$ and the deception is perfect. If it is something like $(P \rightarrow Q(x))$, in which the name P isn't a predicate, then the result may be $(P[x\backslash c] \rightarrow Q(c))$ and the secret has leaked out.

Provisos

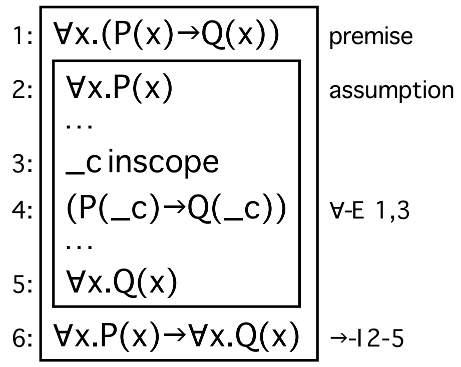
The rules that Jape uses internally are all tree rules, not box-and-line rules. It also uses provisos to stop rules being applied incorrectly. The proviso that it makes most use of is 'x NOTIN A', which means 'variable x doesn't appear free in formula A'. That proviso is automatically generated by the FRESH proviso which is included, for example, in the ∀-I rule. That rule *really* reads, once Jape has translated it from the natural deduction form:

$$\frac{\Gamma, \text{var } c \quad A[x\backslash c]}{\Gamma \quad \forall x.A} \text{(FRESH } c\text{) } \forall - I$$

The symbol Γ matches all the assumptions in force at the line where the rule is applied. Suppose those assumptions are $H_1, H_2, \ldots, H_n$: then the proviso FRESH c translates into the provisos c NOTIN H_1, c NOTIN H_2, ..., c NOTIN H_n, c NOTIN $\forall x.A$. Most of those provisos are obviously satisfied, and Jape throws them away. Occasionally there is one which isn't.

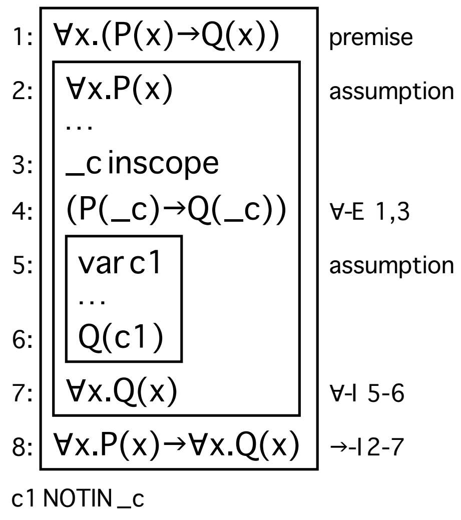
If a proviso is violated, Jape won't make the proof step – that's what provisos are for, to stop invalid steps. If the status of a proviso is unclear, neither obviously satisified nor clearly violated, Jape writes it into the proviso pane and keeps an eye on it in future proof steps. That most often happens when there is a proviso which includes an unknown, and it happened in an example above when rules were applied in the wrong order.

In that example an application of ∀-E introduced an unknown into the proof:



```
1: ∀x.(P(x)→Q(x))          premise
2:   ∀x.P(x)               assumption
     ...
3:   _c inscope
4:   (P(_c)→Q(_c))         ∀-E 1,3
     ...
5:   ∀x.Q(x)
6: ∀x.P(x)→∀x.Q(x)         →-I 2-5
```

Then an application of ∀-I went through, but it left a proviso (shown here at the bottom left of the illustration):



```
1: ∀x.(P(x)→Q(x))          premise
2:   ∀x.P(x)               assumption
     ...
3:   _c inscope
4:   (P(_c)→Q(_c))         ∀-E 1,3
5:     var c1              assumption
       ...
6:     Q(c1)
7:   ∀x.Q(x)               ∀-I 5-6
8: ∀x.P(x)→∀x.Q(x)         →-I 2-7
c1 NOTIN _c
```

That proviso arises because line 4 can be used as an assumption during a proof of line 6. It’s outside the scope box, so it’s necessary to apply a condition that whatever variable replaces *_c1*, it mustn’t be *c*. That stops you, later on in the proof, from unifying *_c1* and *c*, which is exactly as it should be no matter how frustrating it might feel at the time.

Jape uses invisible NOTIN provisos to help it when pretending to use predicate notation – but that’s its own private business and won’t affect what you do.