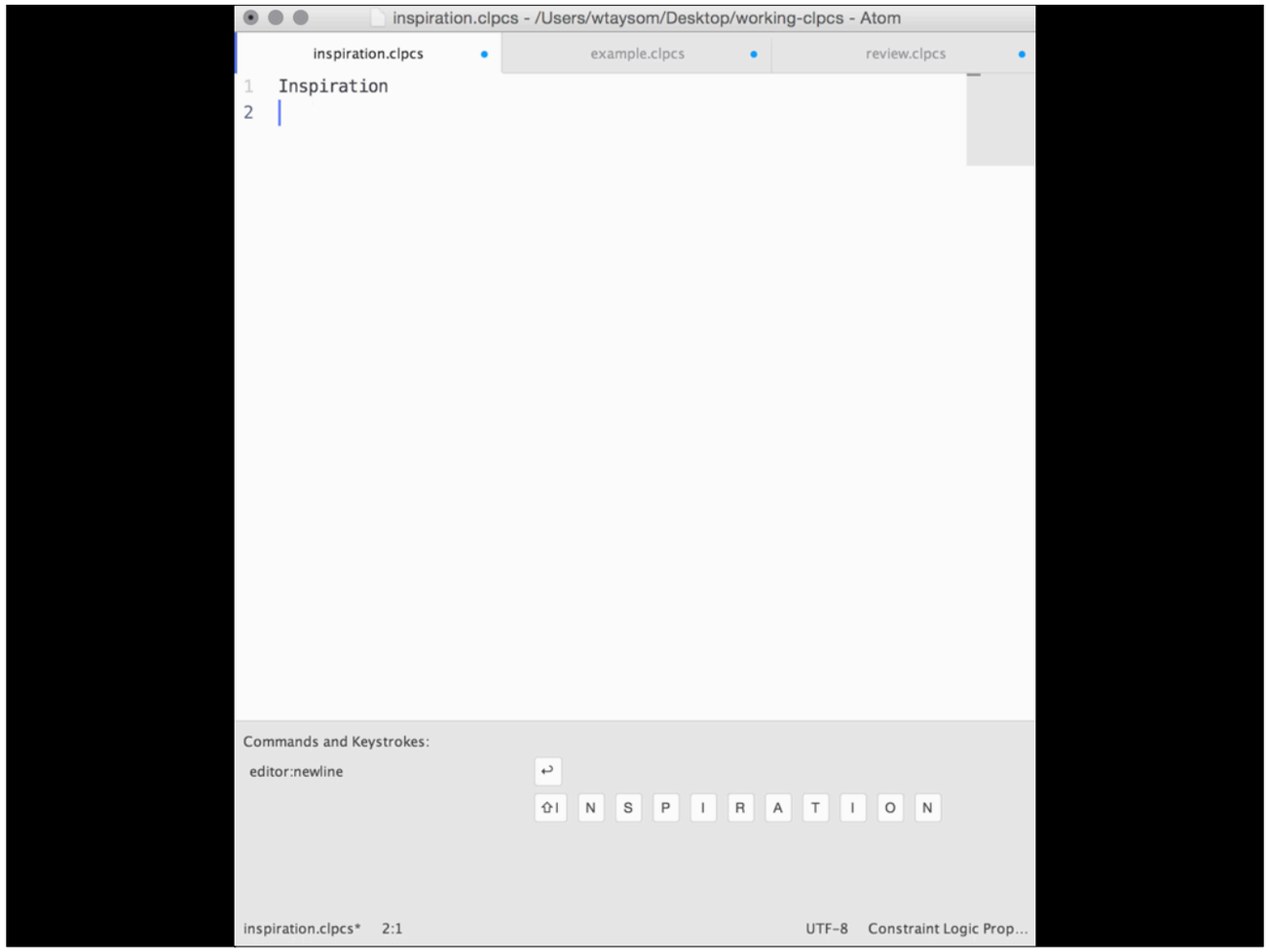
Constraint Logic Propagation Conflict Spreadsheets

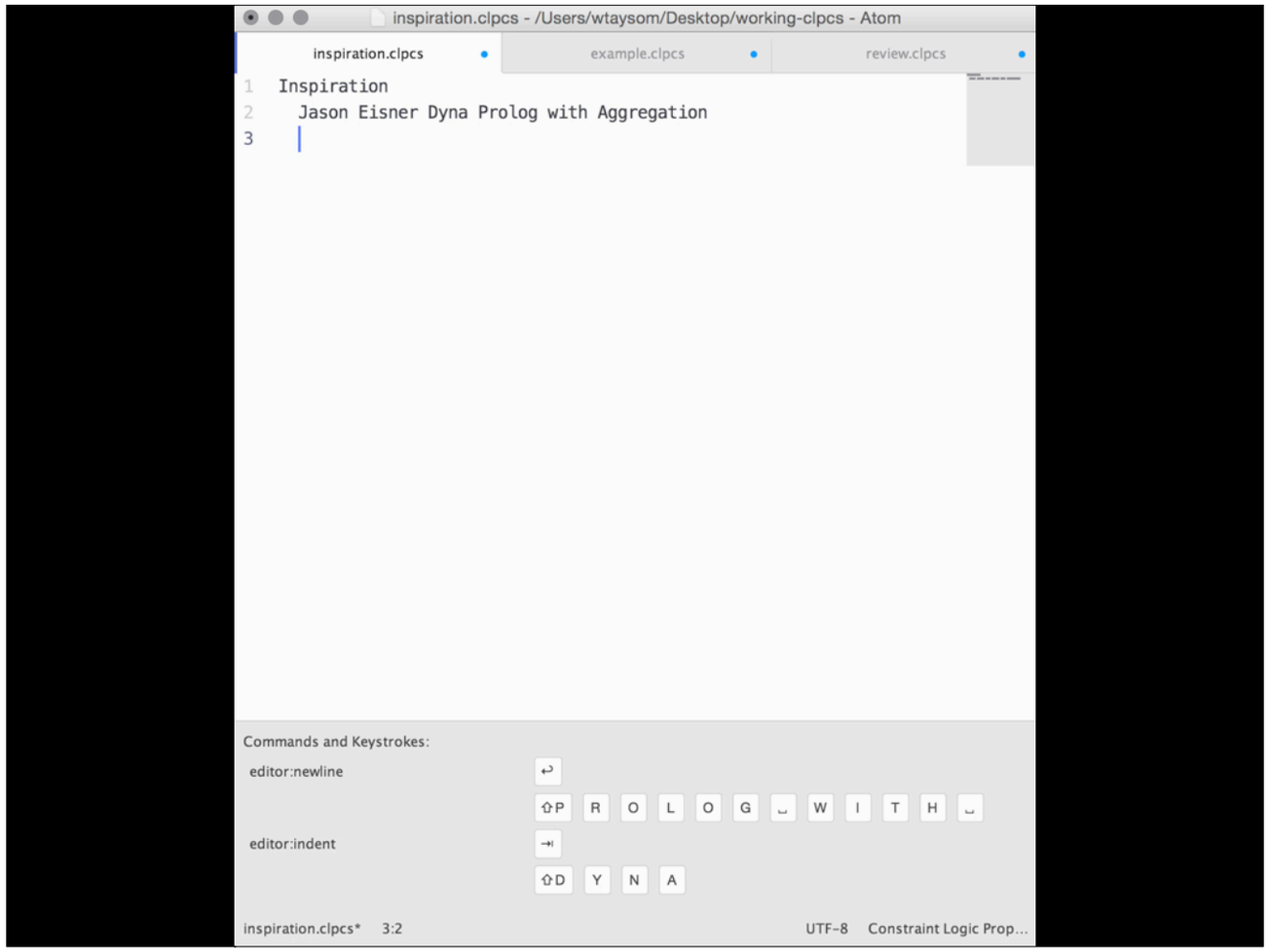
William Taysom

2015

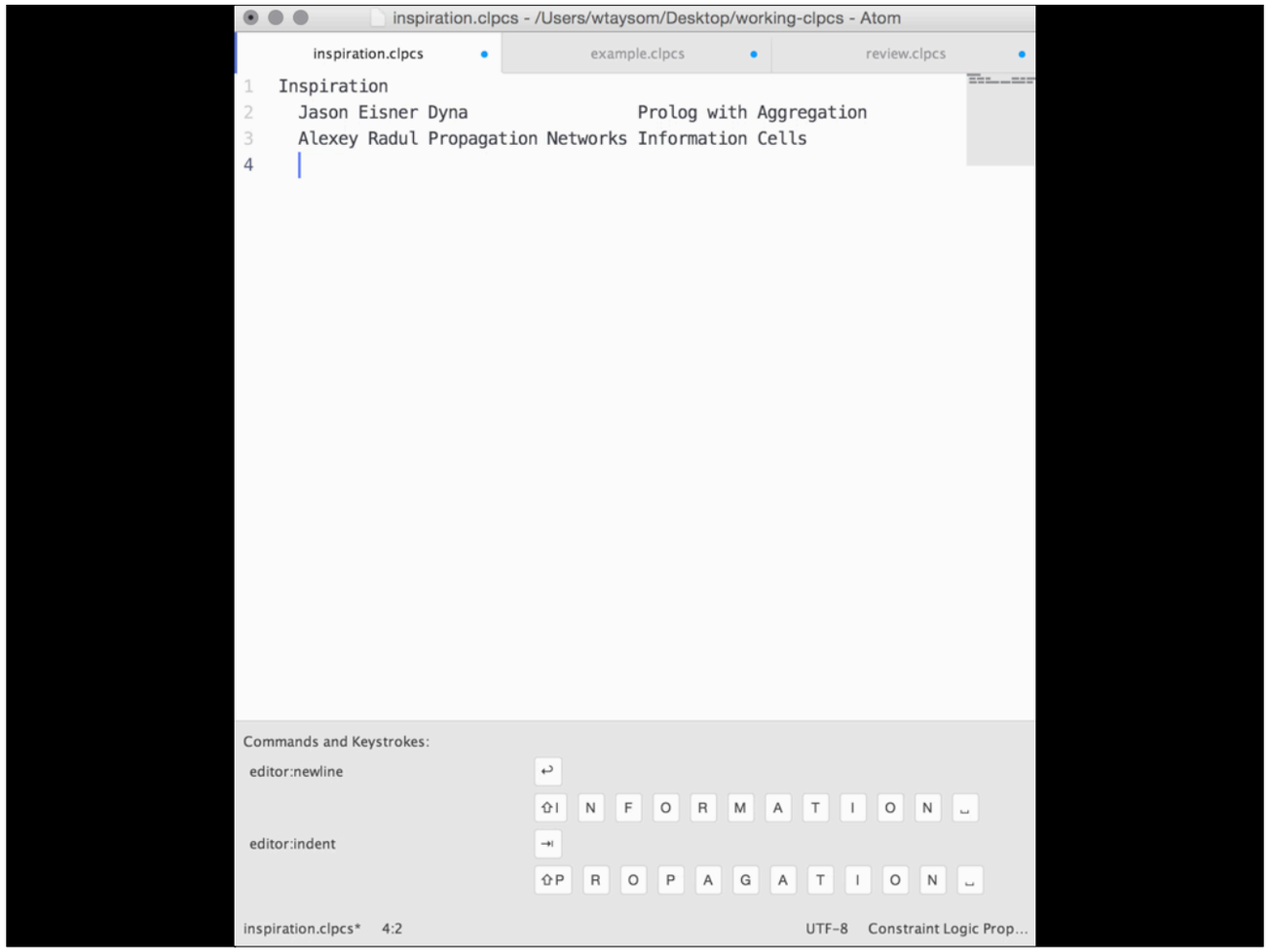
…an unruly name for an unruly topic.



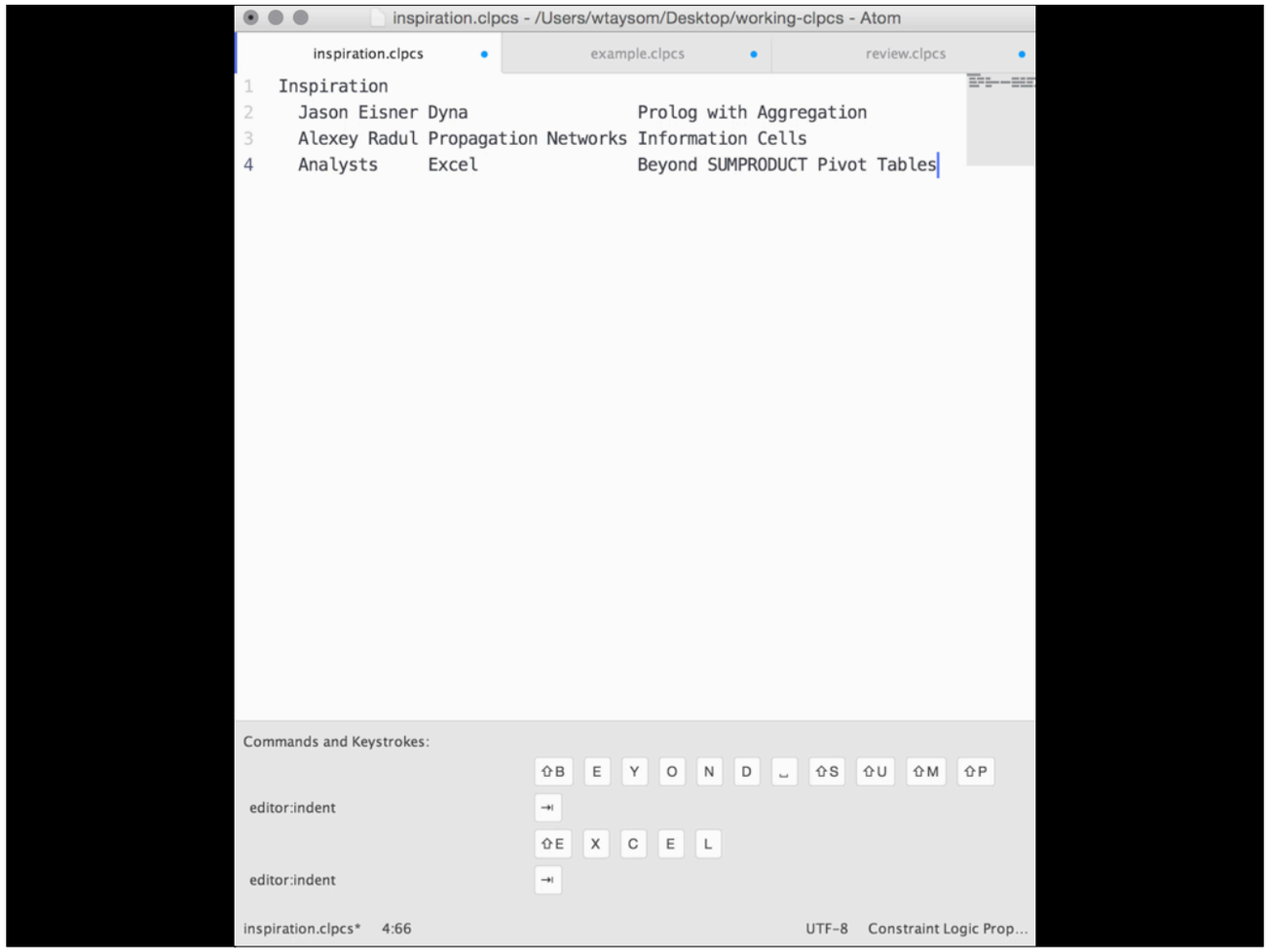
Good afternoon. Today is the future. Let me share a programming concept inspired by the confluence of three things:



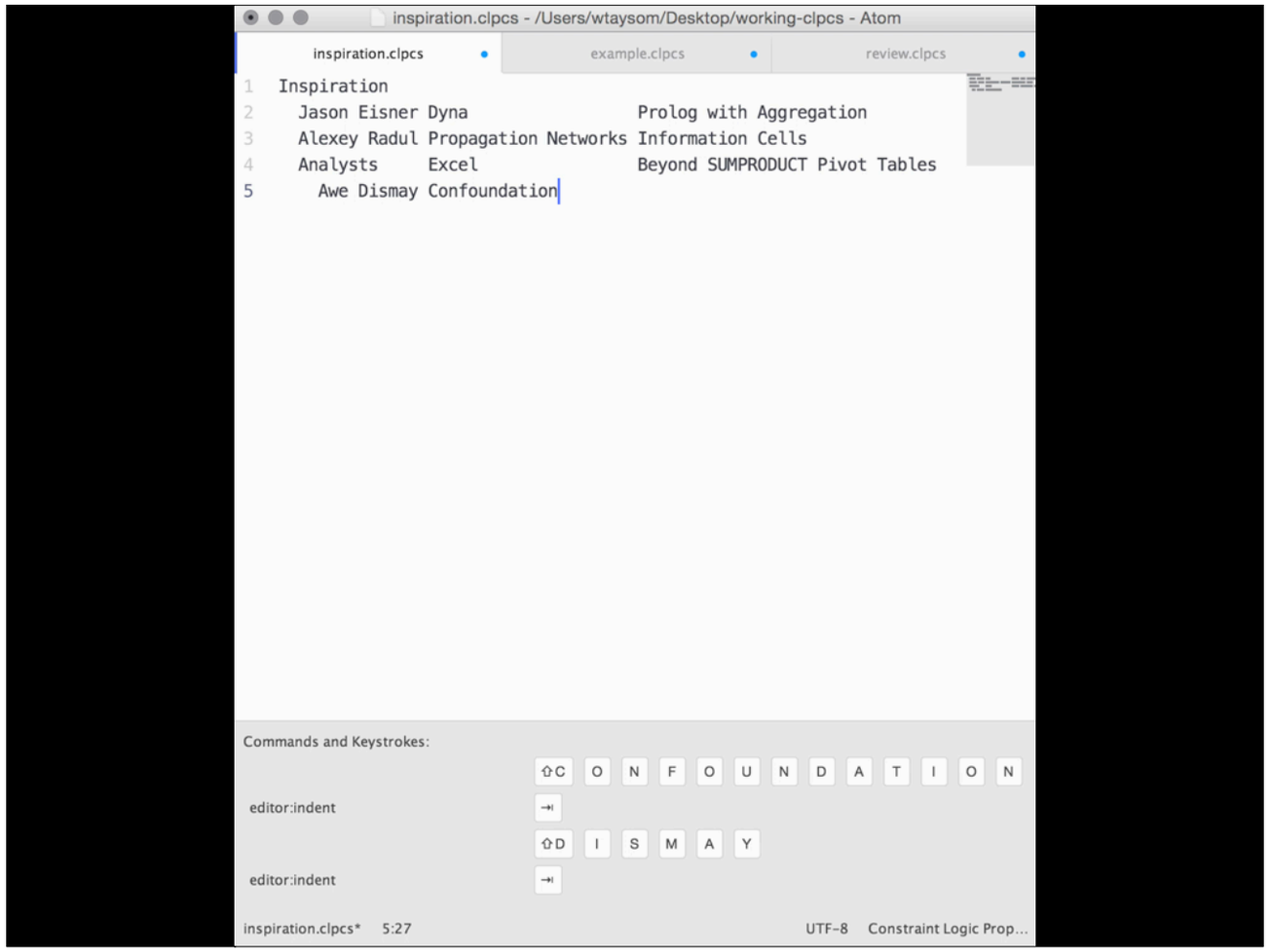
Jason Eisner's Dyna programming language, a sort of Prolog with aggregation operators,



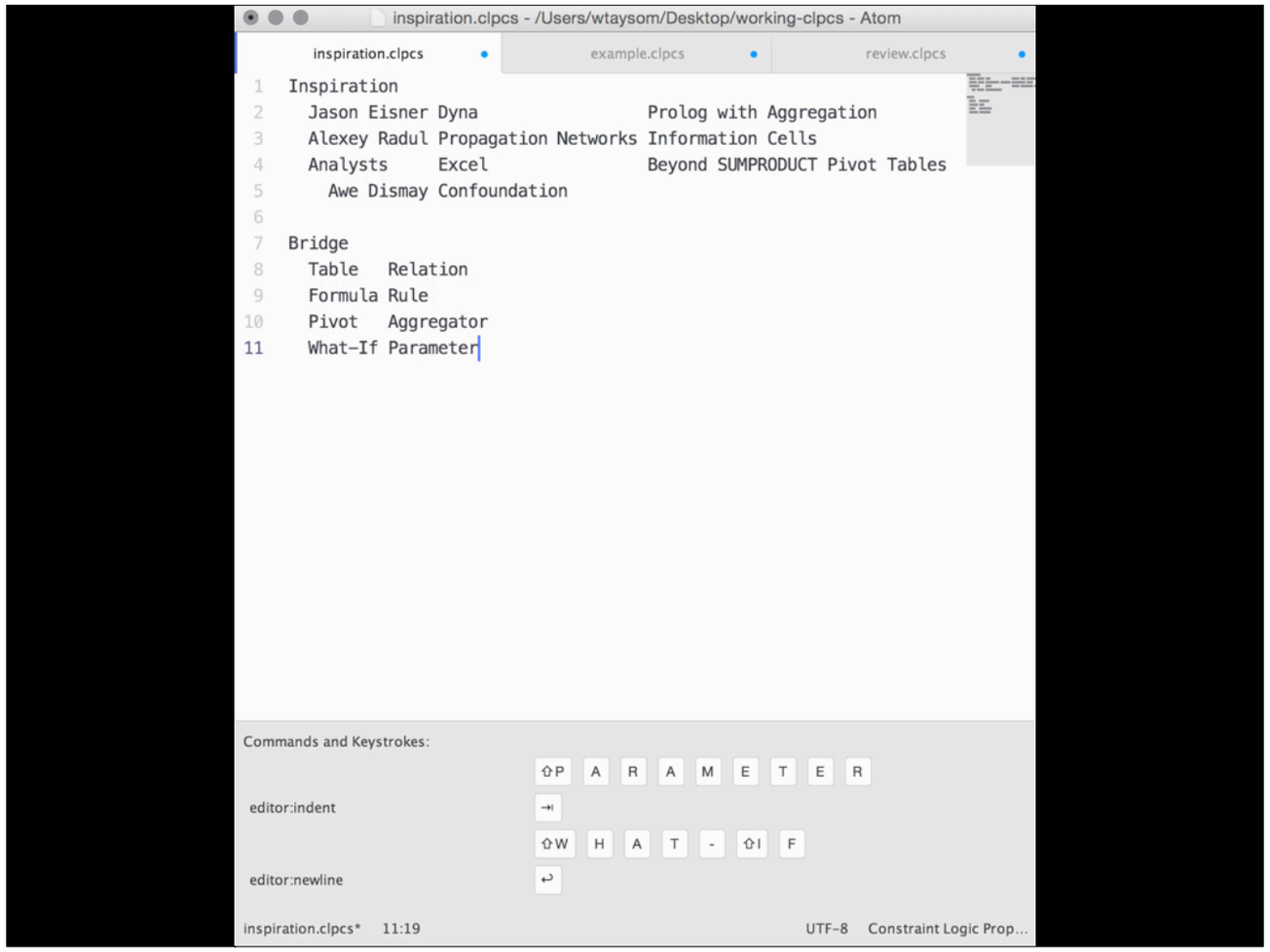
Alexey Radul's Propagation Networks in which reactive cells store information about values rather than just values,



and, most especially, work with analysts using Excel in realtime during multibillion dollar spectrum auctions. Imagine a secure, windowless room in a national capital, locked in with consultants, economists, analysts. No external devices, no internet access except for the auction software. The job is to help them make sense of what they're seeing, to live code beyond the SUMPRODUCTs and Pivot Tables at which Excel excels.



On these excursions, I'm awed by how efficiently analysts munge their data, dismayed by the hacks they go through to maintain their spreadsheets, and confounded that despite all my programmer prowess, they come up with their numbers faster. All the same, after a week, I have scripts to tabulate interesting figures, and I spend most of the next month stuck in the room ruminating over what it all means. This idea is the outcome of my most recent engagement: what can we do to enhance the data-centric reactivity of spreadsheets?



Constraint logic programming provides a natural bridge. Tables correspond to relations; formulas to rules; pivot tables to aggregators; what-if analysis to parameterization.
That's all cute, but then I had an idea worth sharing:

Treat Prolog

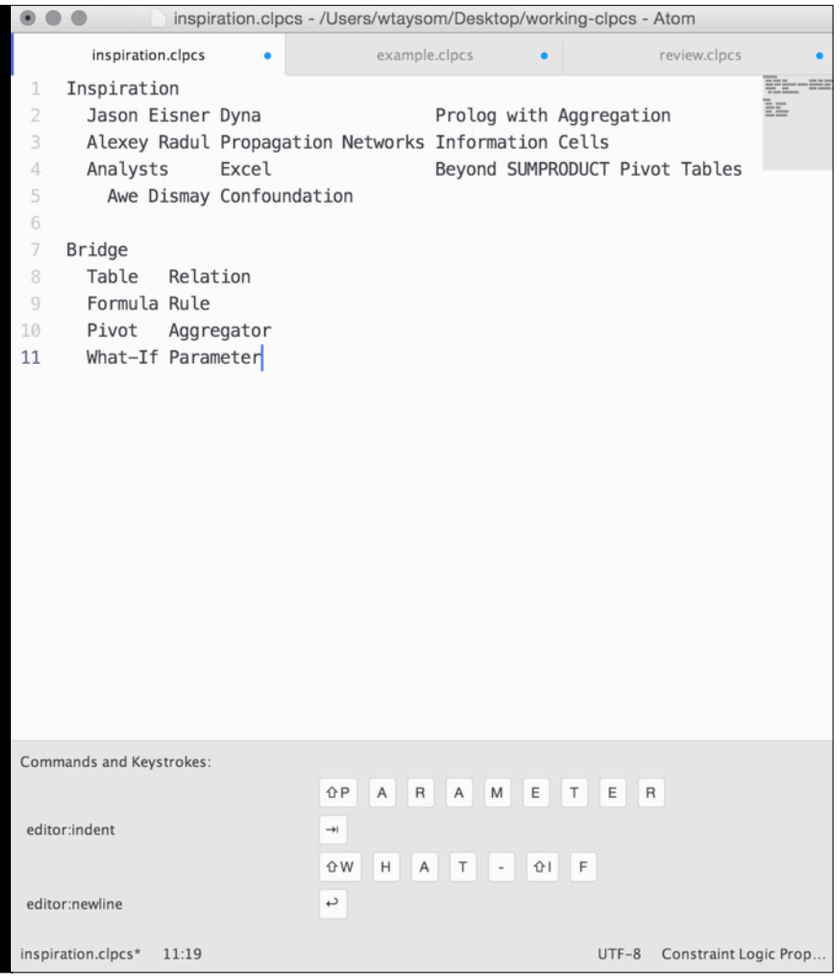
facts and relations

as

nodes and edges

in a

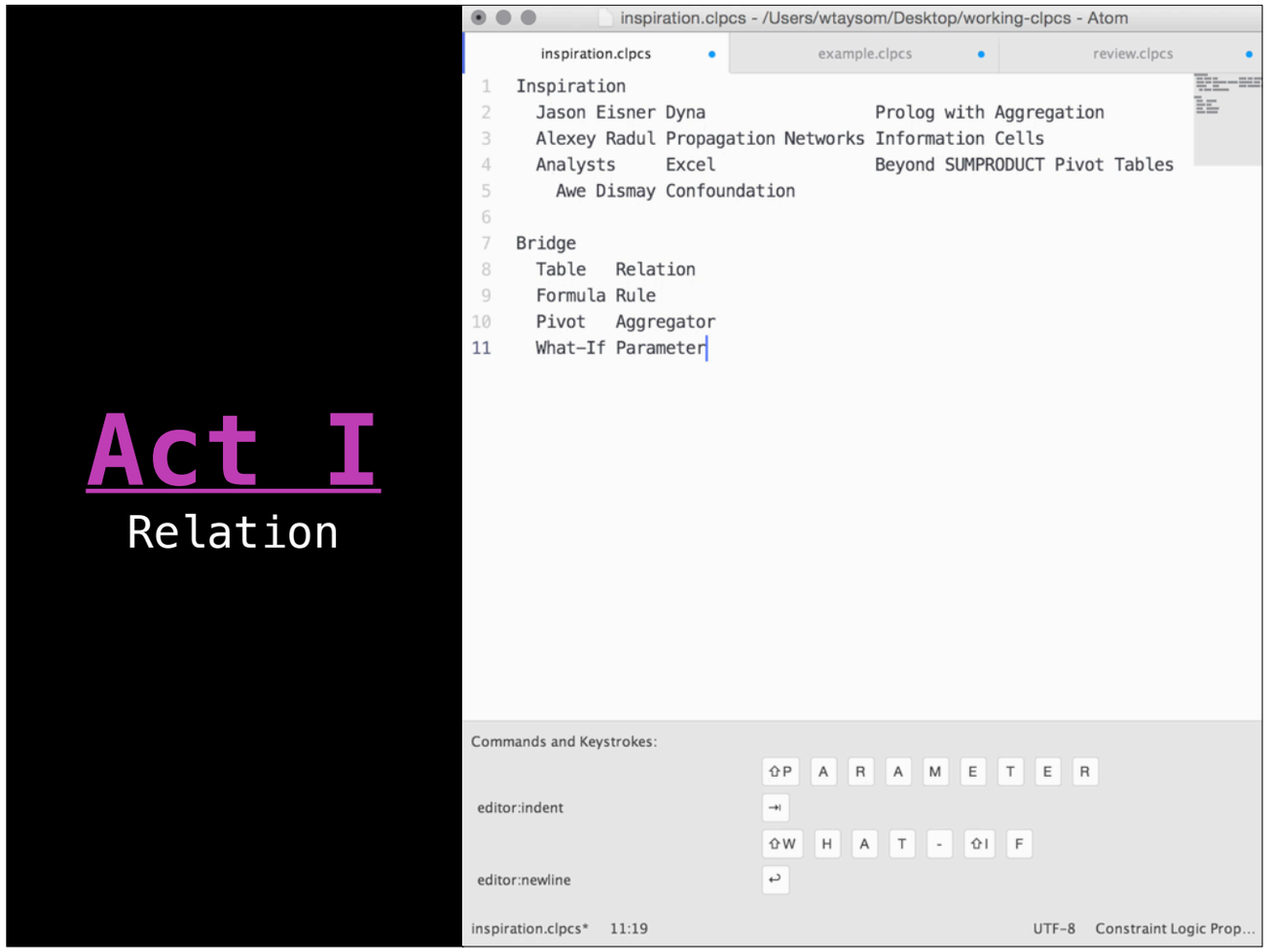
propagation network.



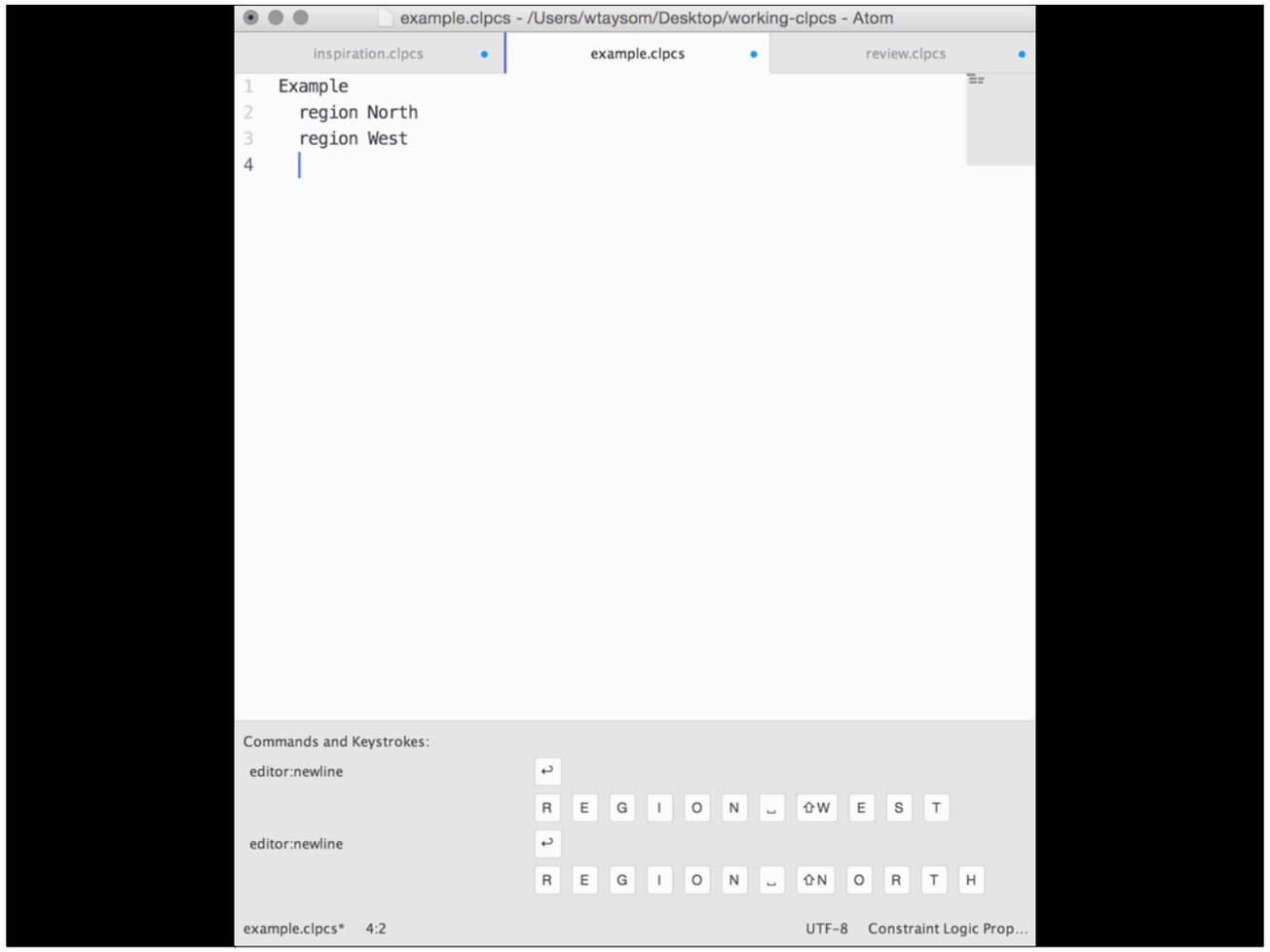
What happens when we treat prolog facts and relations as nodes and edges in a propagation network? By way of example, let's see how this might play out.

Act I

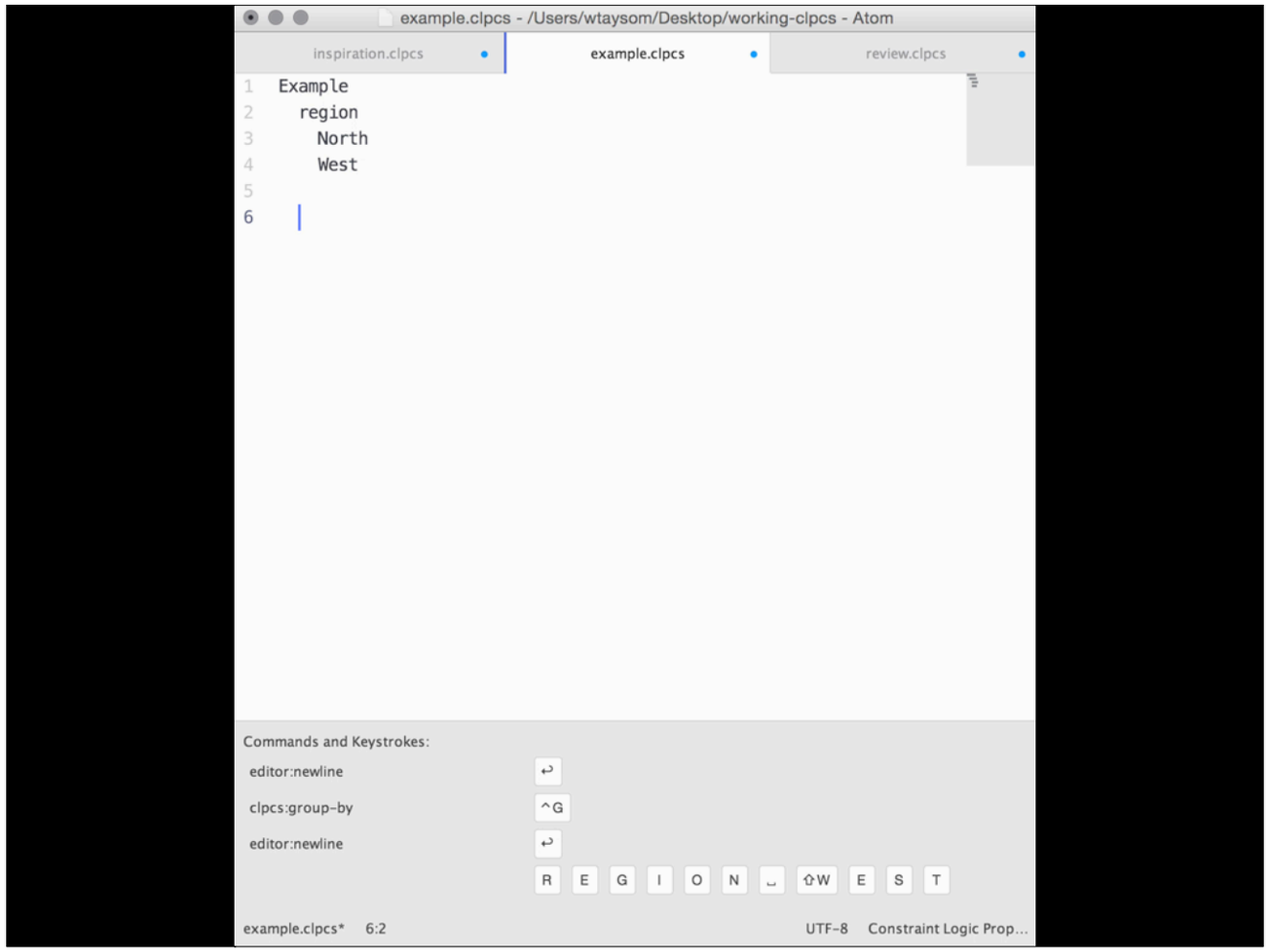
Relation



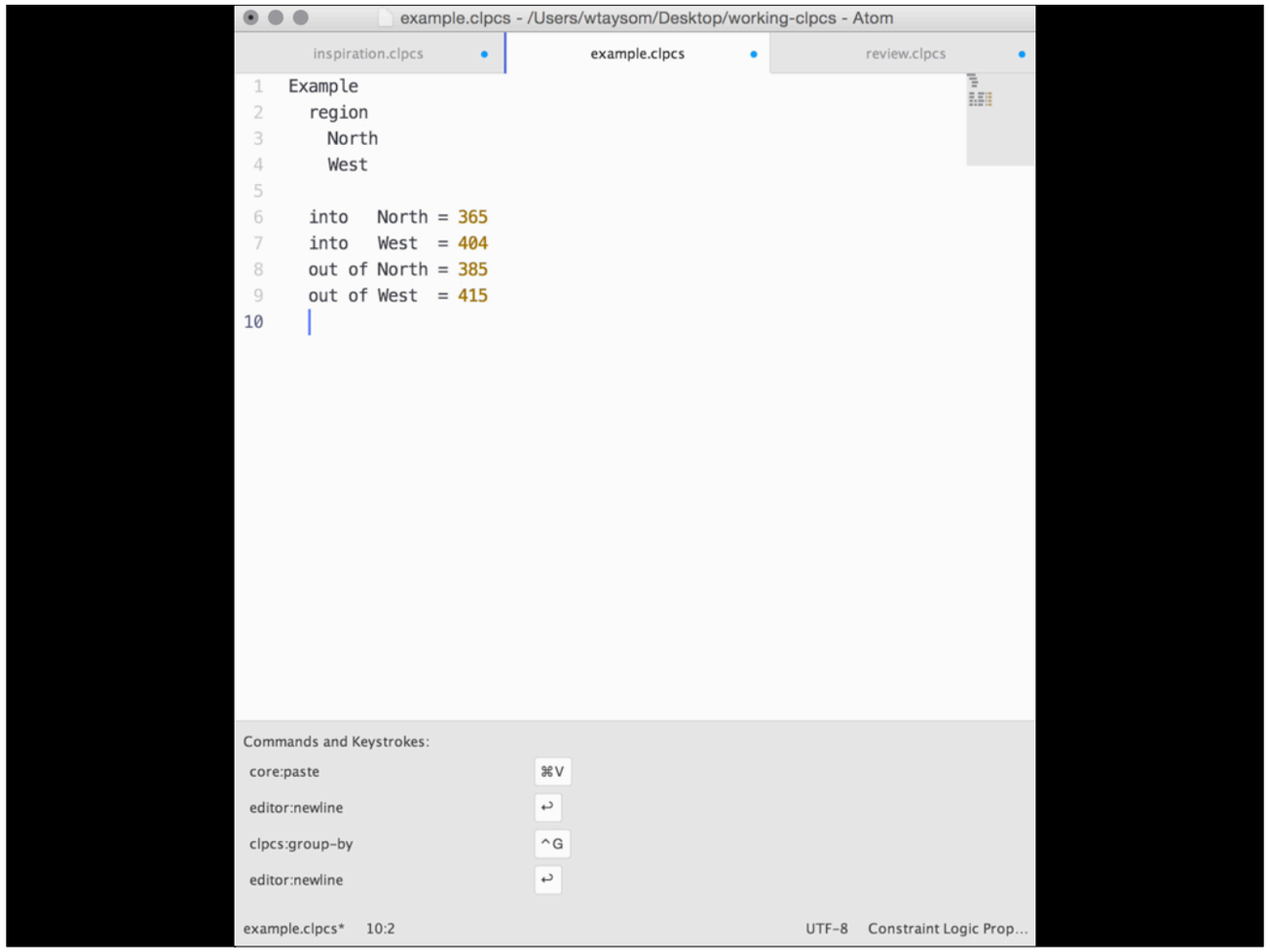
In the auction room, we often have measurements of an aggregate phenomena from which we want to infer the contribution of individual factors. For illustration, suppose we can measure all traffic (of some sort) into a region, but we would really like to find how much comes from one specific place. Certainly, the amount has to be between all and none, and it turns out that provided information about traffic into and out of other regions, we can make inferences.



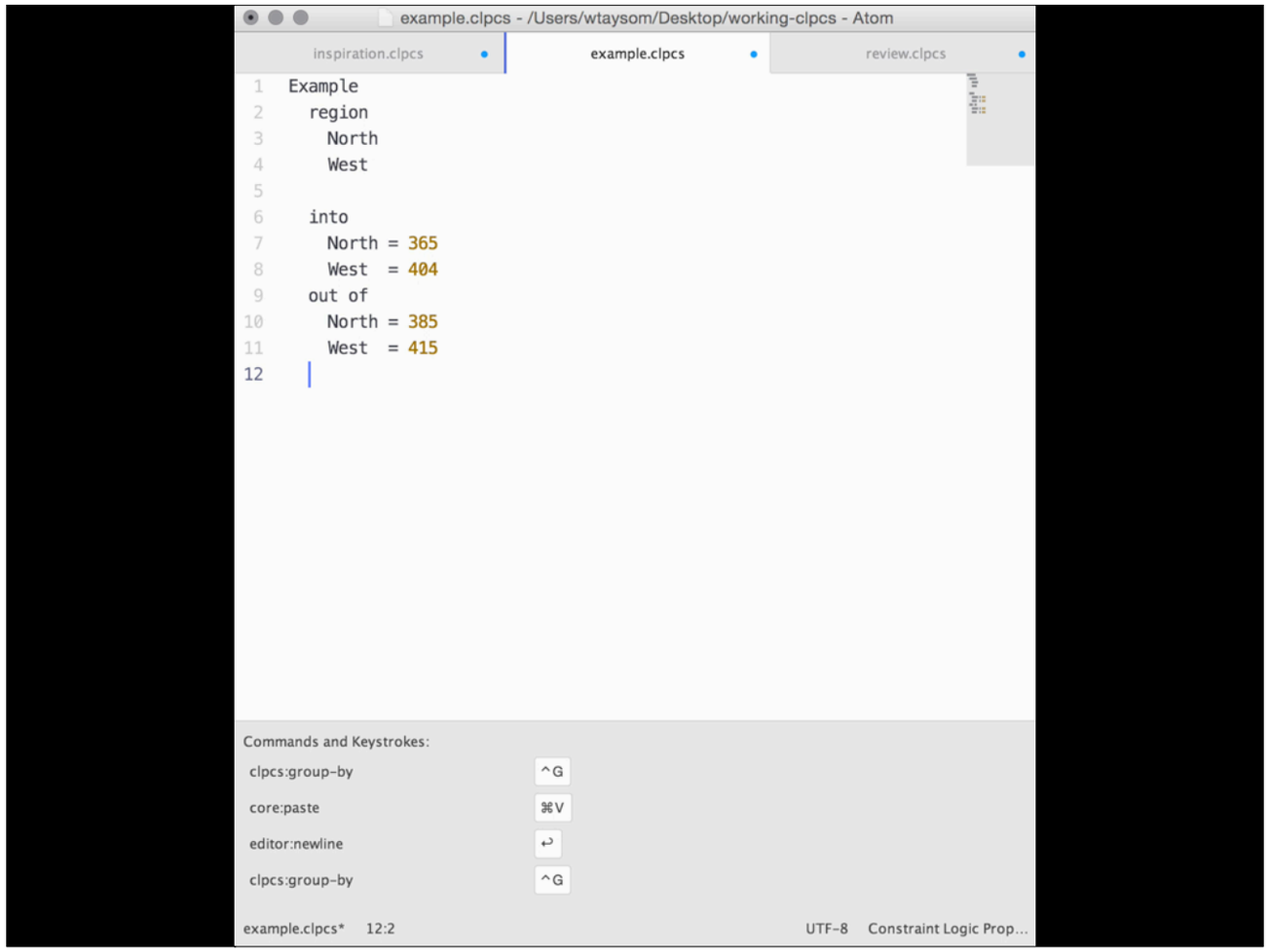
Suppose we live on a big island divided into regions. We have the North and the West. Two facts. Basically Prolog minus parenthesis and case insensitive.



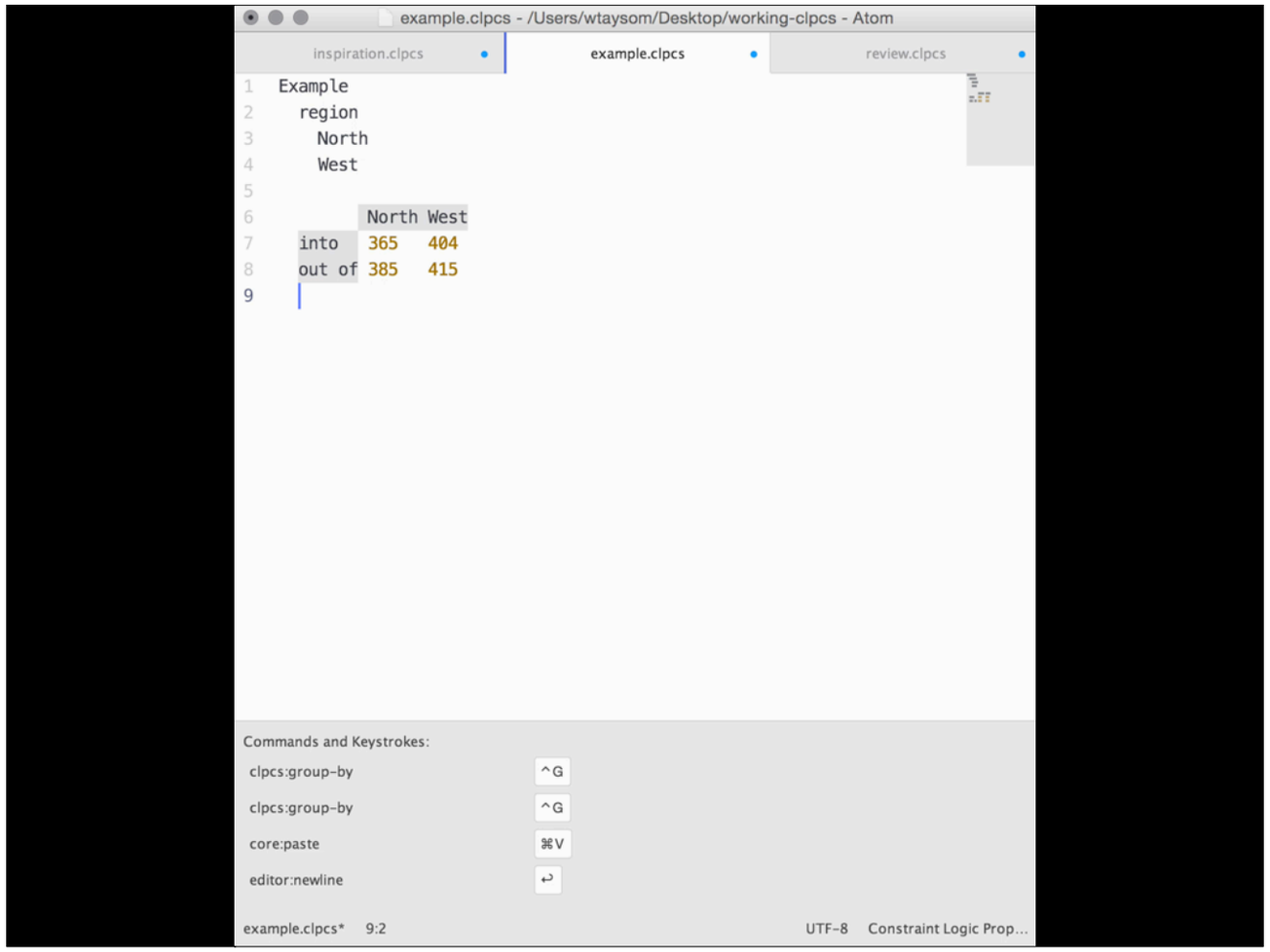
Let's look at the facts differently. Type ctrl-g to "group by" region. It's a table, just like the Inspiration, Bridge, and Example tables we've already seen. I typed notes for the introduction so you could see that this tool starts with structured text like Emacs Org mode. The grouped region block is basically the same as prefixing each line with `region`. Let's see another relation.



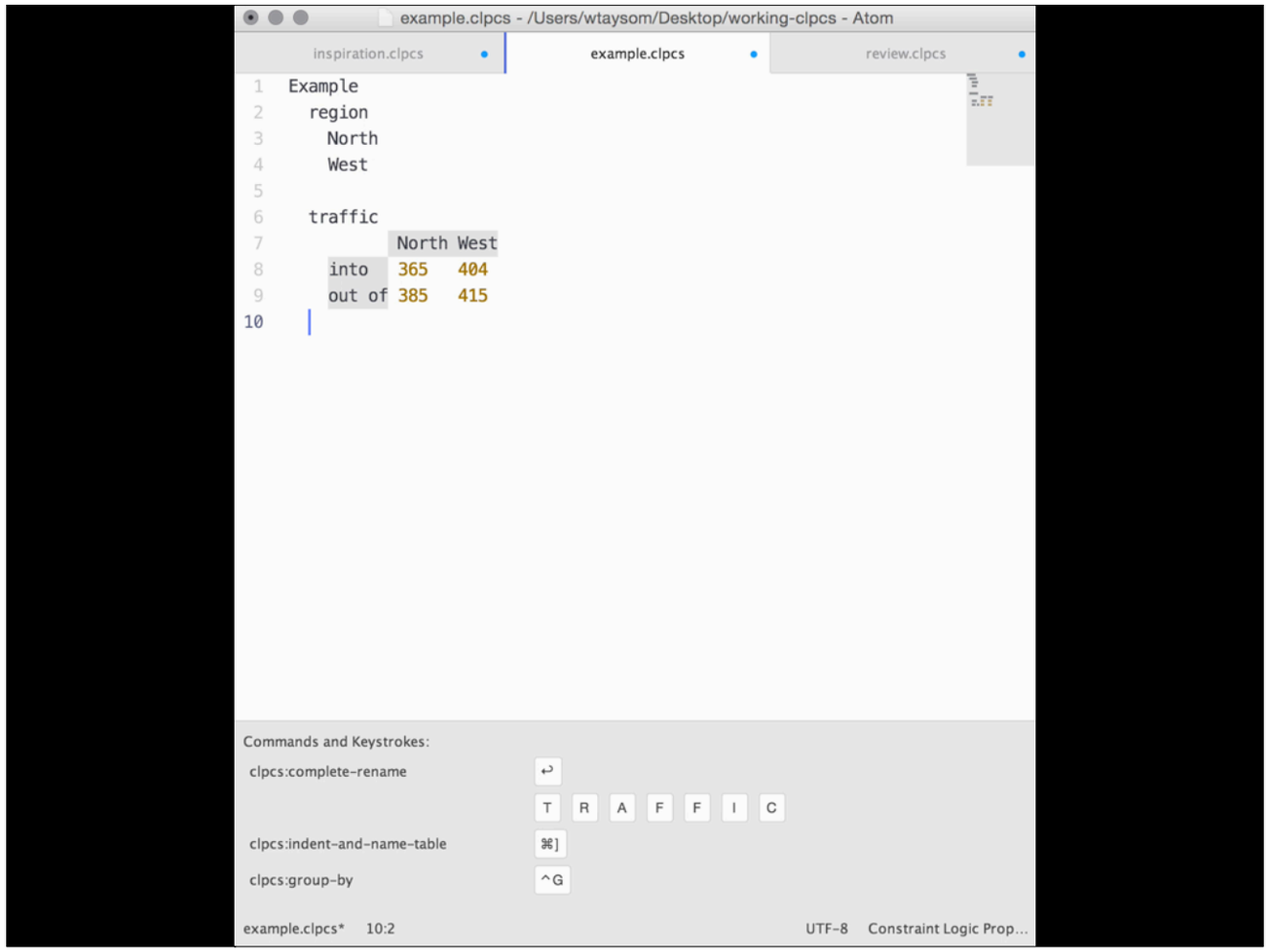
Traffic data into and out of regions. These facts have associated values. Notice from the spacing that this data is tabbed into columns. Now let's pivot the table.



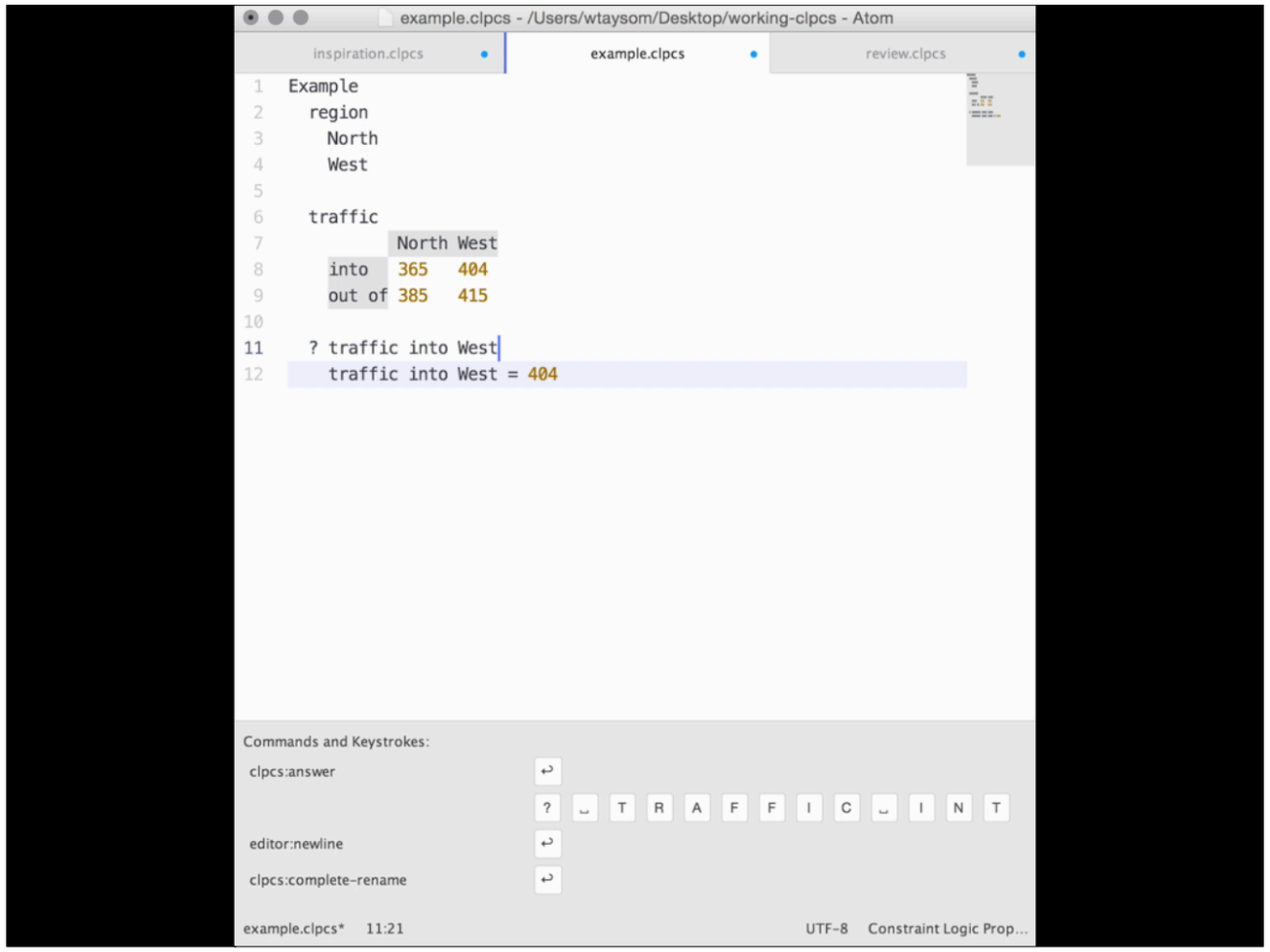
Group by the direction of traffic.



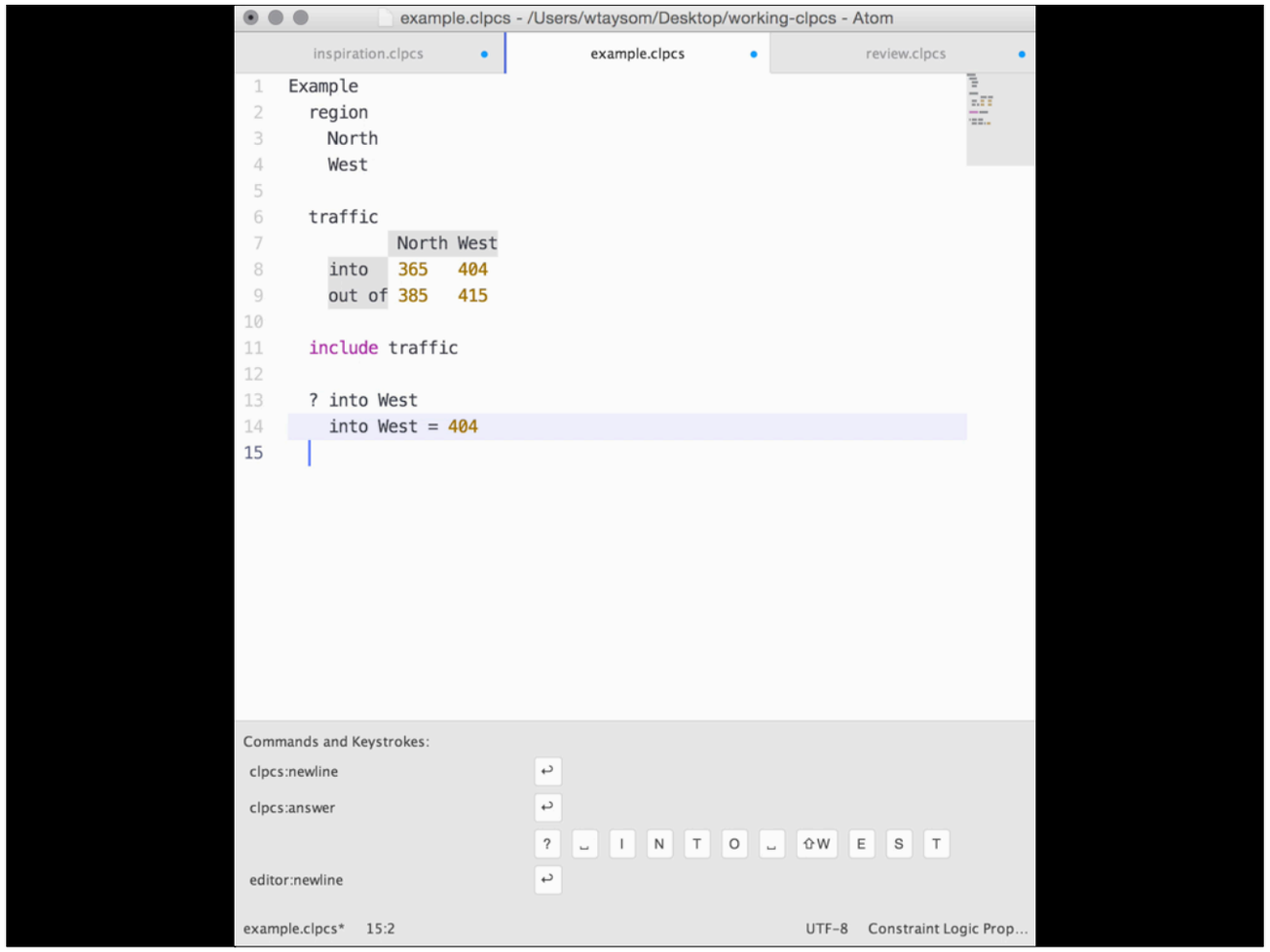
Group by region. We have a table with column and row headers. The data is the same. The group-by command changes the view: both its appearance and the commands for interacting with it. I present the interface in a text editor not because text is best (has some advantages) rather as programmers we are all used to it. I chose the Atom editor because it's fairly easy to instrument. A greater man would, no doubt, use Emacs. And some of you have built your own.



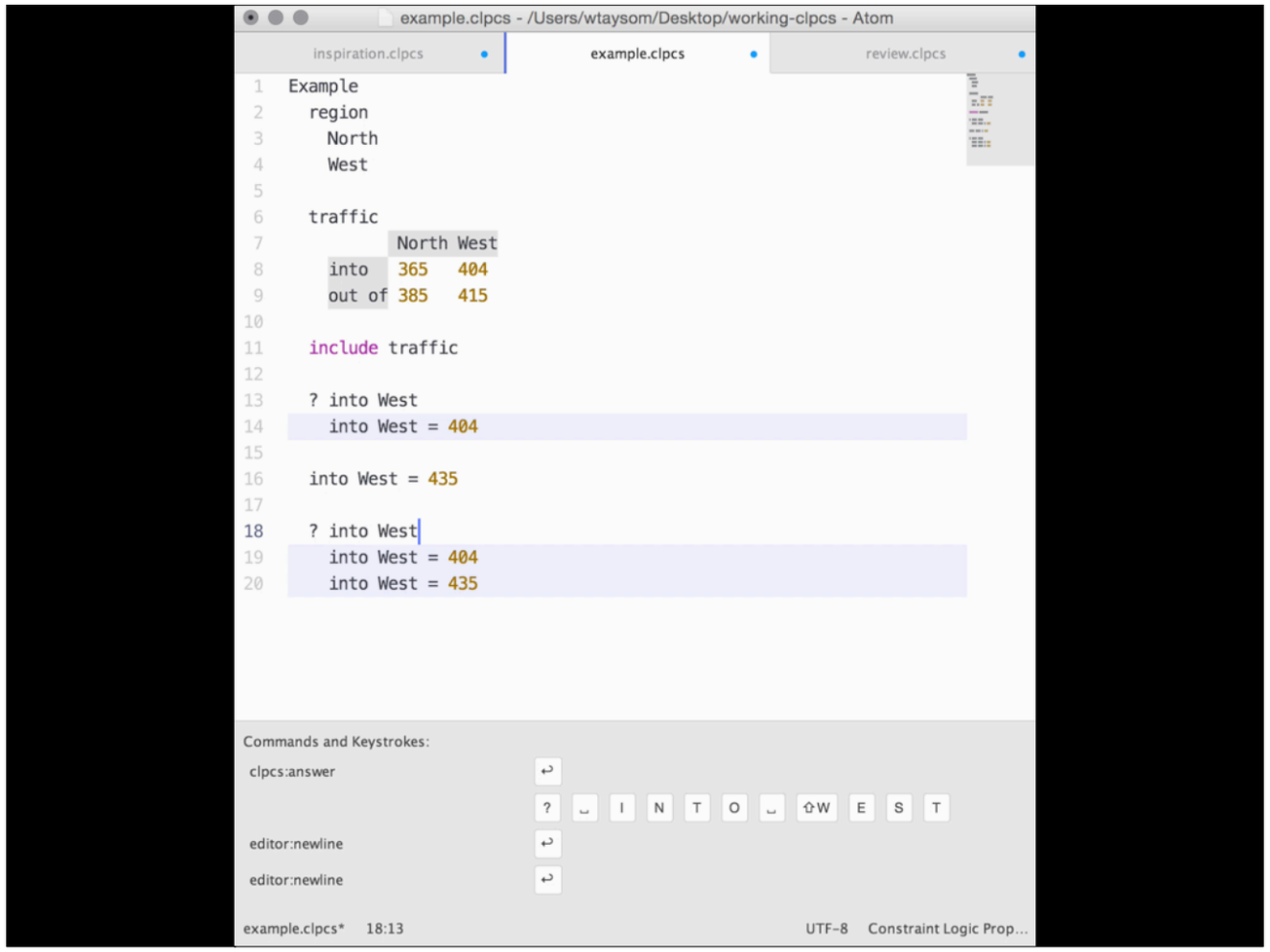
Name the table. Indenting in this system creates a new scope. By naming the table, we add a prefix to all the relations represented in the table.



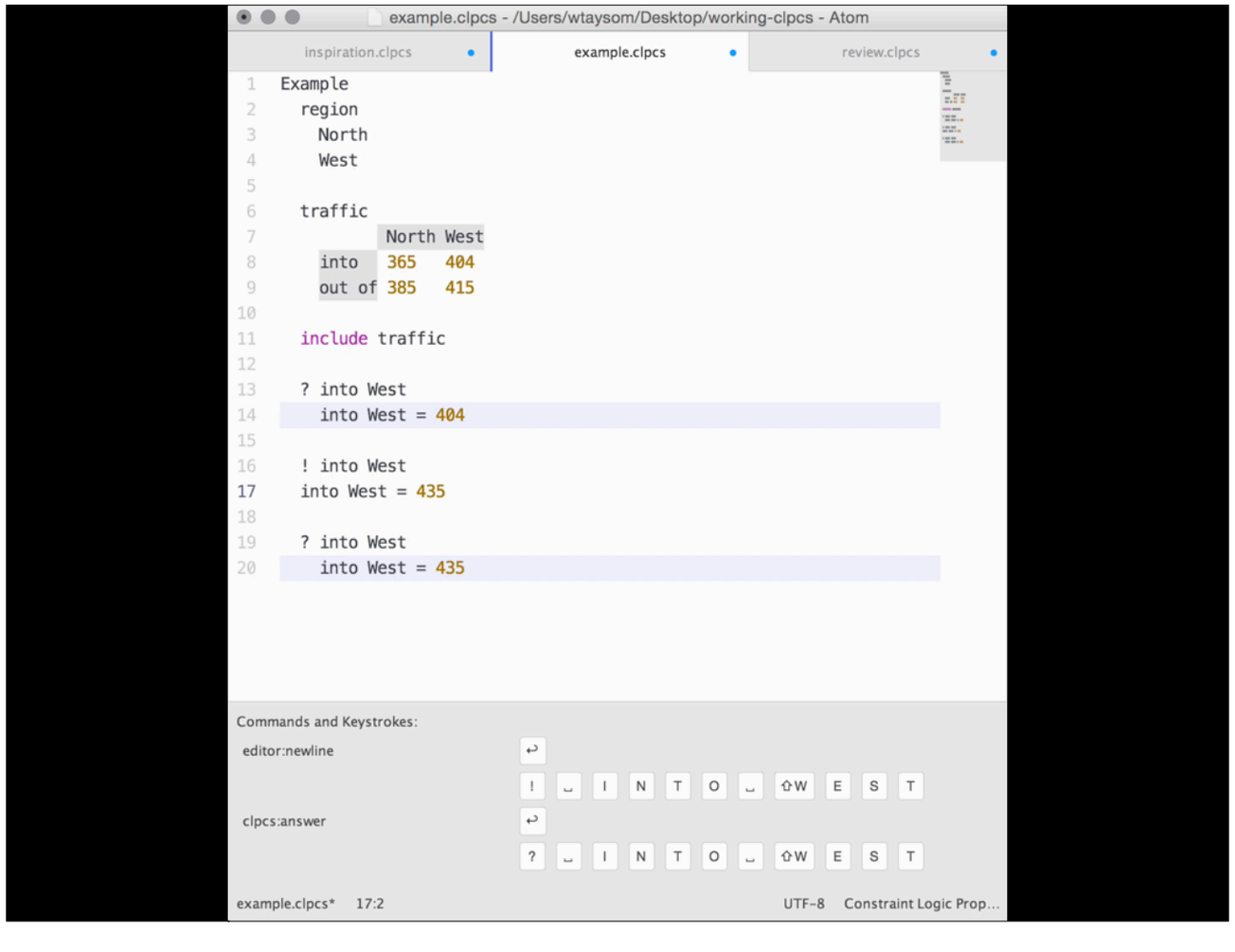
Let's ask some questions about traffic. What do we know about traffic into the West? The traffic equals 404.



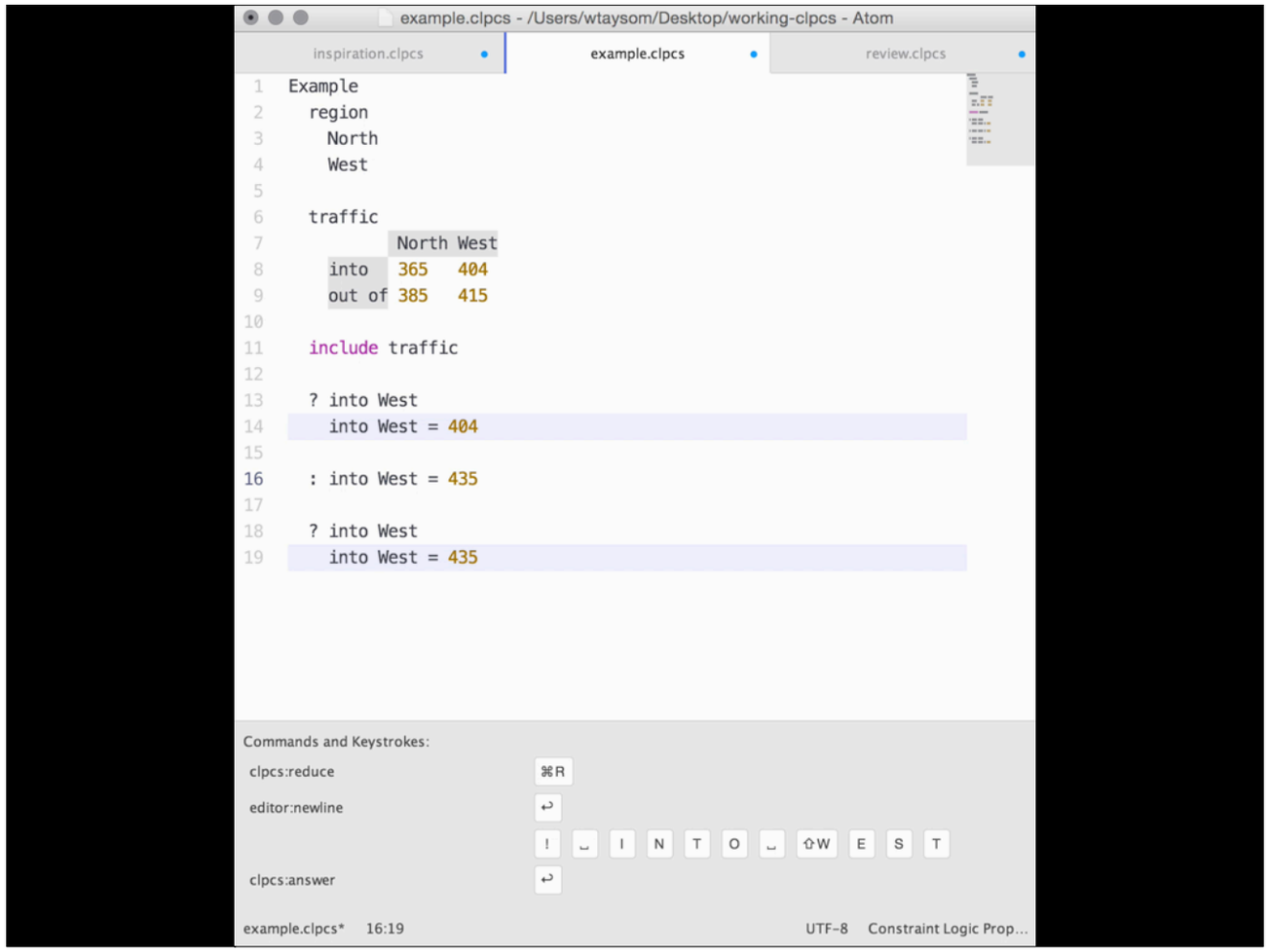
To avoid the prefix, we can include traffic facts as Example facts.



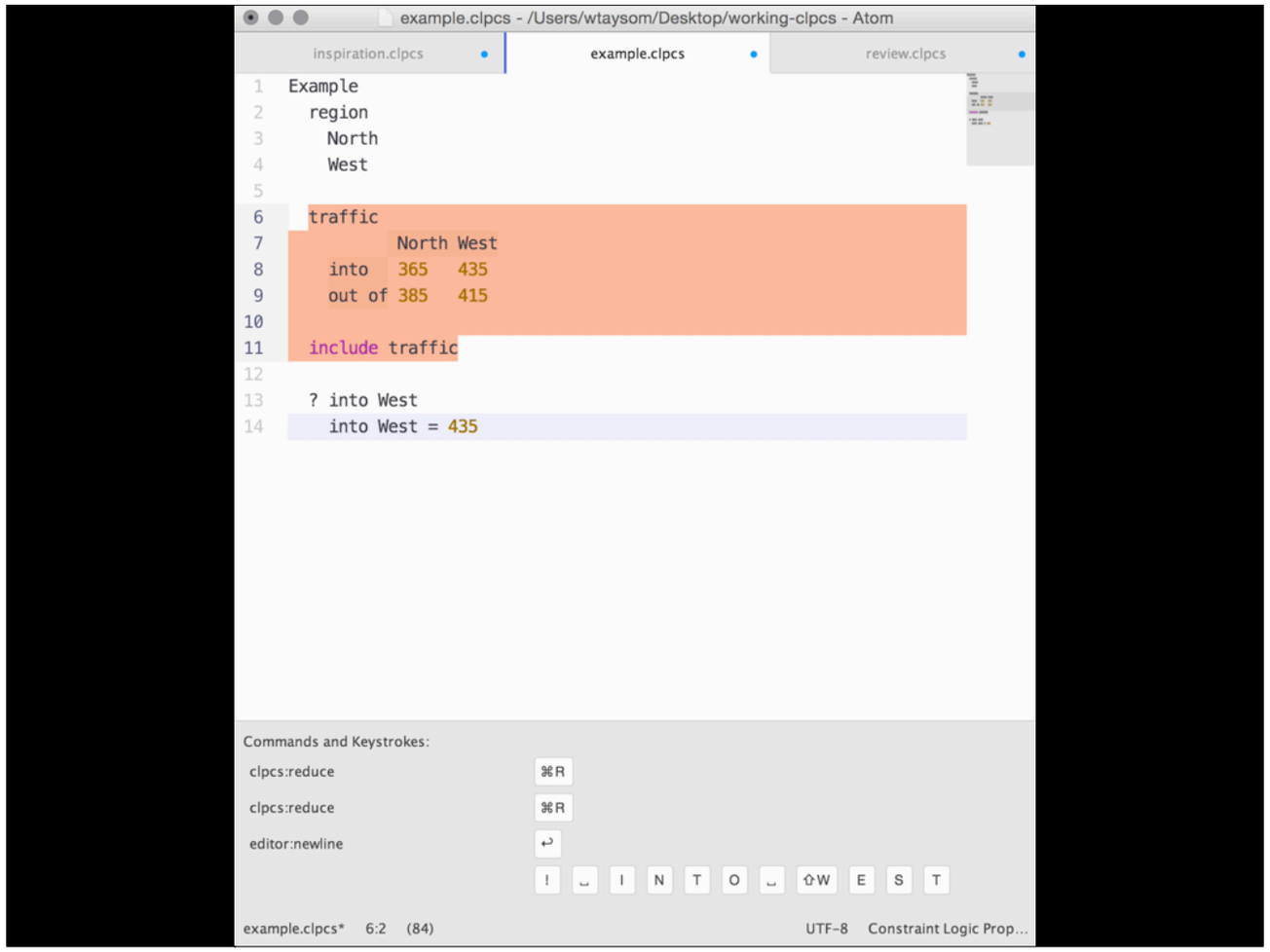
If we "change" the value, what then? We get a conflict: two values that are at odds. In the future, the equals sign is never used for assignment. Equals is an assertion that two expressions have the same value. Here we have assigned `into West` to multiple values: both 404 and 435.



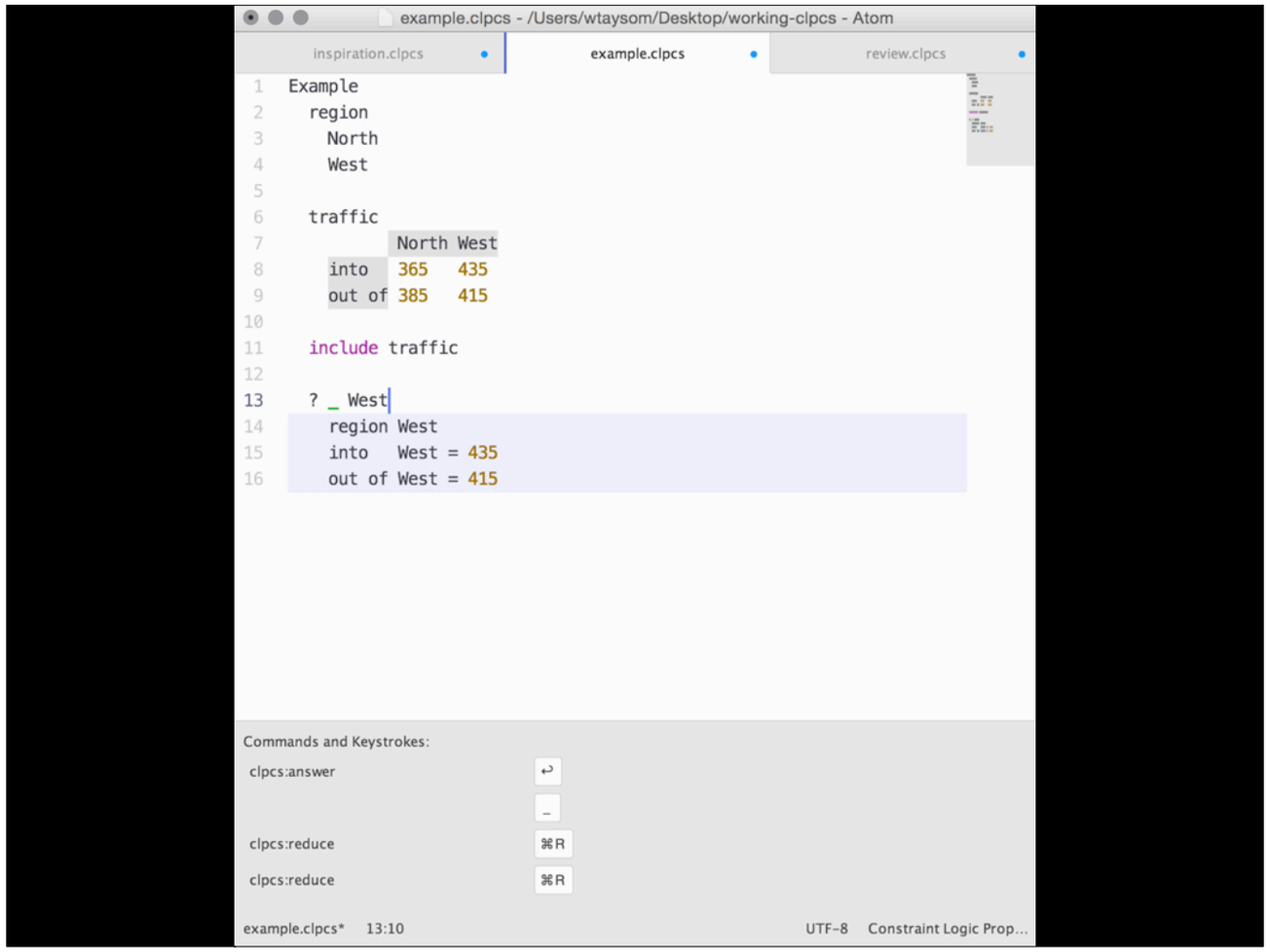
To properly override a value, first we should forget what we already asserted. To forget suggests time and state. In order to permit imperative refinement of a model while retaining reactive update, facts collect from top to bottom. Each assertion is a function from one set of facts to another. A change to one fact may change the answers to all the questions below it. In contrast reevaluating an expression in a Mathematica notebook does not affect related expressions, so the text can get out of sync with the heap of definitions. Ask non-programmer Mathematica-ticians about their workarounds sometime.



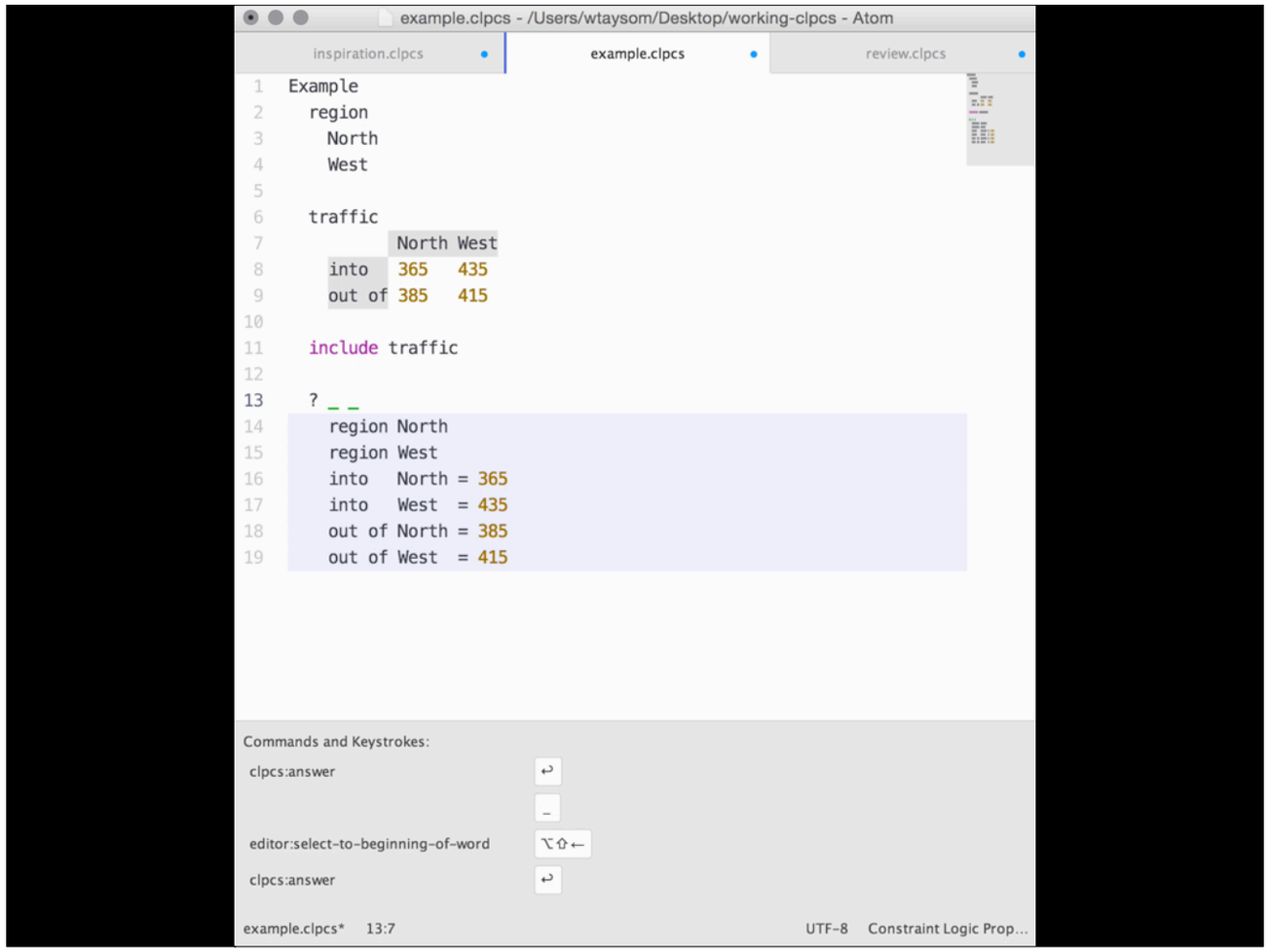
To retract and assert together is to revise. The reduce command merges facts. It normalizes their total effect on the model. Call it function composition.



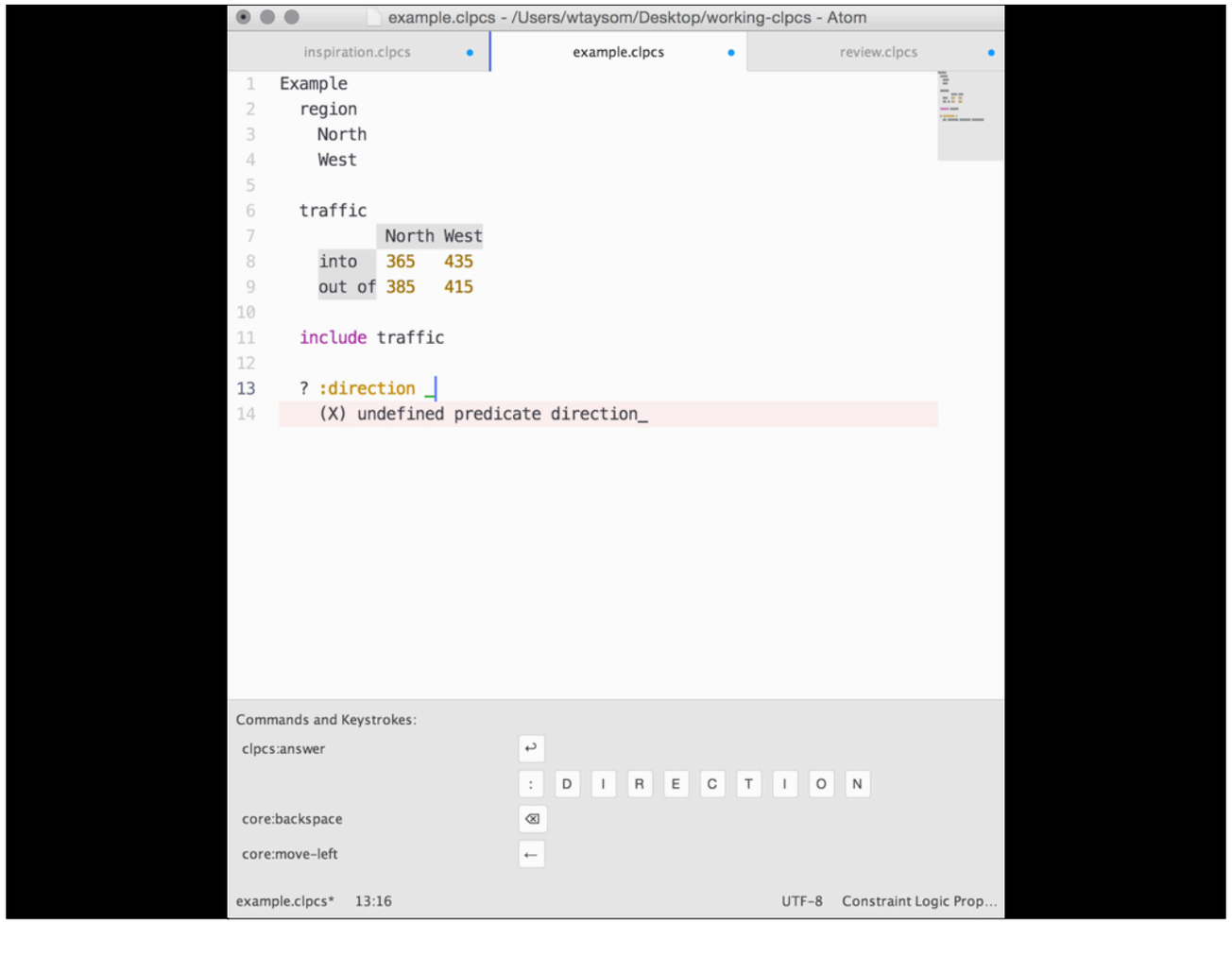
Let's reduce again to revise the original. We update `into West` by patching in-place.



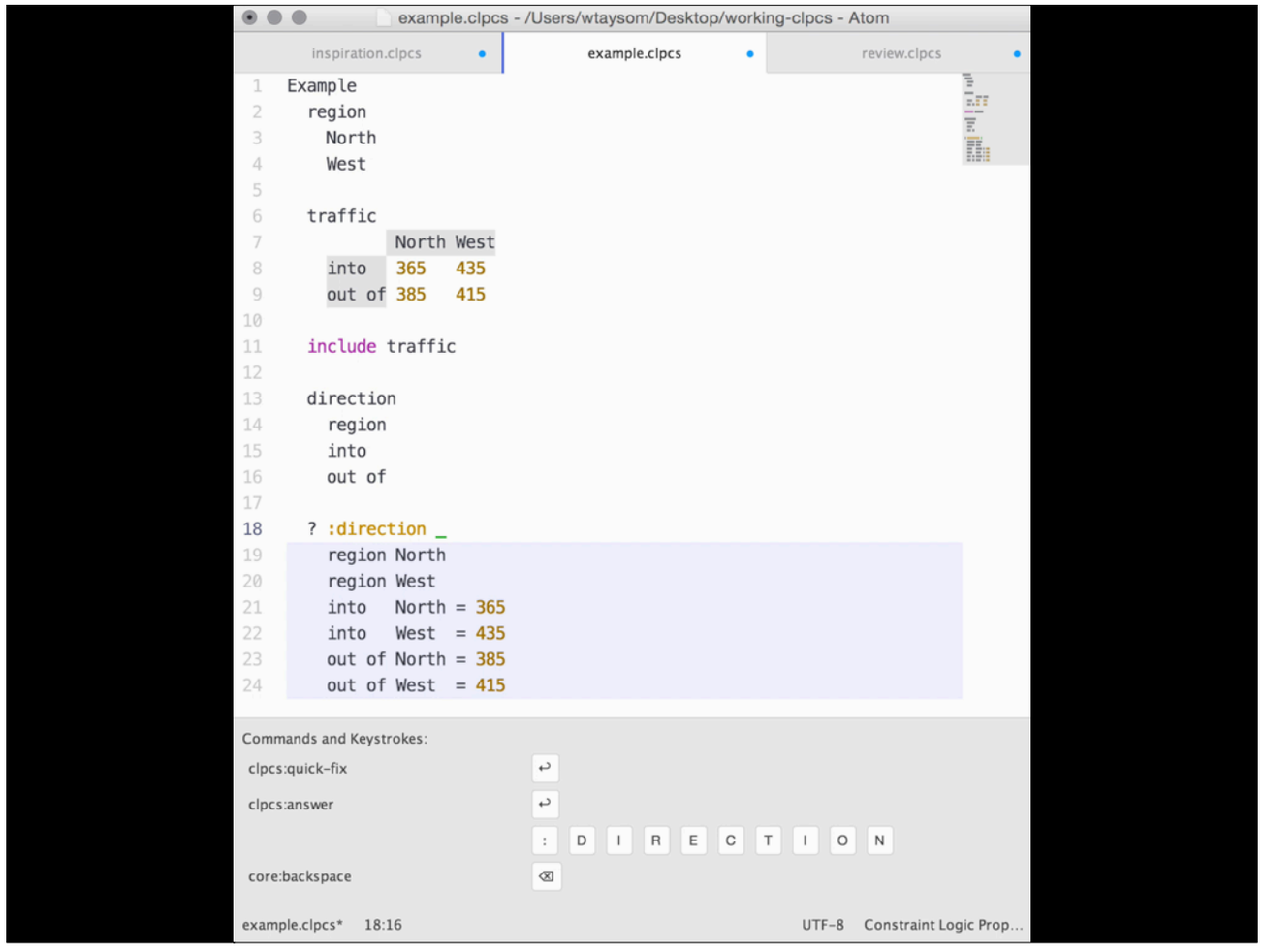
What do we know about the West overall? A blank in a question is answered by every unifying substitution. The West is the object of three predicates: `region`, `into`, and `out of`.



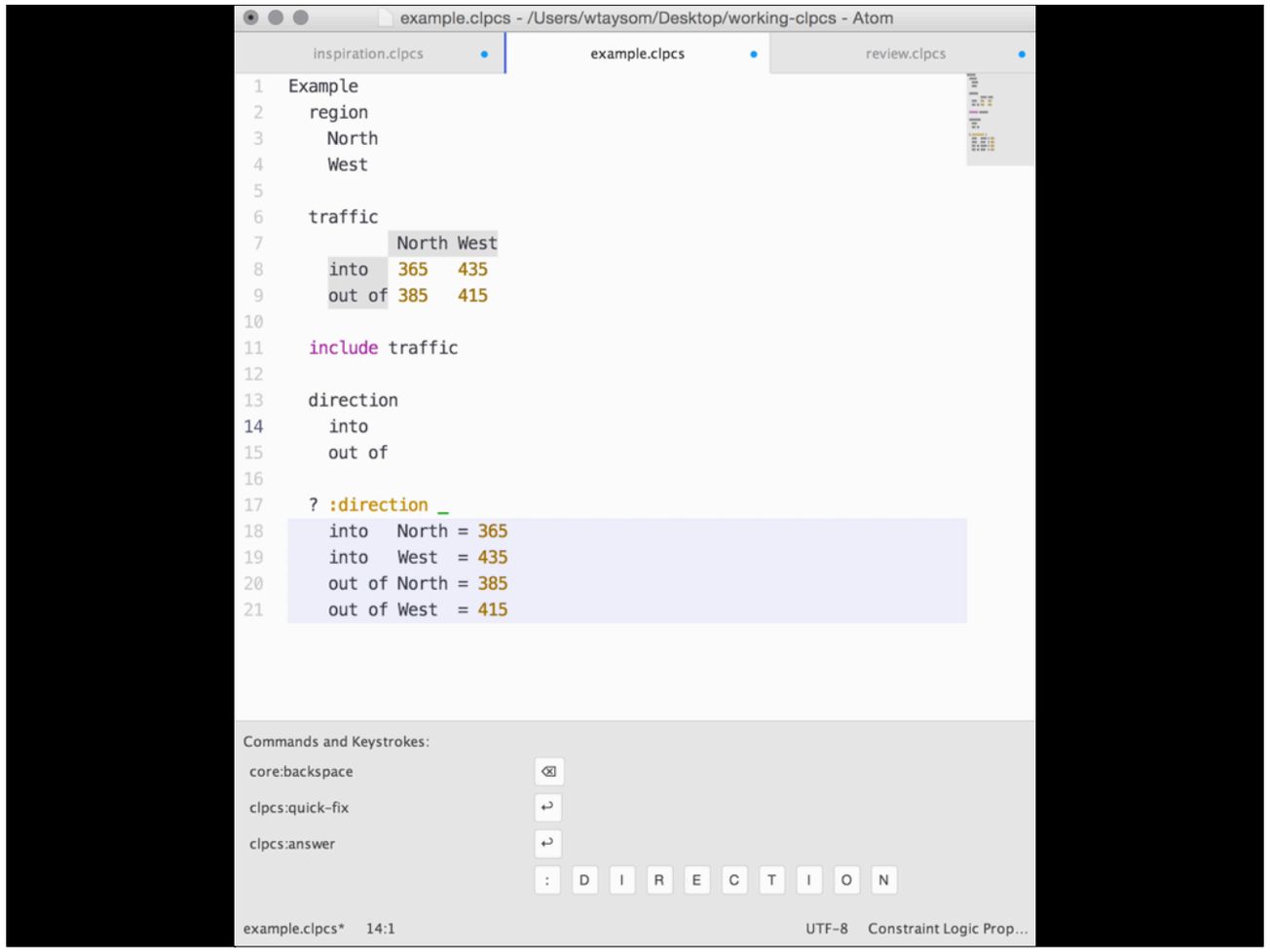
And if we add another blank? We get all of our facts because they all happen to be predicate object pairs.



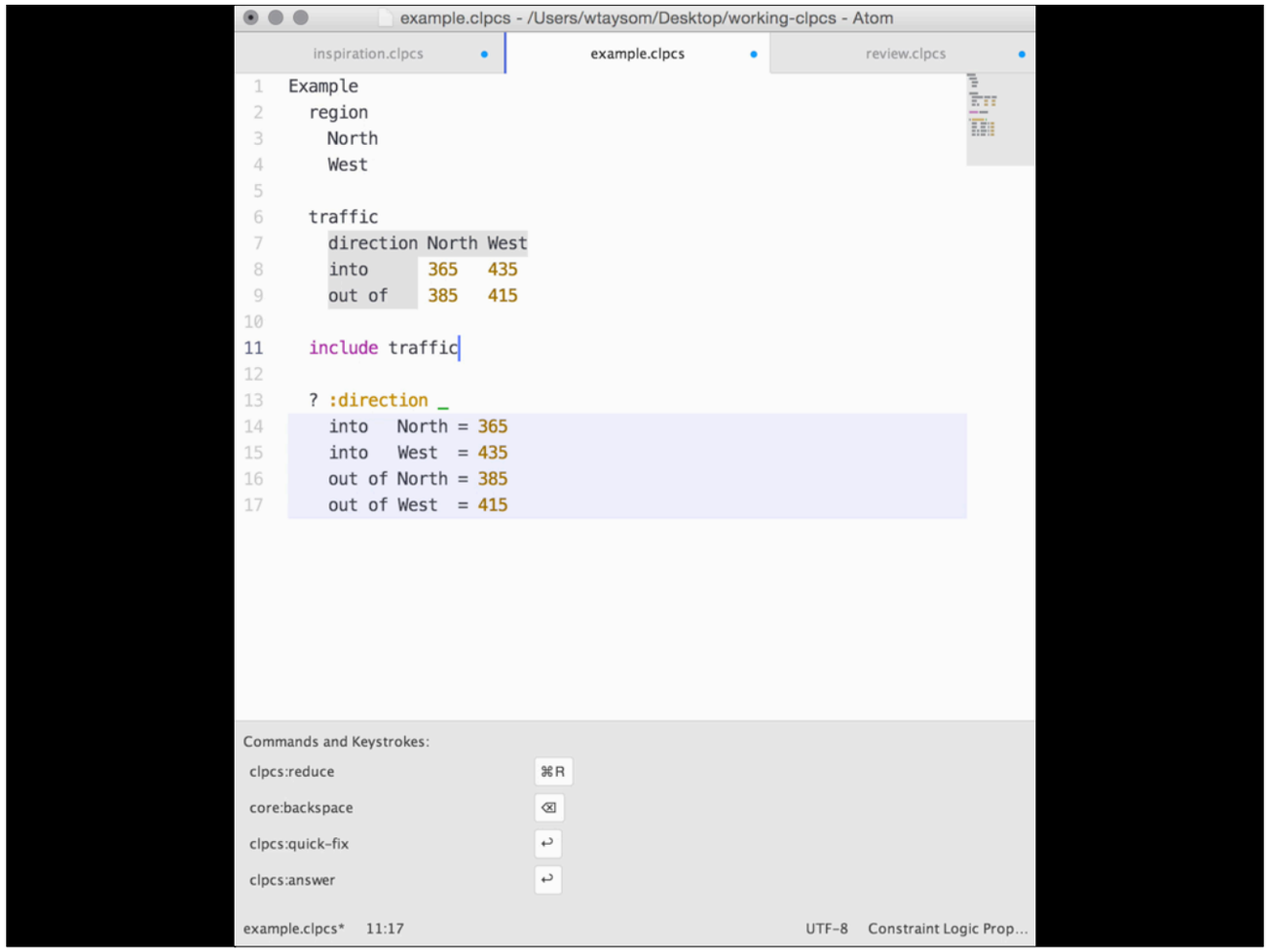
Suppose we just want the facts involving a direction. `:direction` unifies with everything that satisfies the direction predicate; however, `direction` is undefined.



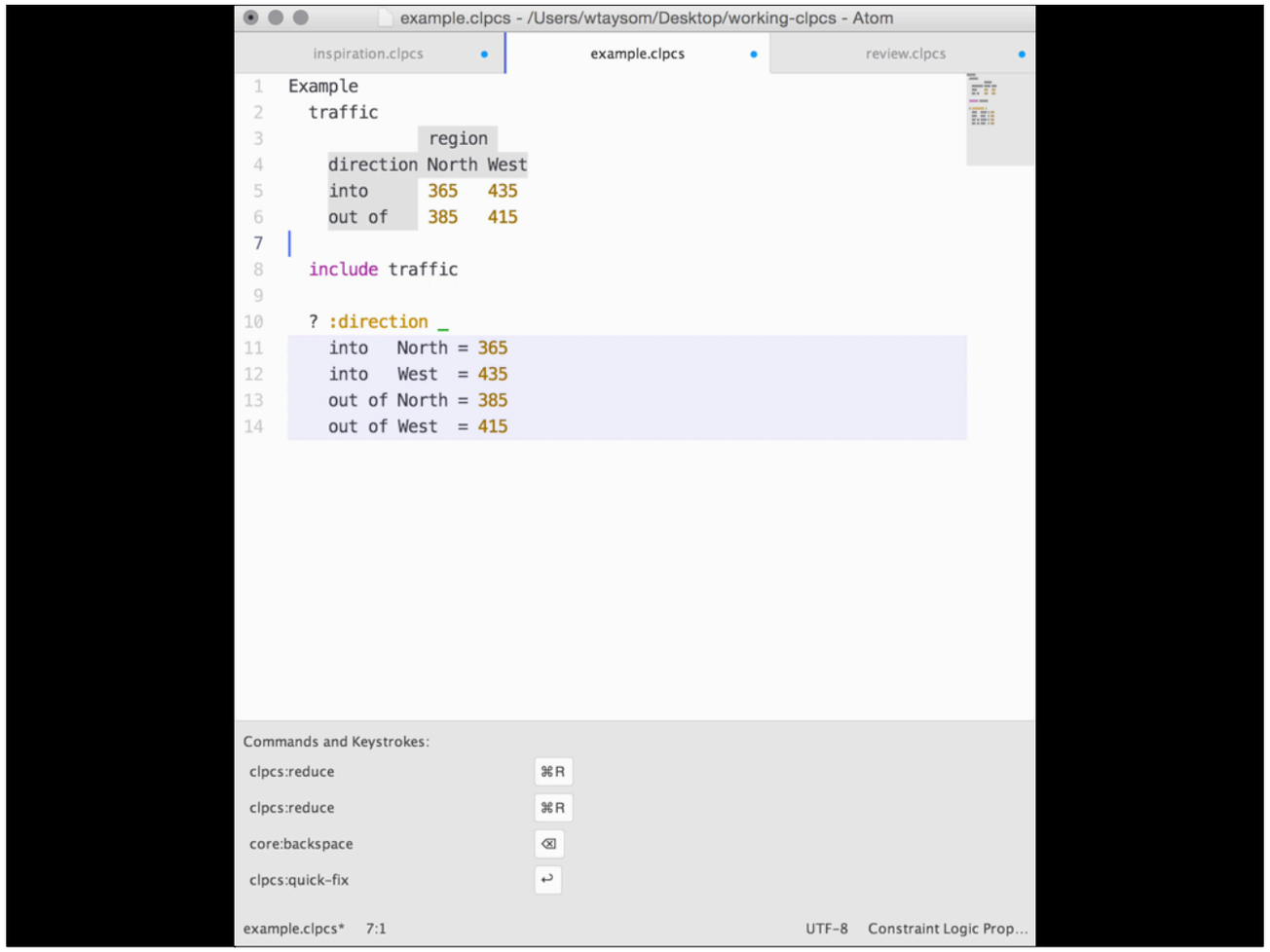
Hit enter again for a quick fix. The system can guess what might be a direction from the predicates that already match an unrestricted blank. No magic here, just requires some context. The trick is having a system where context is readily available.



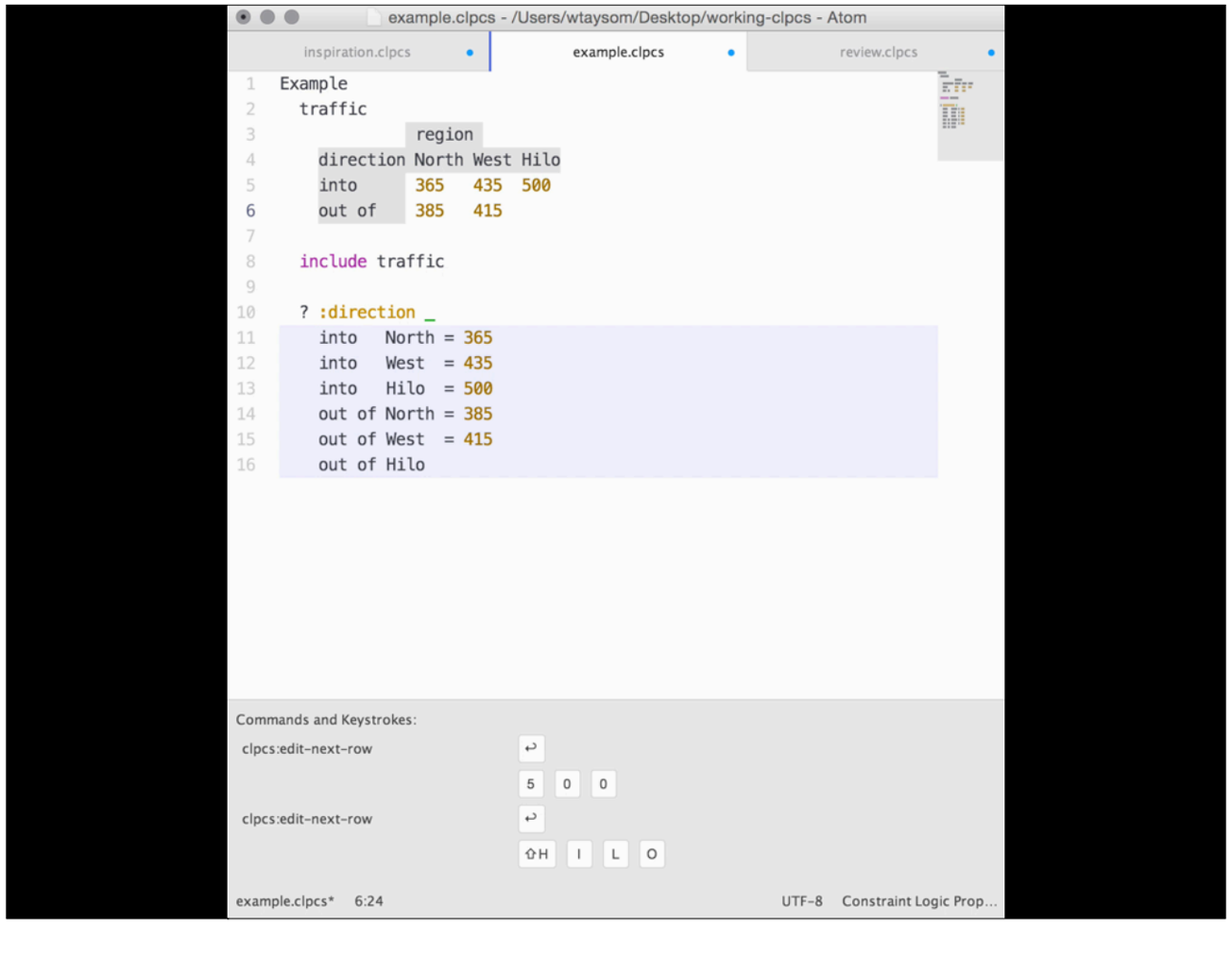
Restrict `direction` properly, and we reproduce the traffic table.



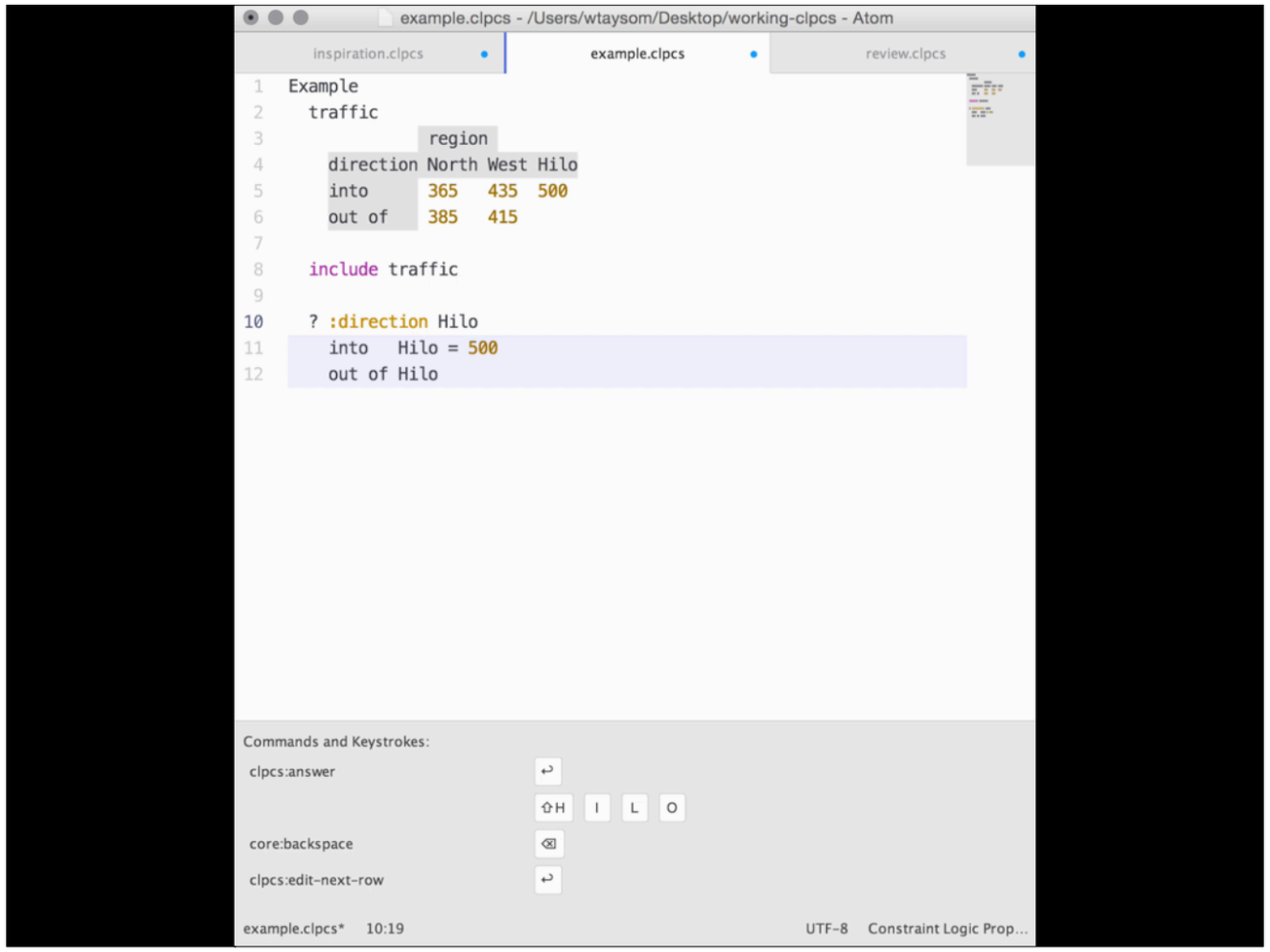
Since directions label a column in the traffic table, we can combine them.



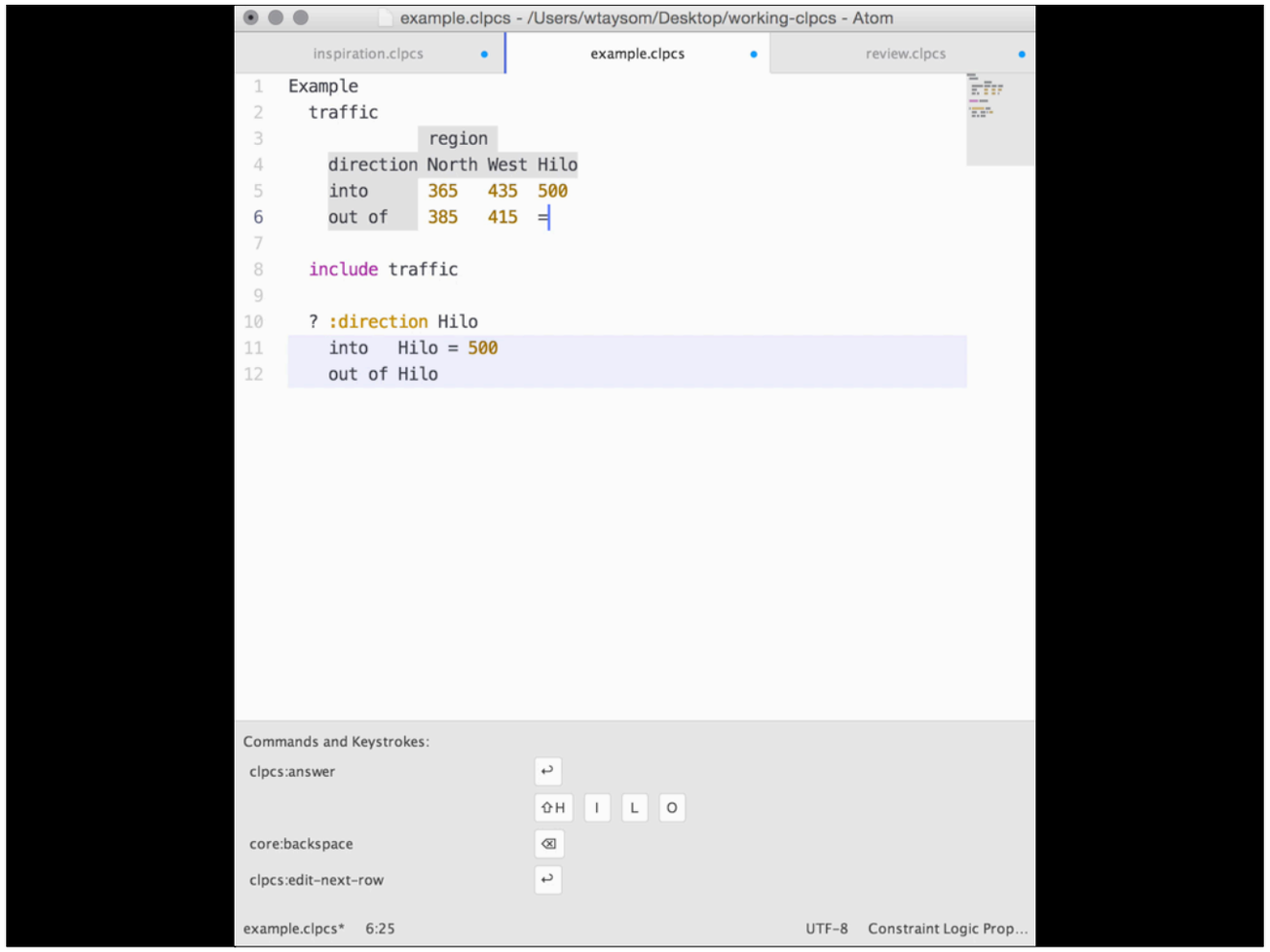
Likewise, since regions label a row, let's merge again. We now have three relations embedded in one table: region, direction, and how their product maps onto values.



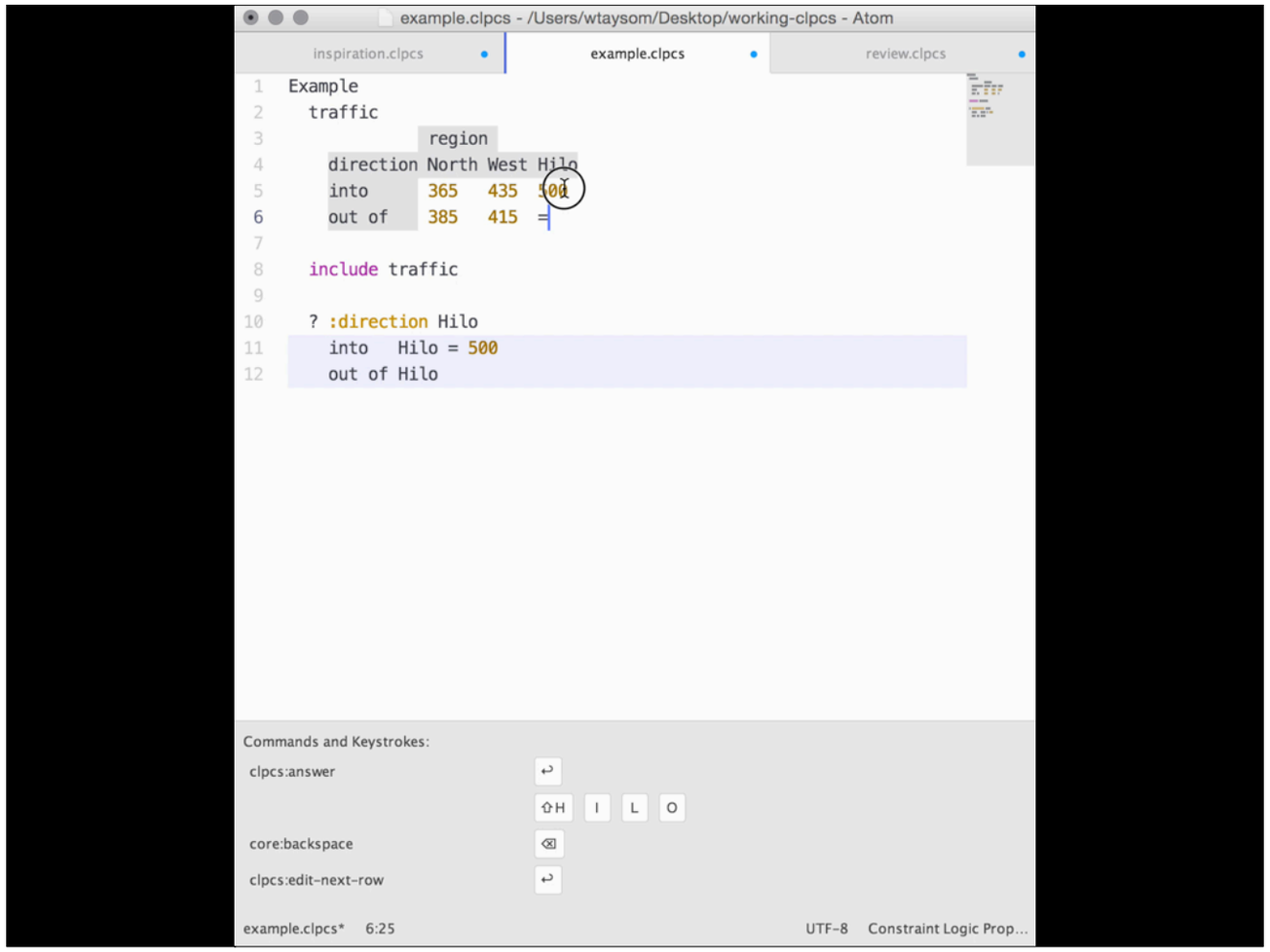
Now when we modify our traffic table, we assert several facts at once. Notice that new Hilo facts appear as an answer to our direction question.



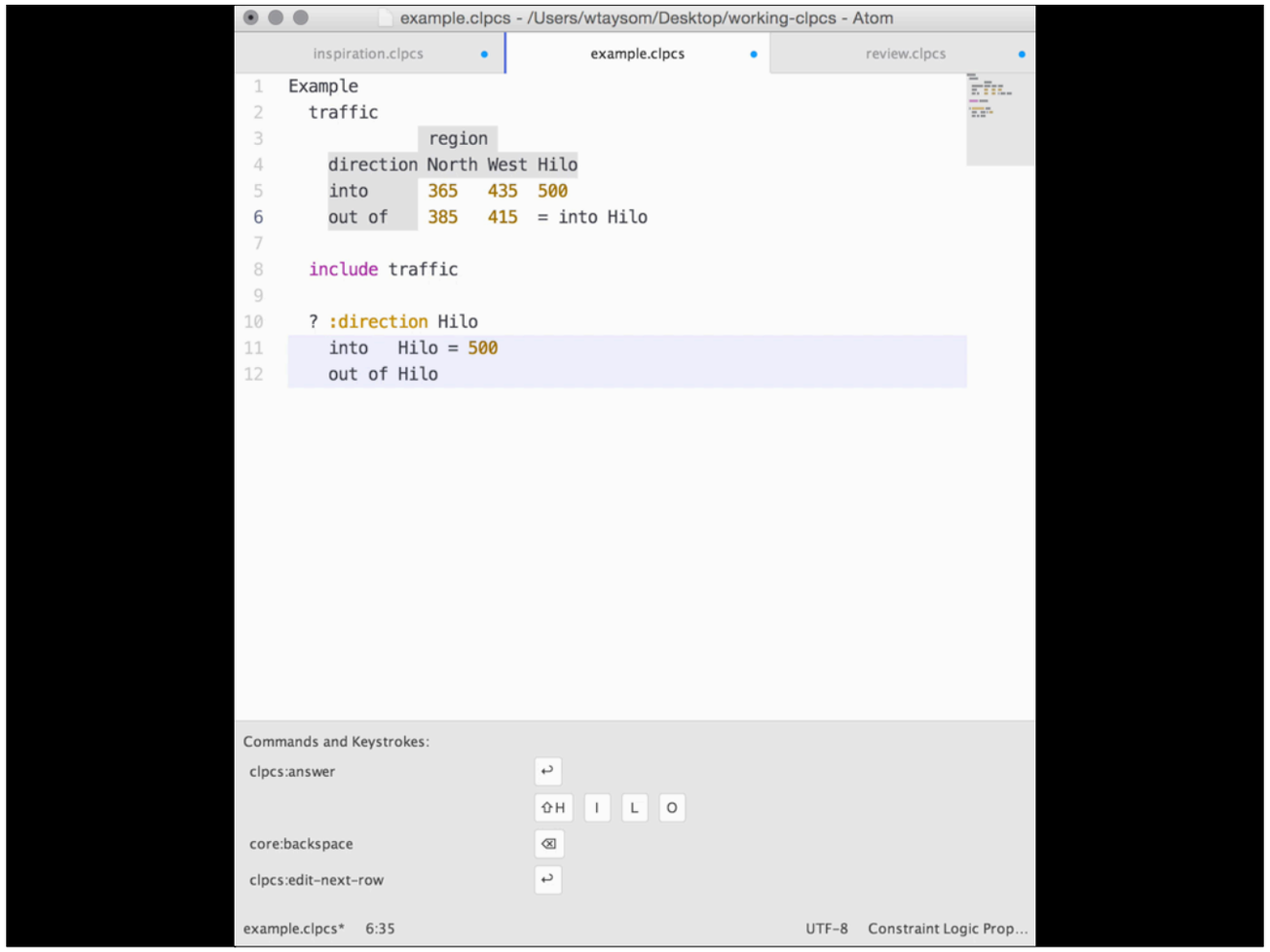
Consider Hilo specifically. We know that `out of Hilo` is a fact, but we don't know its value since we left a cell blank.



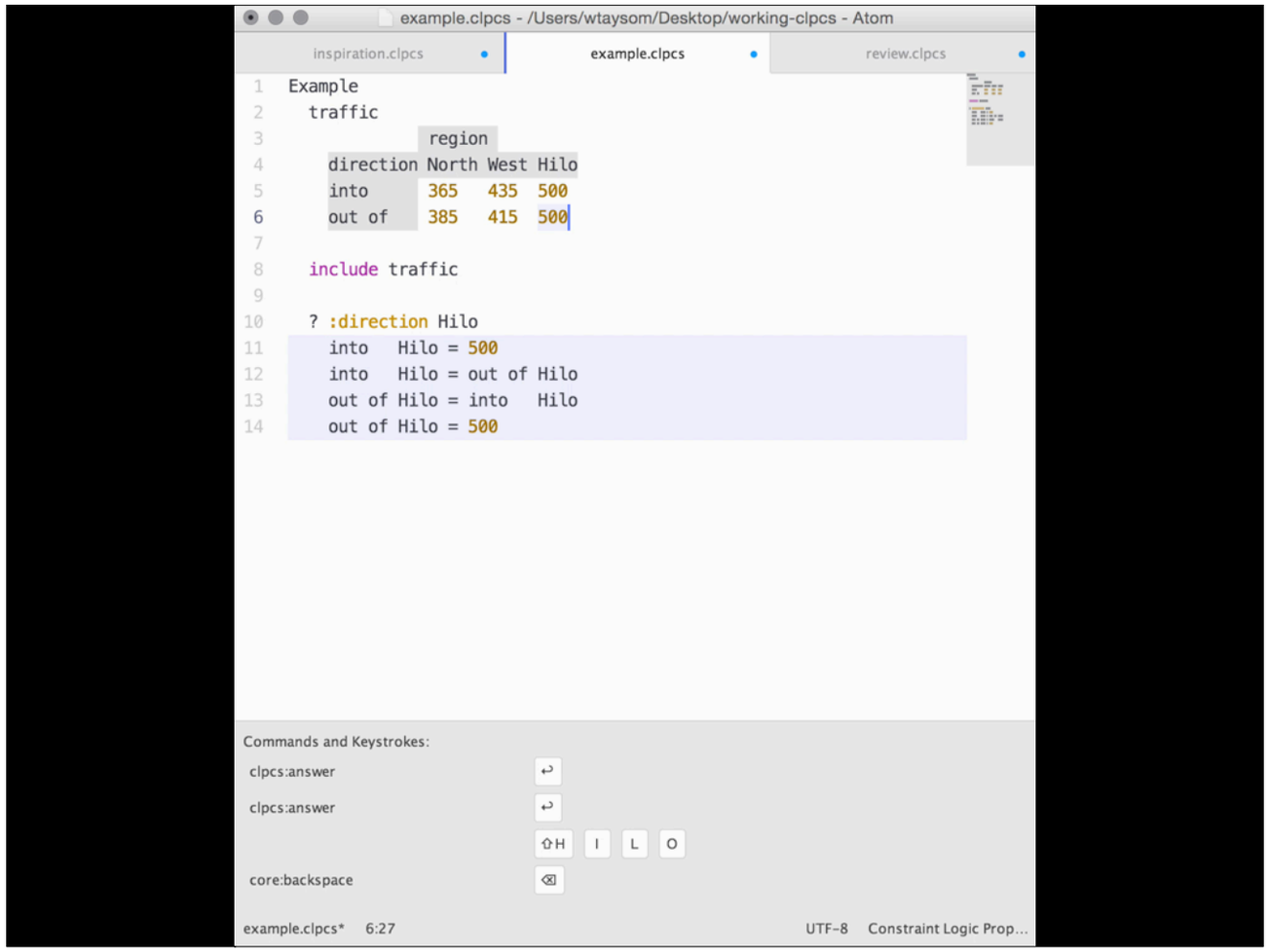
To fix the value, let's use a formula. Equals.


example.clpcs - /Users/wtaysom/Desktop/working-clpcs - Atom
inspiration.clpcs
example.clpcs
review.clpcs
Example
traffic
region
direction North West Hilo
into 365 435 500
out of 385 415 =
include traffic
? :direction Hilo
into Hilo = 500
out of Hilo
Commands and Keystrokes:
clpcs:answer
⇧H I L O
core:backspace
clpcs:edit-next-row
example.clpcs* 6:25
UTF-8 Constraint Logic Prop...

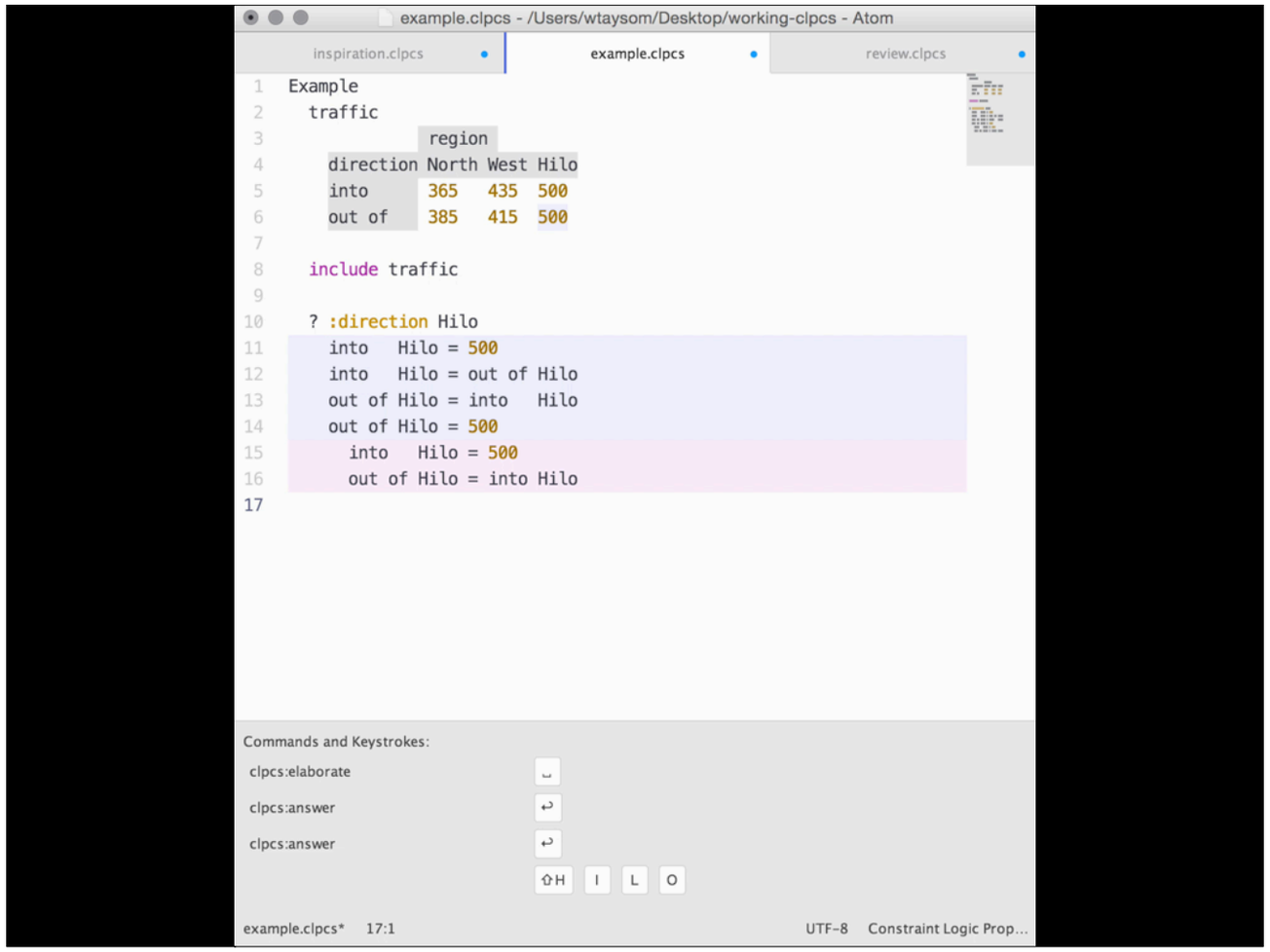
Click.



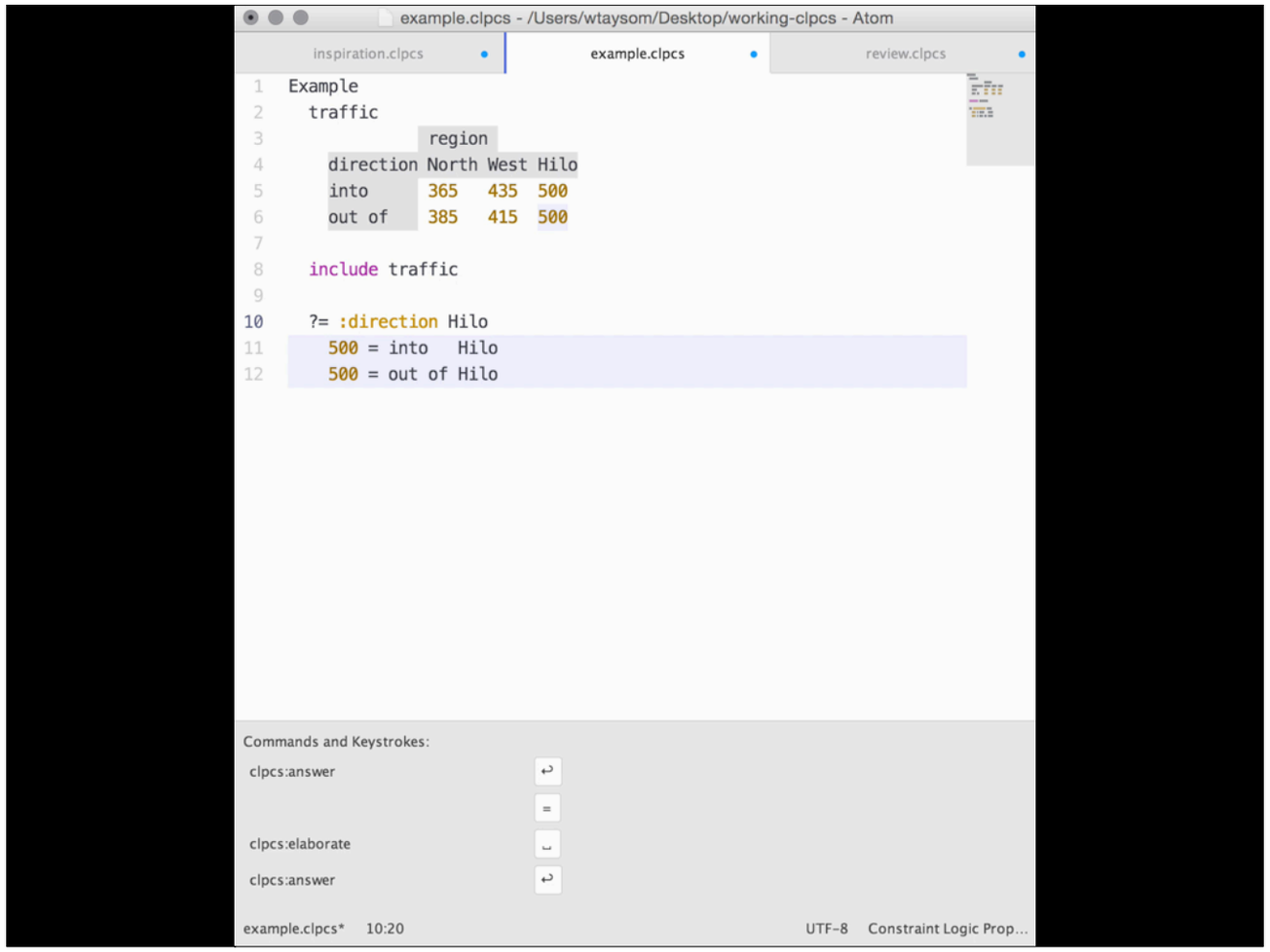
Return. Just like a spreadsheet.



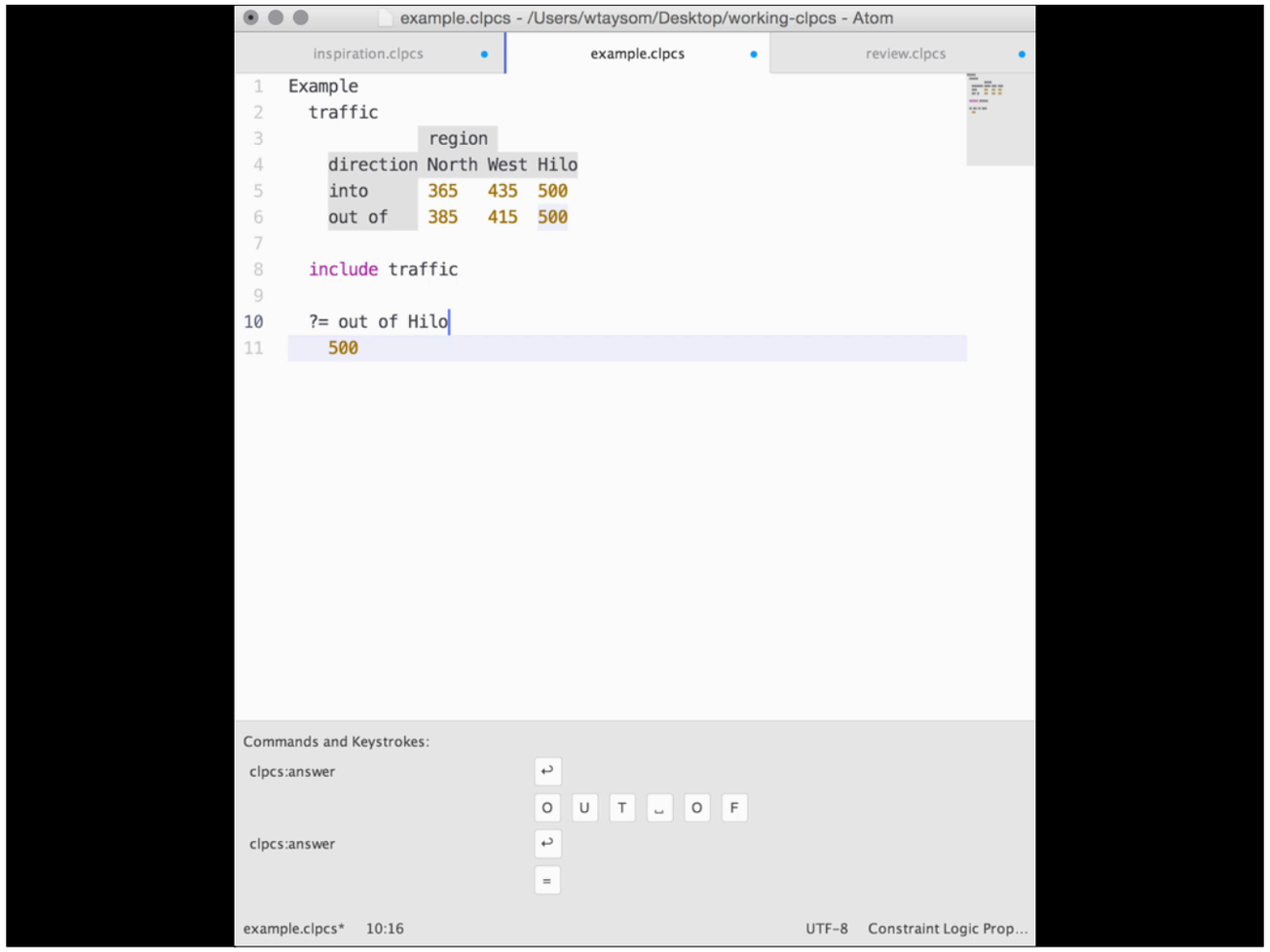
Return. Just like a spreadsheet.



From the formula, the system infers the value of `out of Hilo`. `out of Hilo` equals `into Hilo`. `into Hilo` equals 500. A two link equality chain.



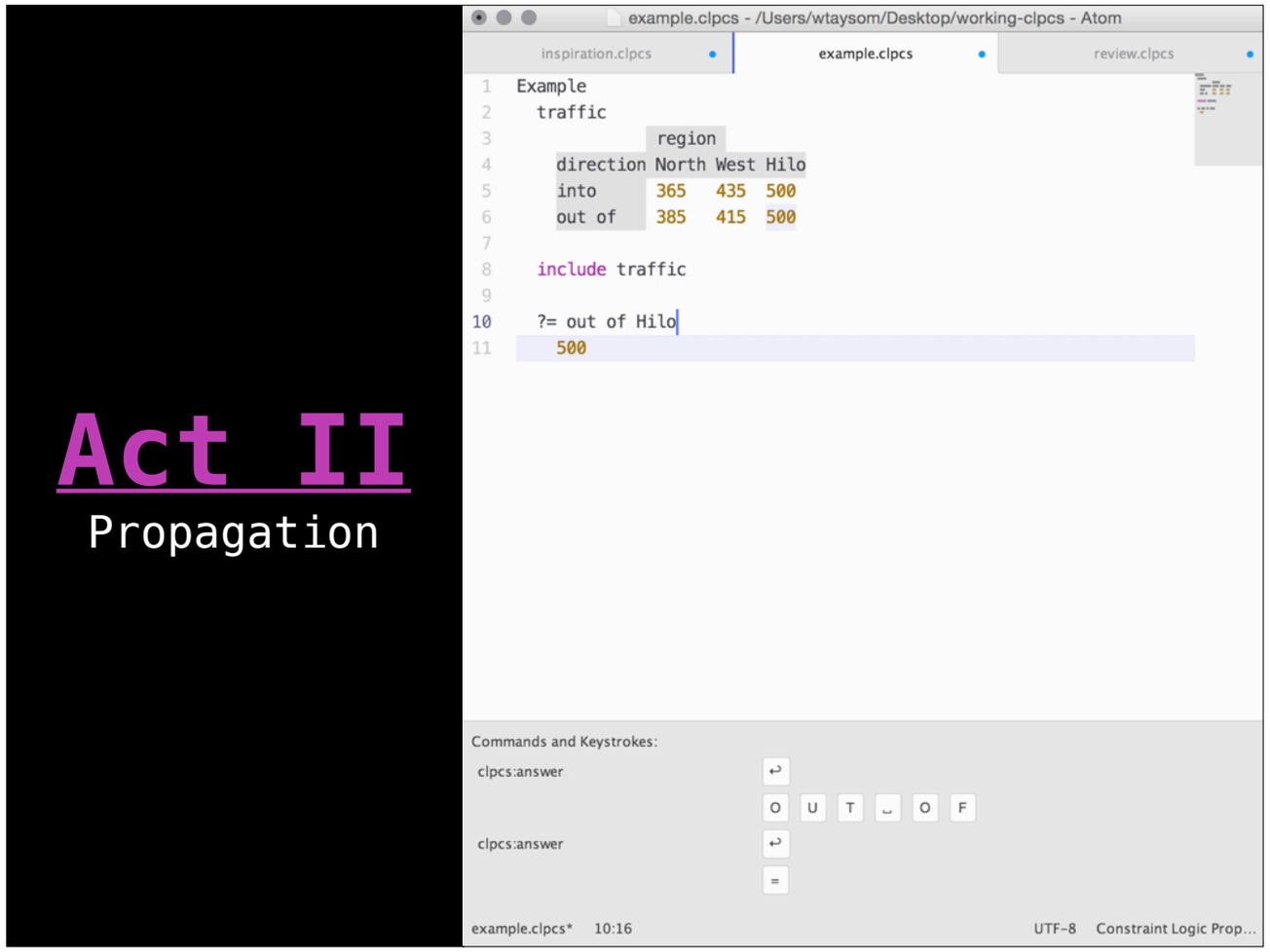
If we want to focus on values instead of facts, we ask a different sort of question.



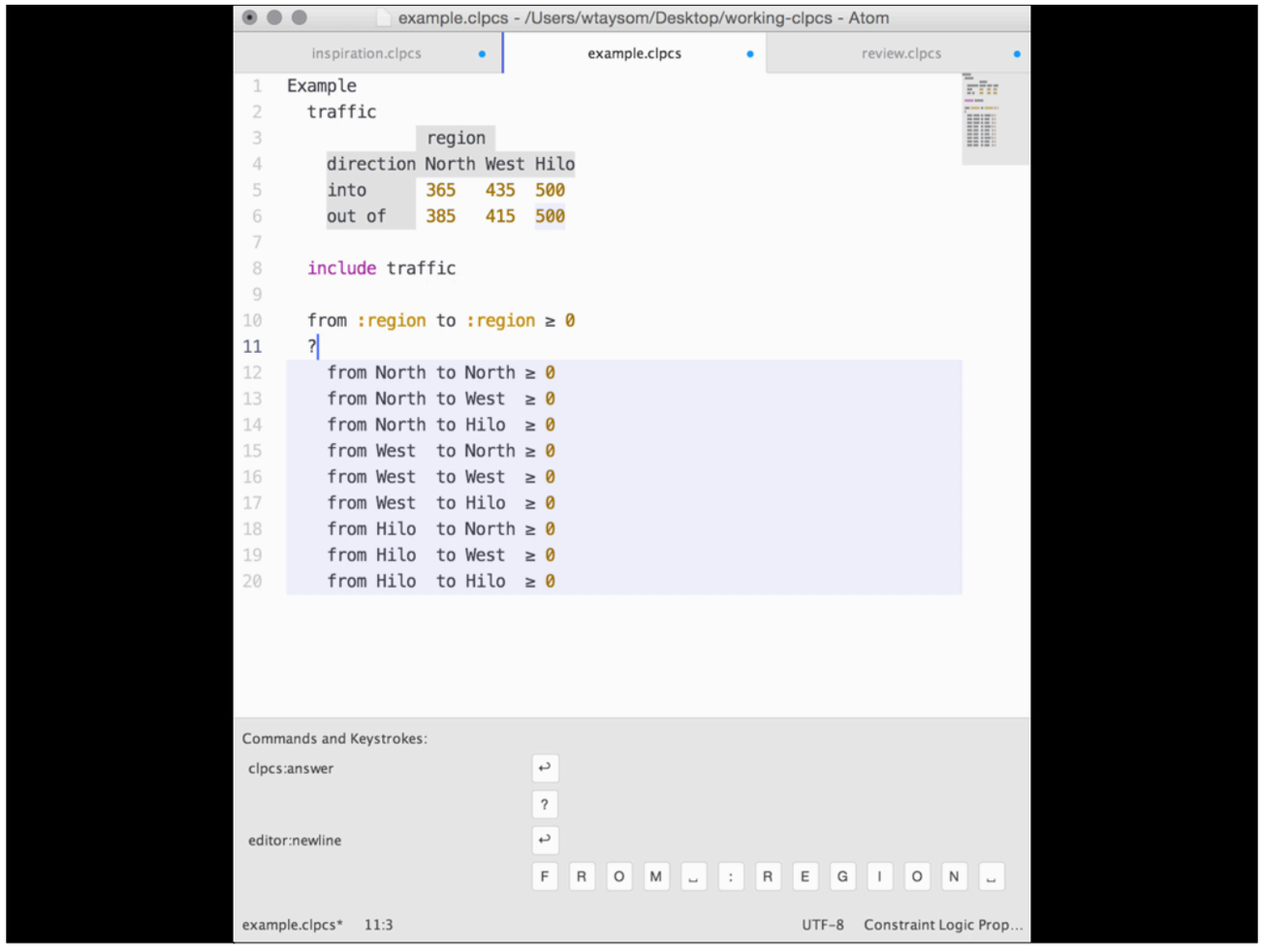
When the question is grounded, we find familiar functional evaluation.

Act II

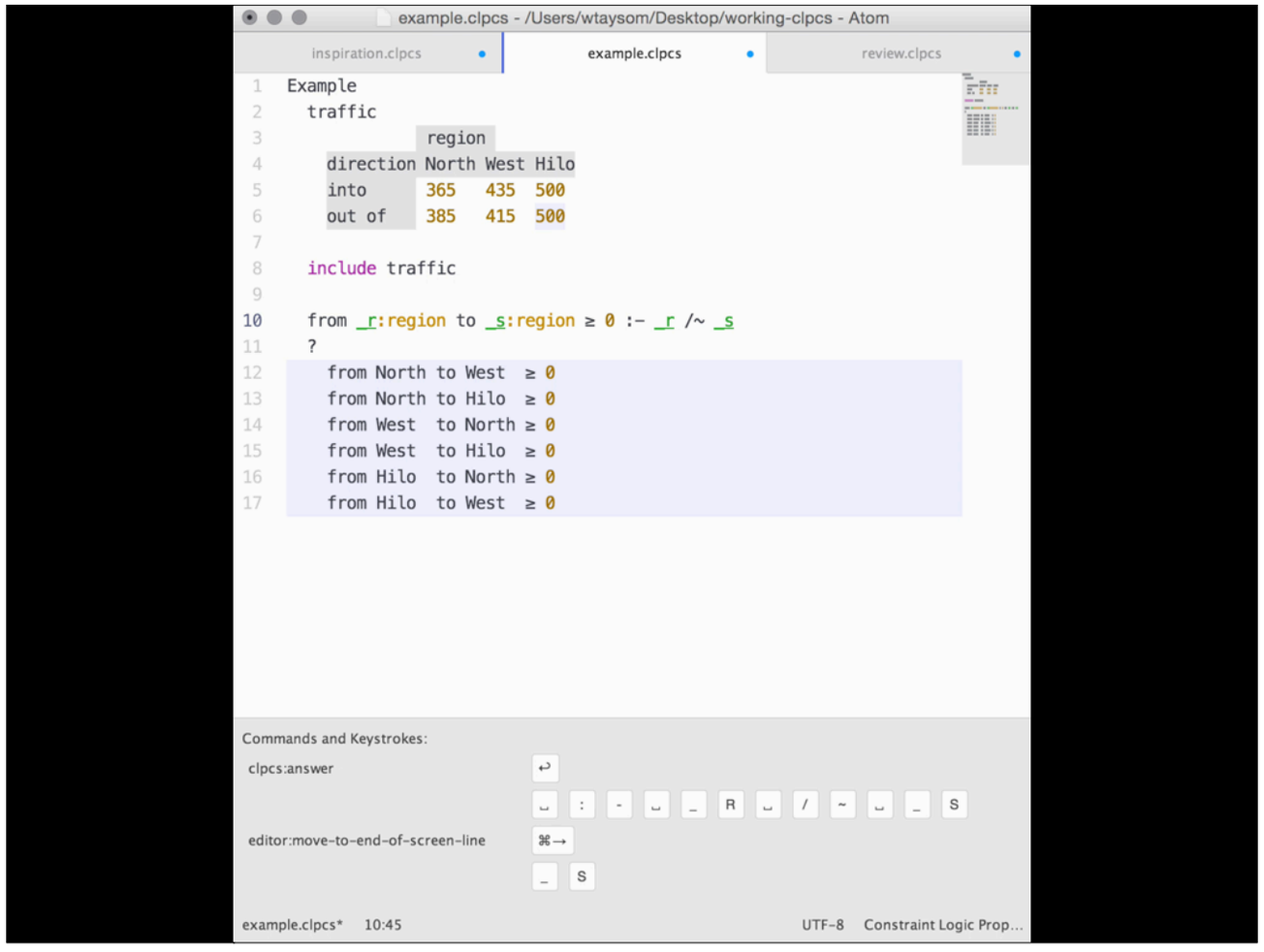
Propagation



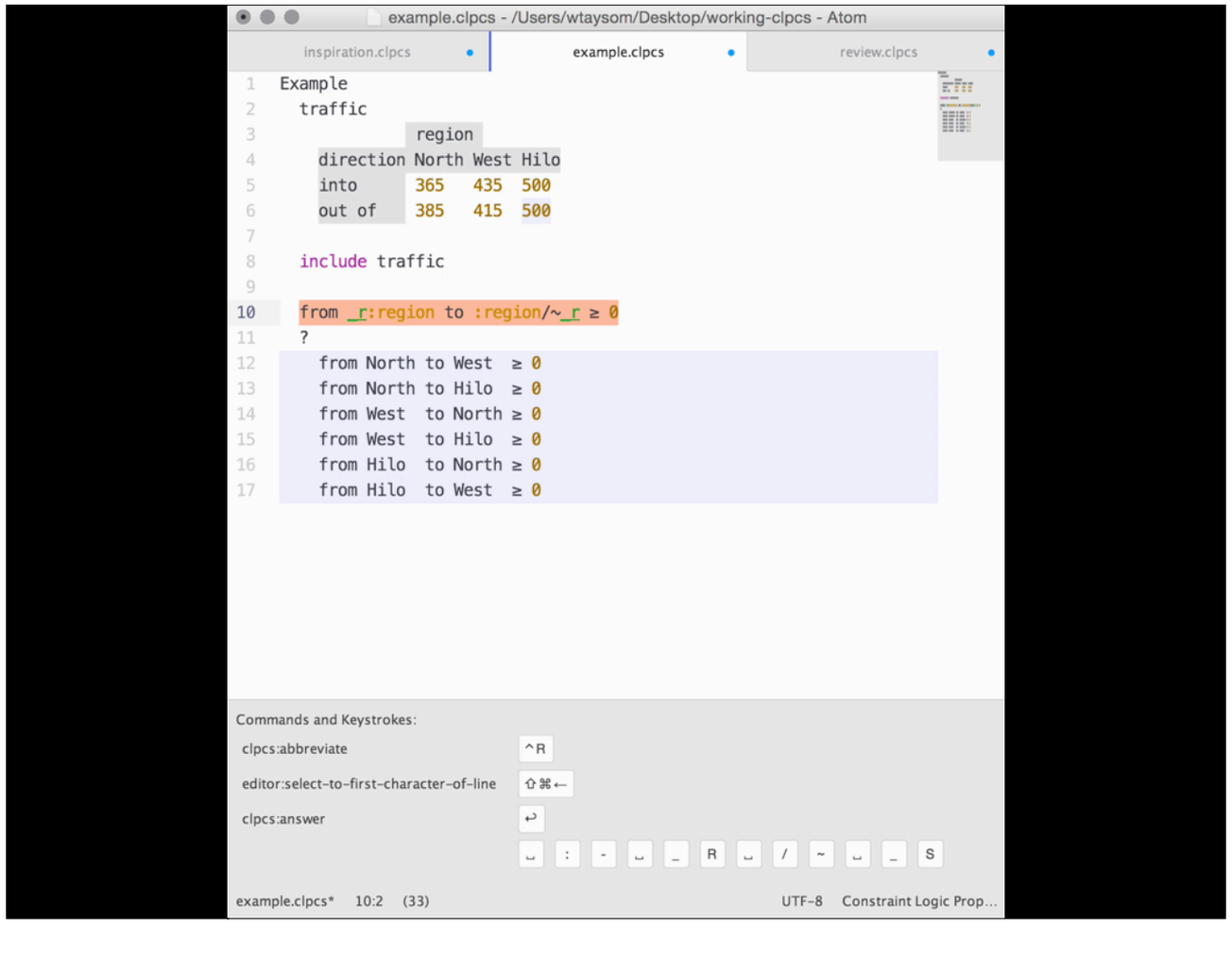
Evaluation is old school; in the future programmers upgrade to inference.



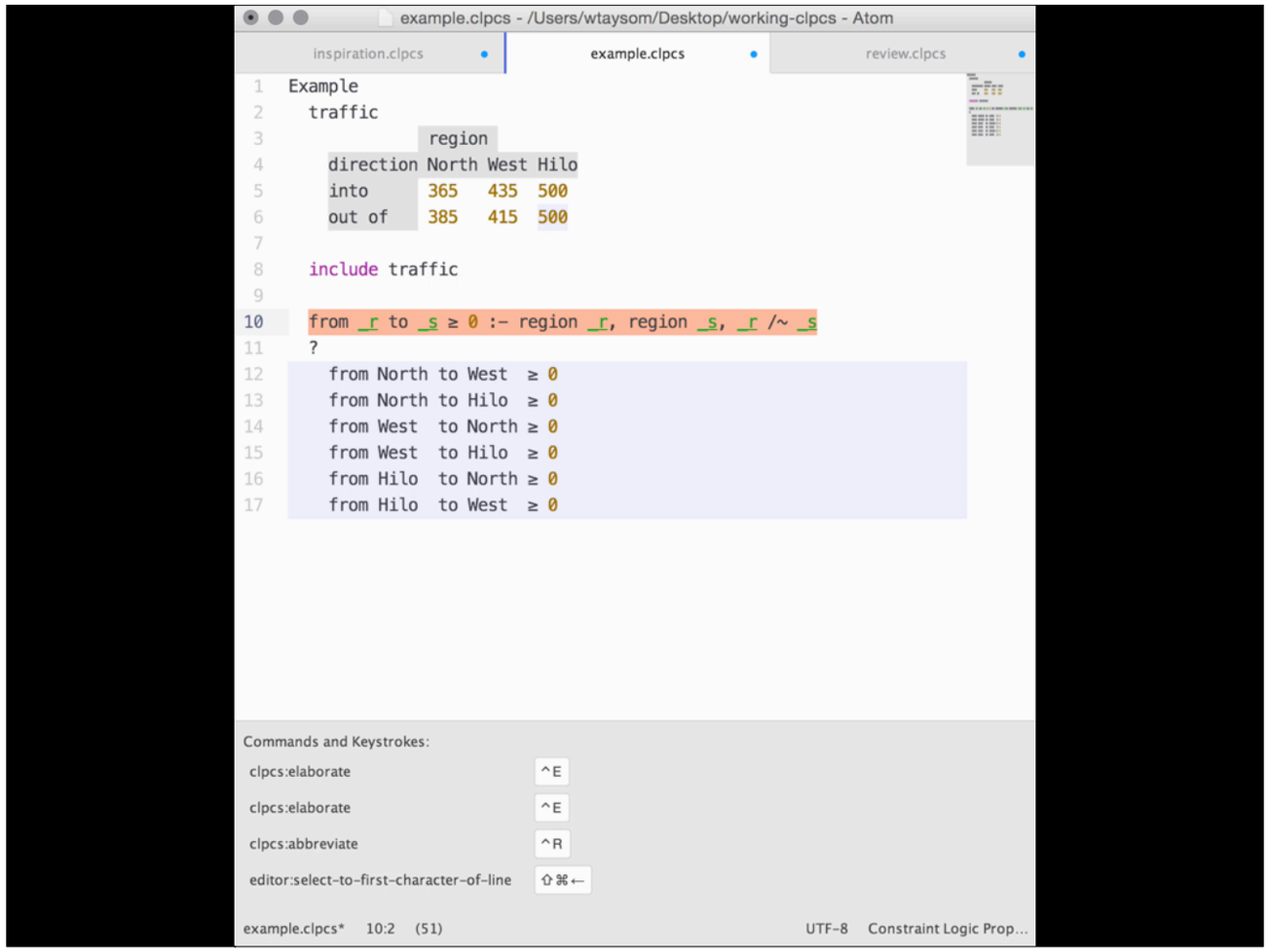
From our data about traffic into and out of regions, we can learn about the traffic going directly from one region to another. Let's see how this could work. The question mark refers to the previous relation. Notice that `from_to_` is defined for every pair of regions. Rather than declaring a type restriction, the colon syntax populates a relation with all values matching a predicate. Too many in this case: traffic only makes sense between regions.



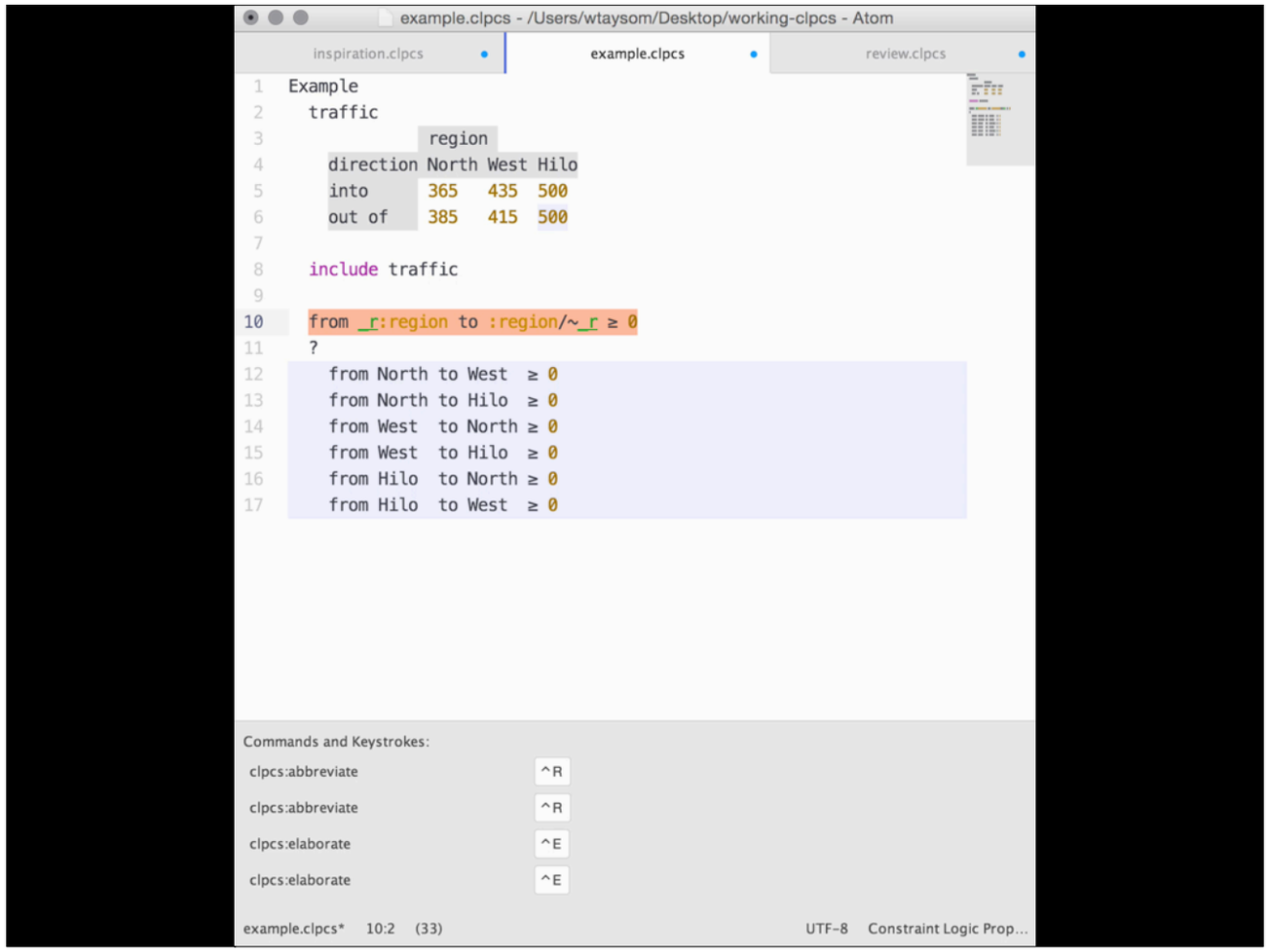
To correctly define the `from_to` relation, we should stipulate that the two regions be different. (Instead of case, I use underscores to mark variables.) At last we see Prolog's famous turnstile notation. The slash-tilde says that the two regions `_r` and `_s` should not unify. So read the whole expression formally as "the traffic from one region to another both exists and is bounded below by zero provided the two regions are distinct."



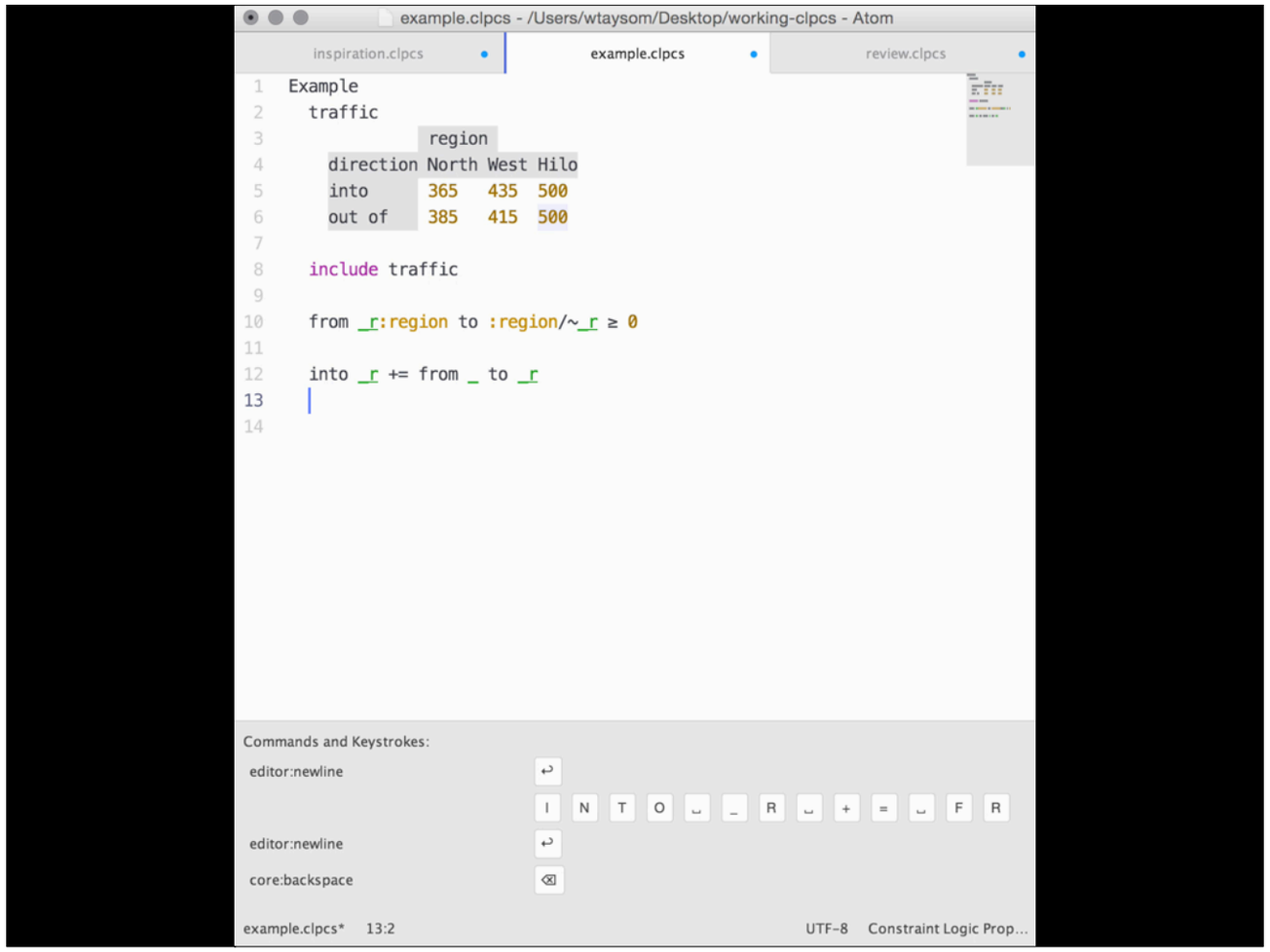
As we did with the grouped and transposed traffic table, we can add syntactic sugar. I feel this makes for a more natural read, "The traffic from one region to another distinct region is bounded below by zero."



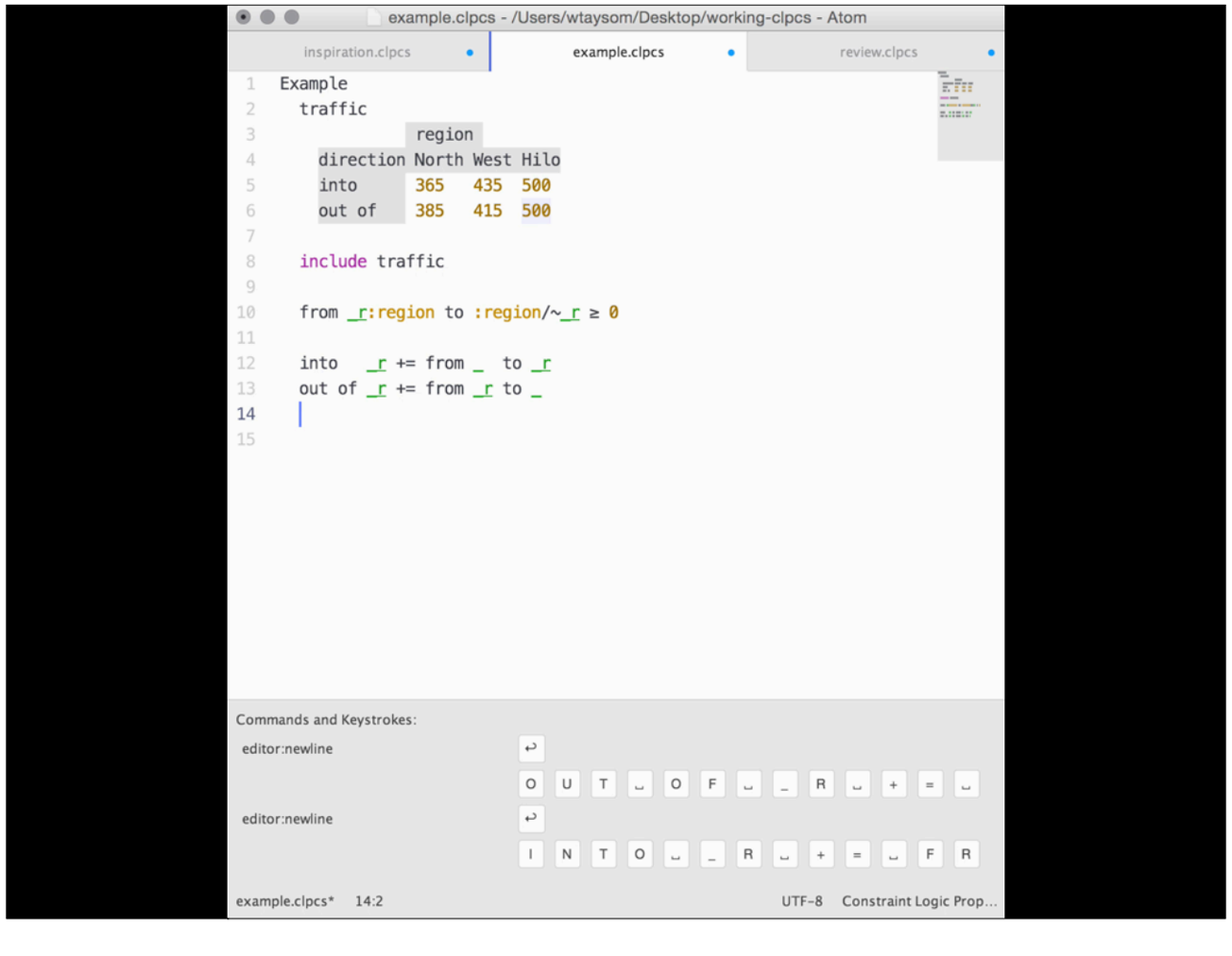
Sifting out syntactic sugar reveals that the colon syntax is shorthand for a predicate in the body.



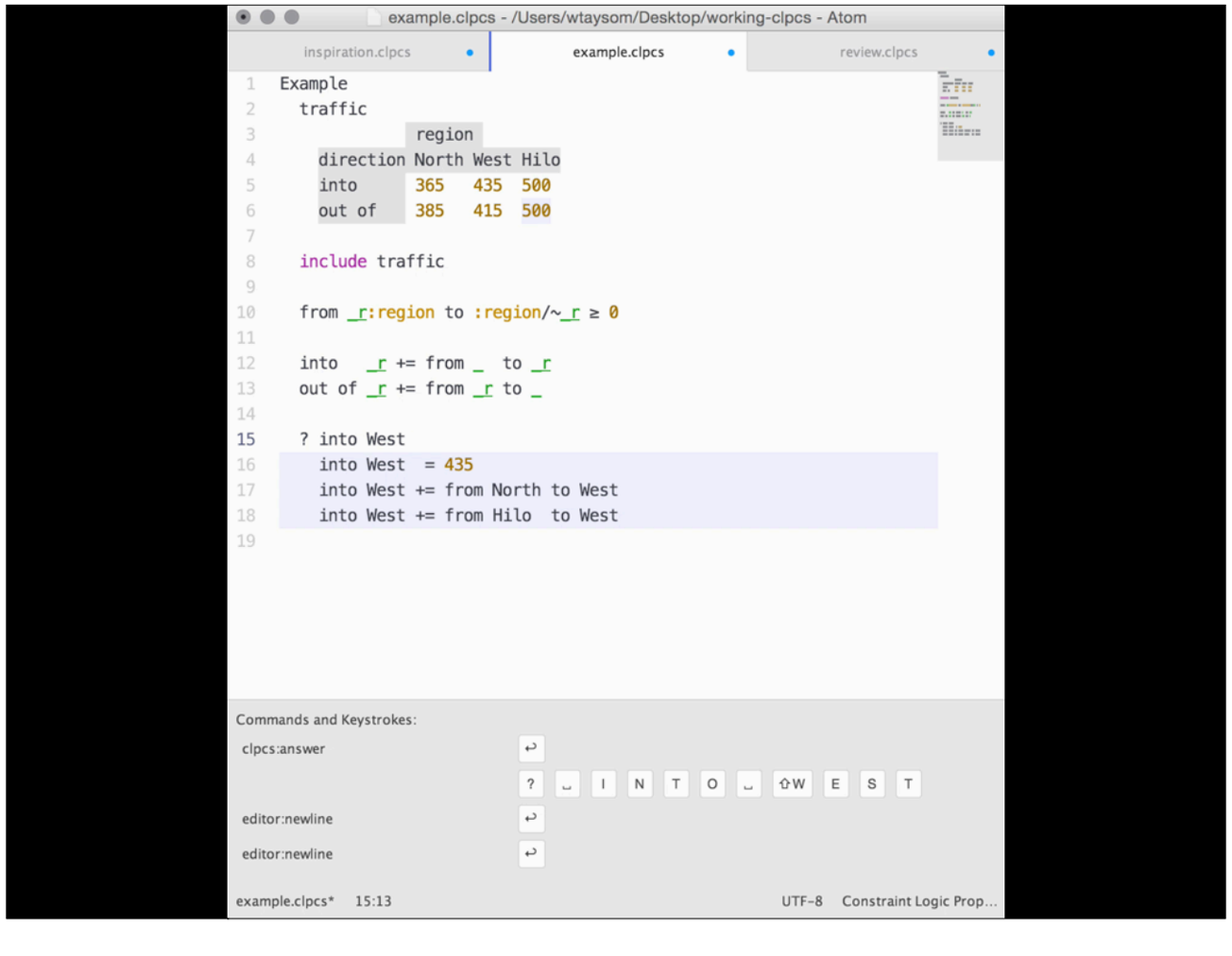
Surface syntax is a view rotated one way or another. To make these syntax transformations bidirectional, one should retain representation metadata such as the variable name `_s` and the order of arguments to the does-not-unify-with operator.



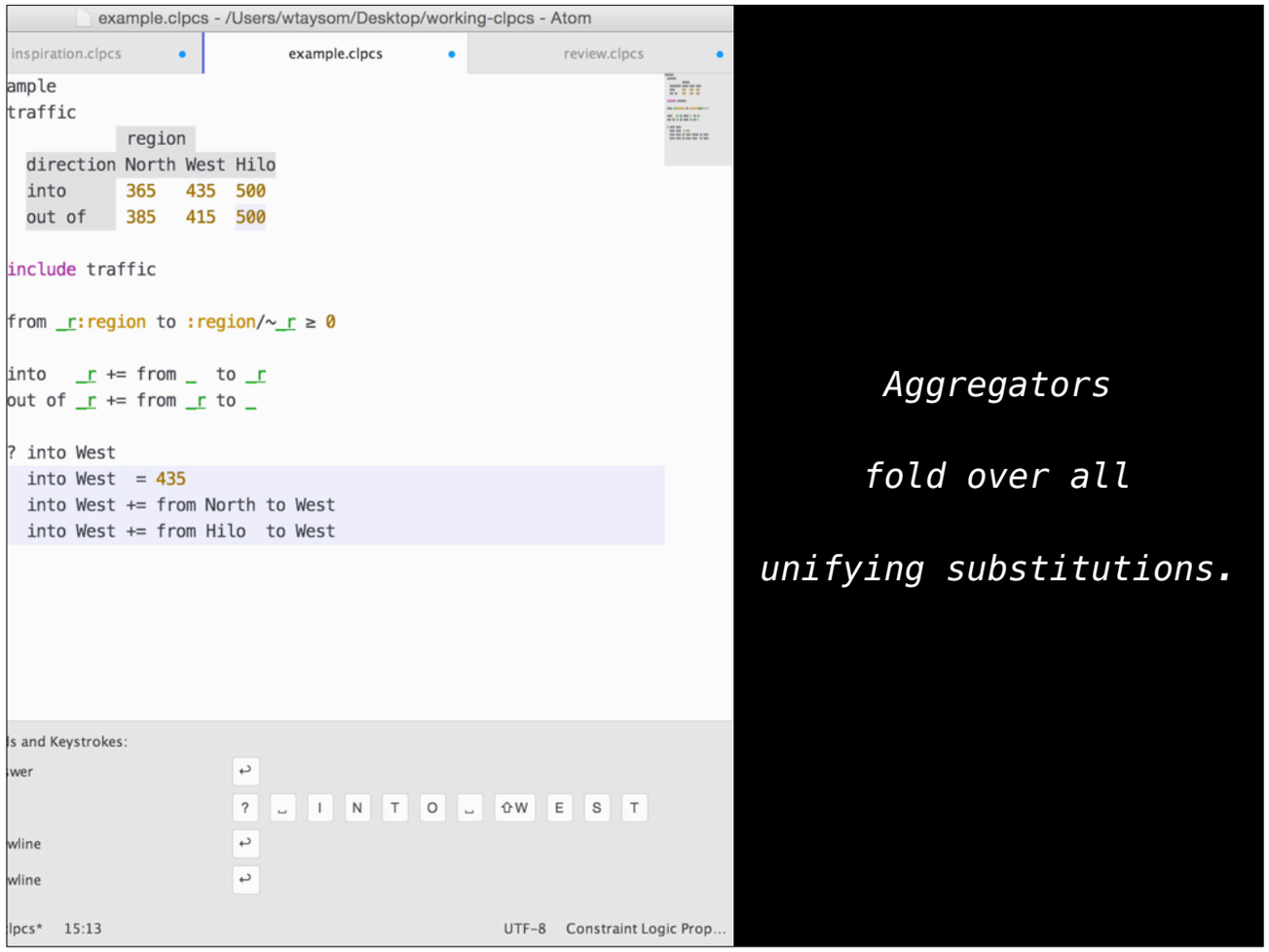
To continue our analysis, associate the `from_to_` relation with our existing traffic info. The traffic into a region is the sum of all traffic from somewhere into the region.



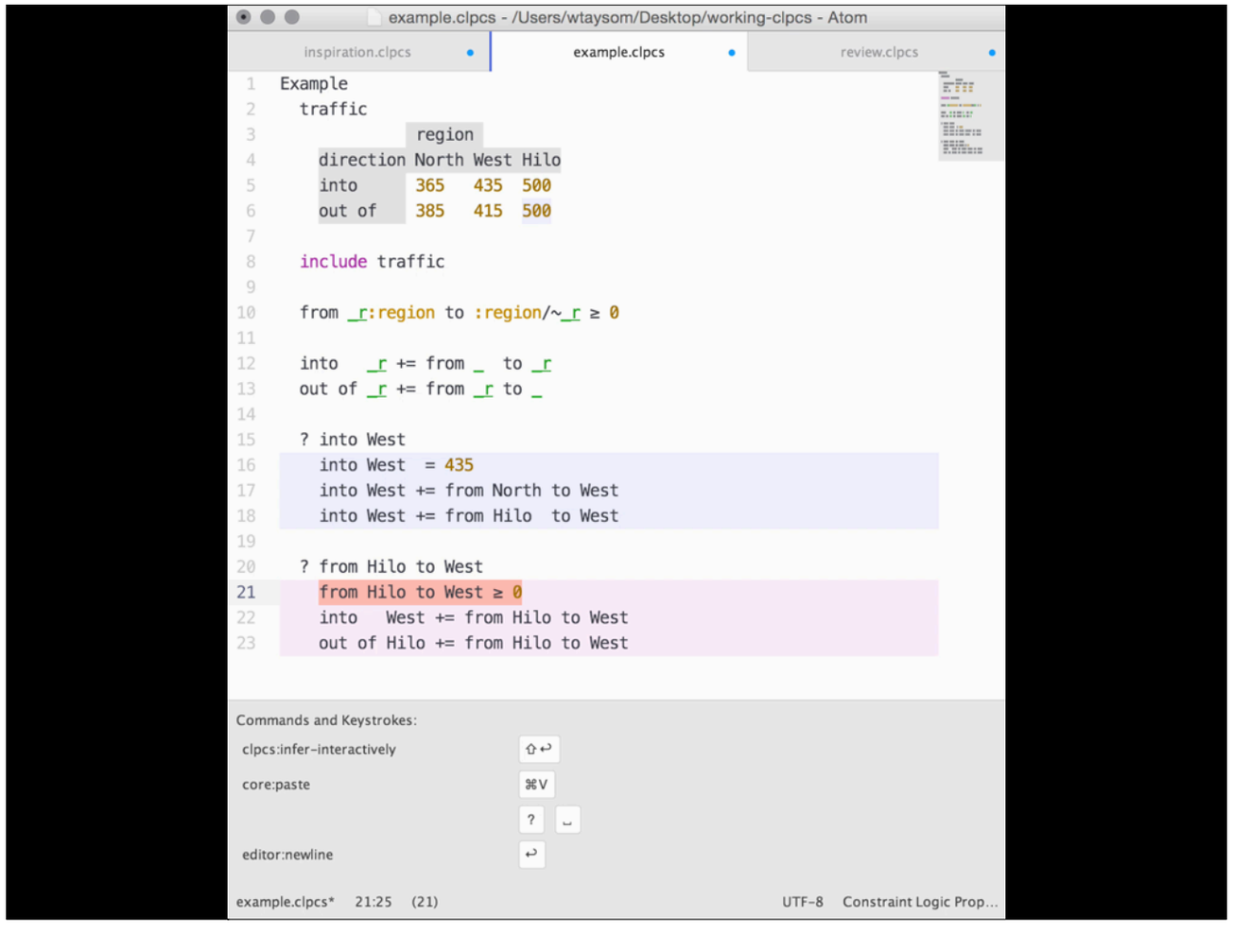
The traffic out of a region is the sum of all traffic from the region to somewhere else.



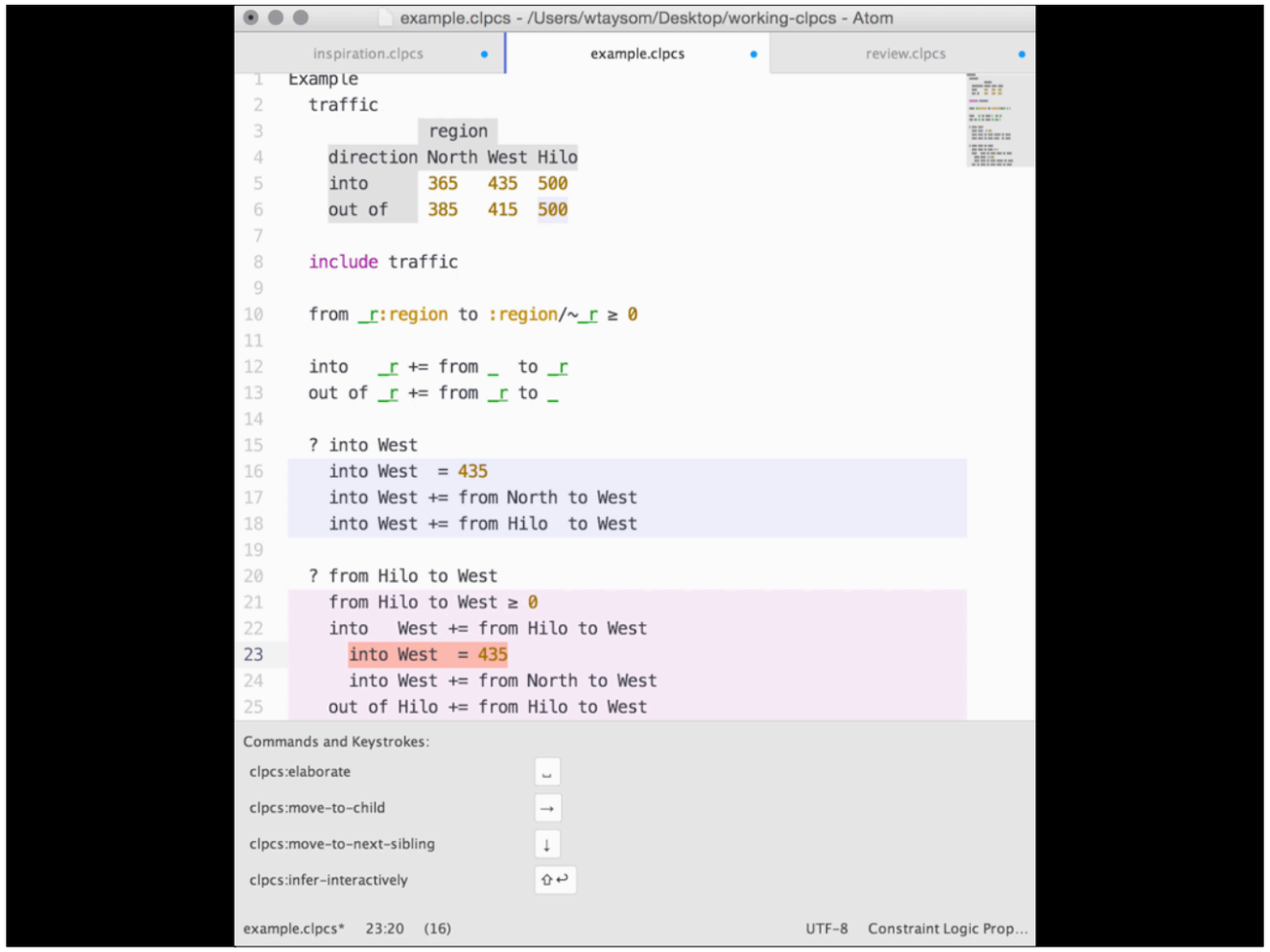
In particular, traffic into the West is the sum of the traffic from the North plus traffic from Hilo.



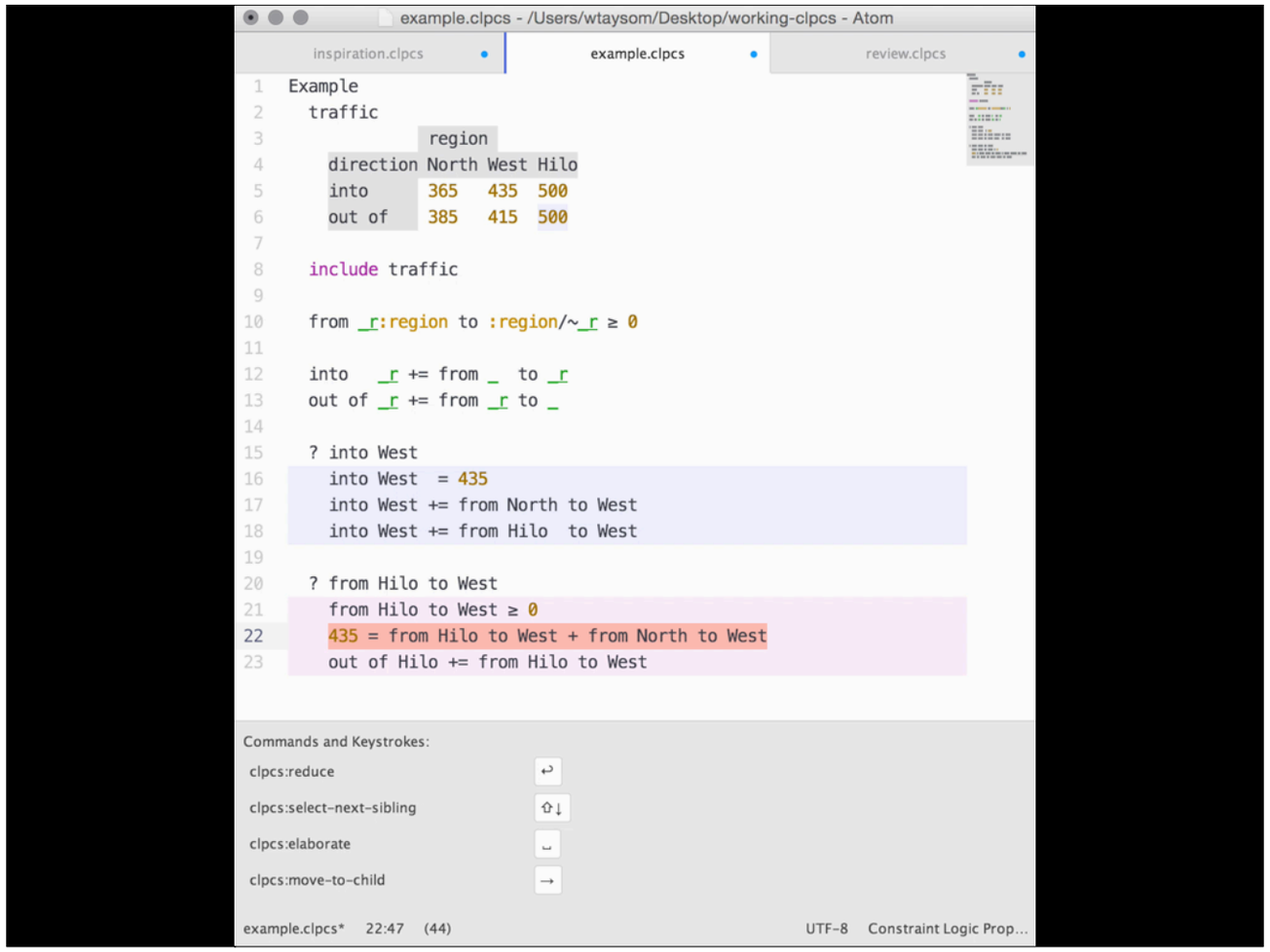
Aggregators fold over all unifying substitutions. They remove the need to explicitly construct a set of items to sum over. Moreover, each line actually defines three sums: one for each region. Three regions in each of two lines makes for six linear equations, six unknowns. A linear dependency ensures that we won't get a unique solution. This makes asking about bounds worthwhile.



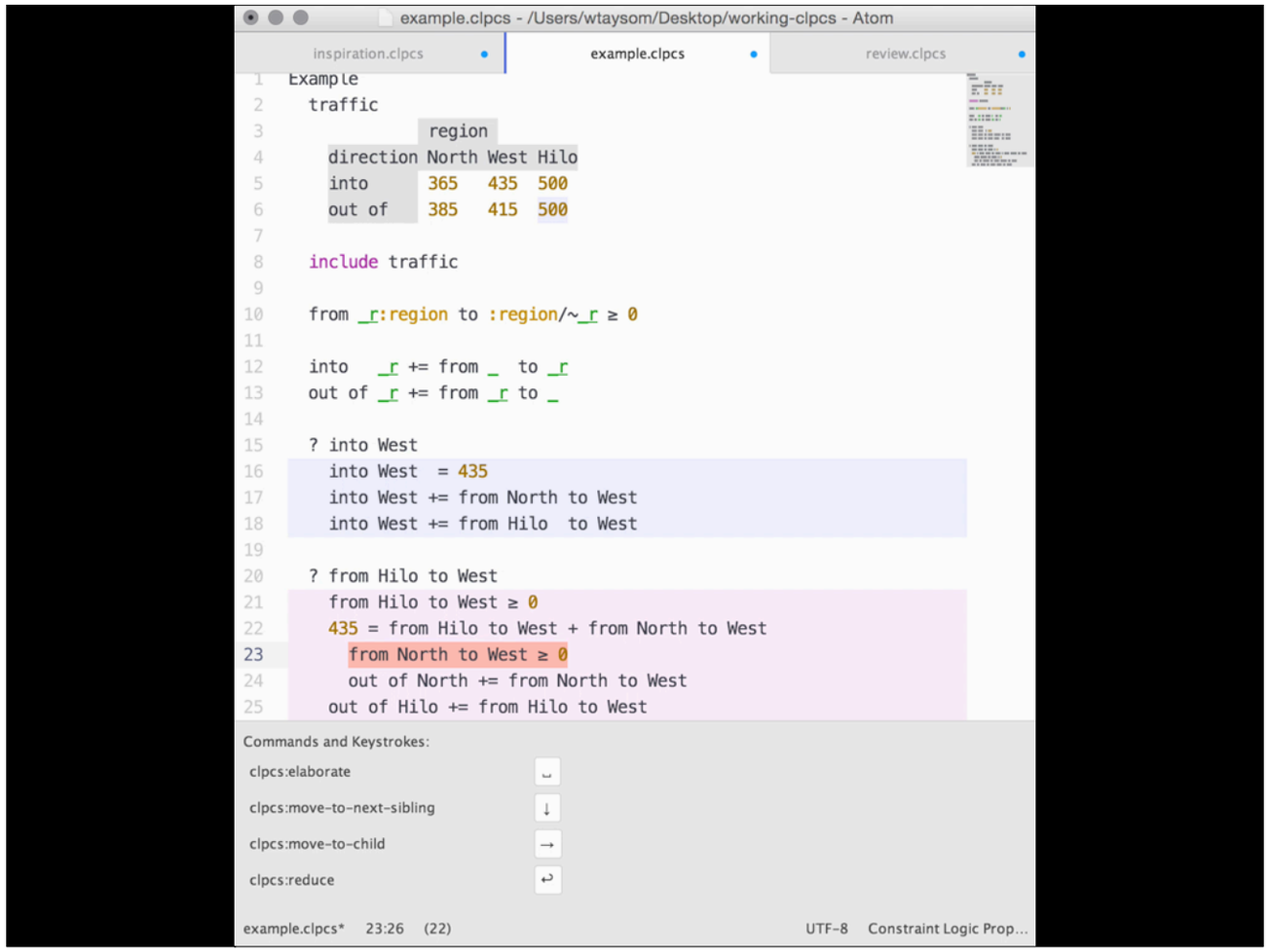
Let's take a look at traffic from Hilo to the West. With shift-enter, we interactively draw inferences.



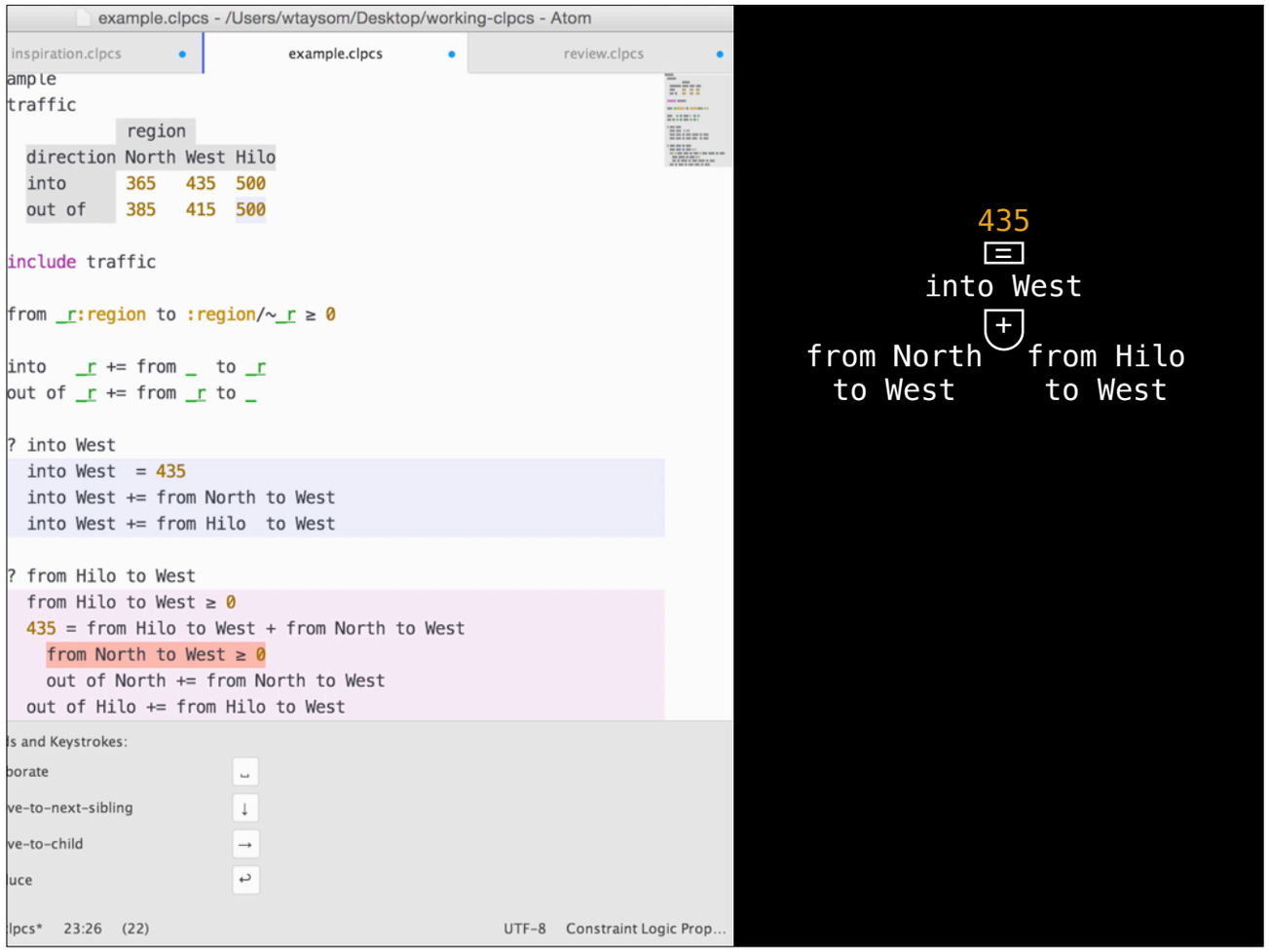
Expand `into West` to see other facts about it.



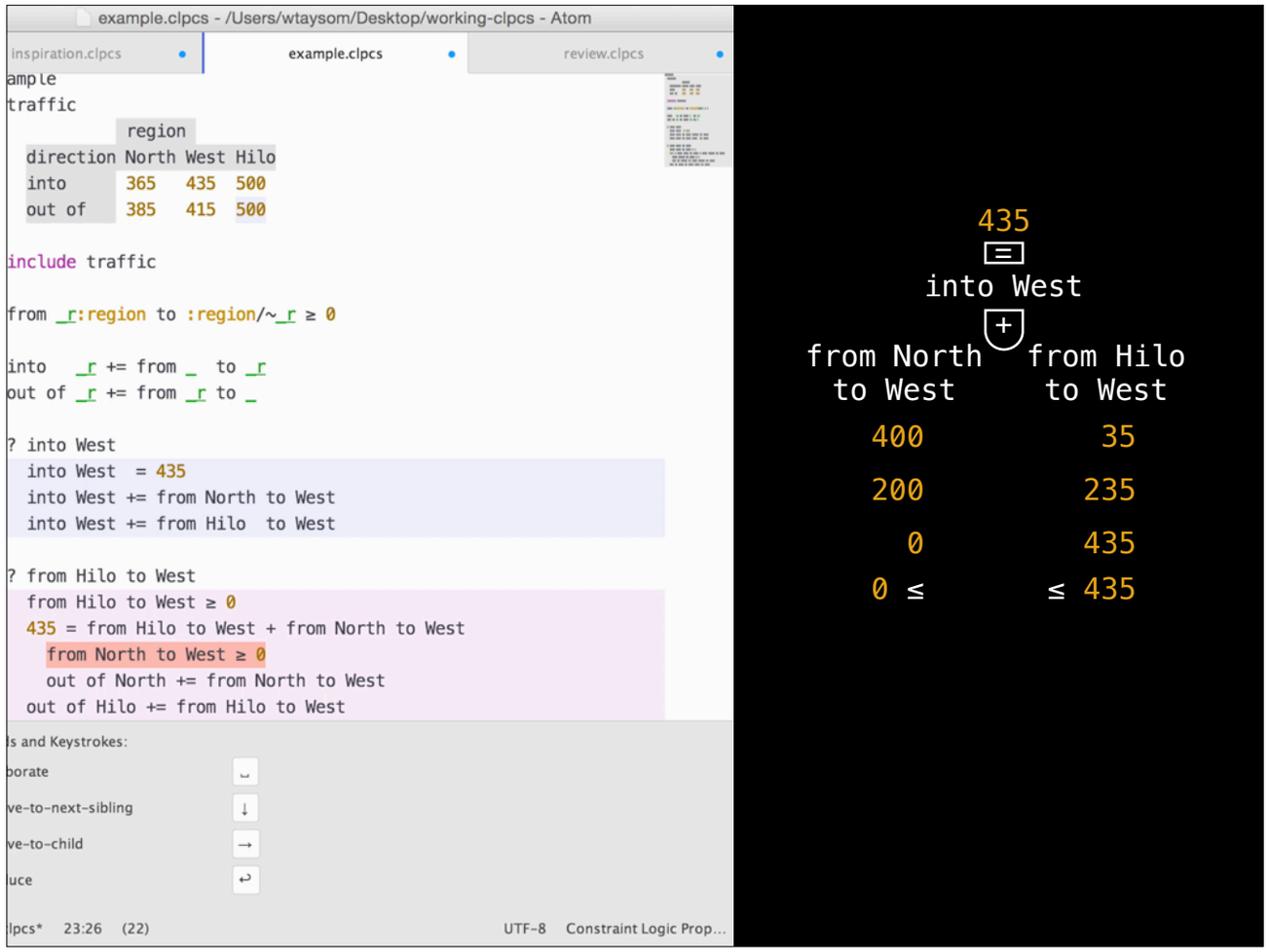
Reduce them into one equation.



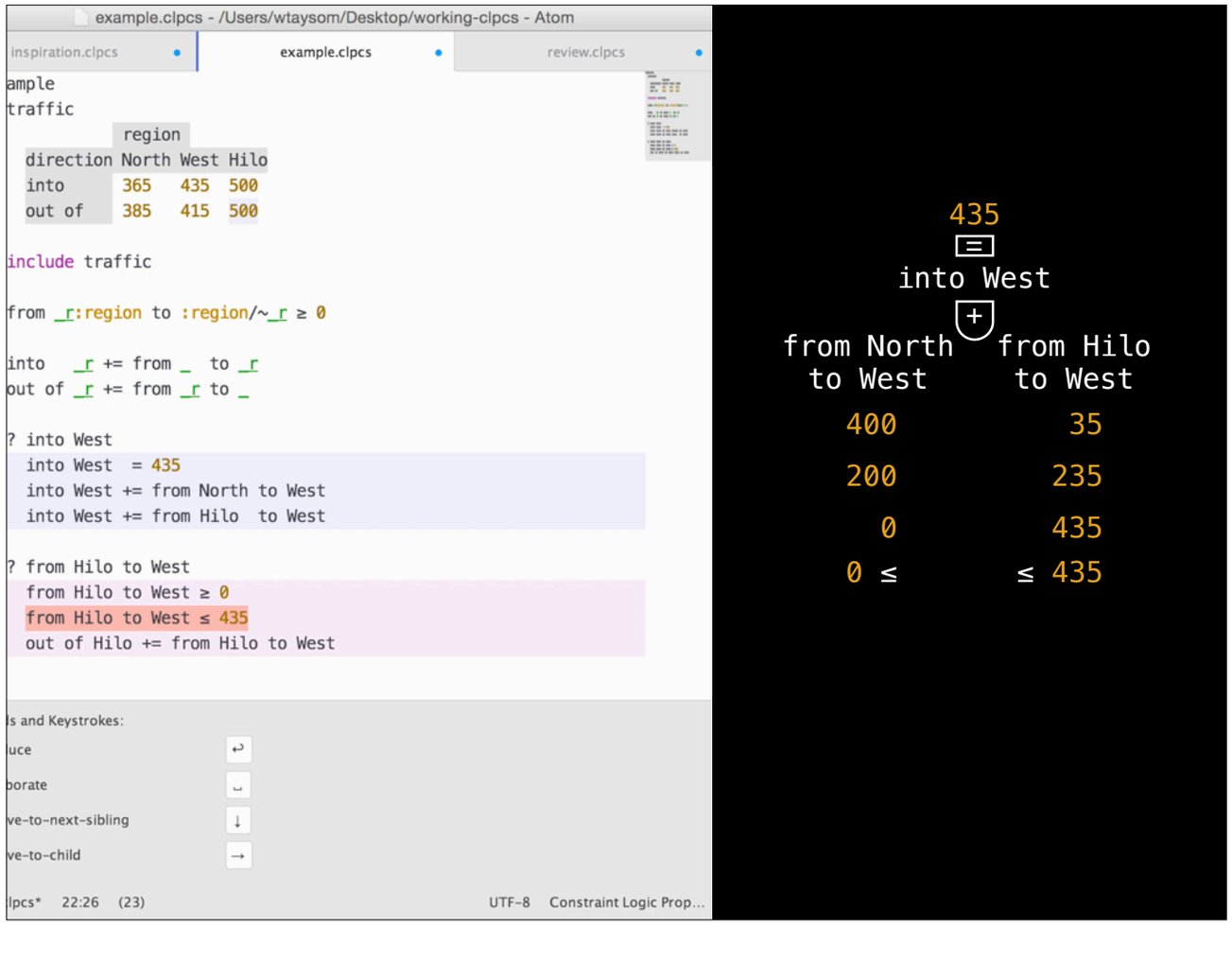
Expand to see more facts. We have arrived at the propagation part of our tour. Remember grounded terms have associated values. Each term is like a slot or cell where you can stick information. Equations relate cells allowing information to spread from one cell to another.



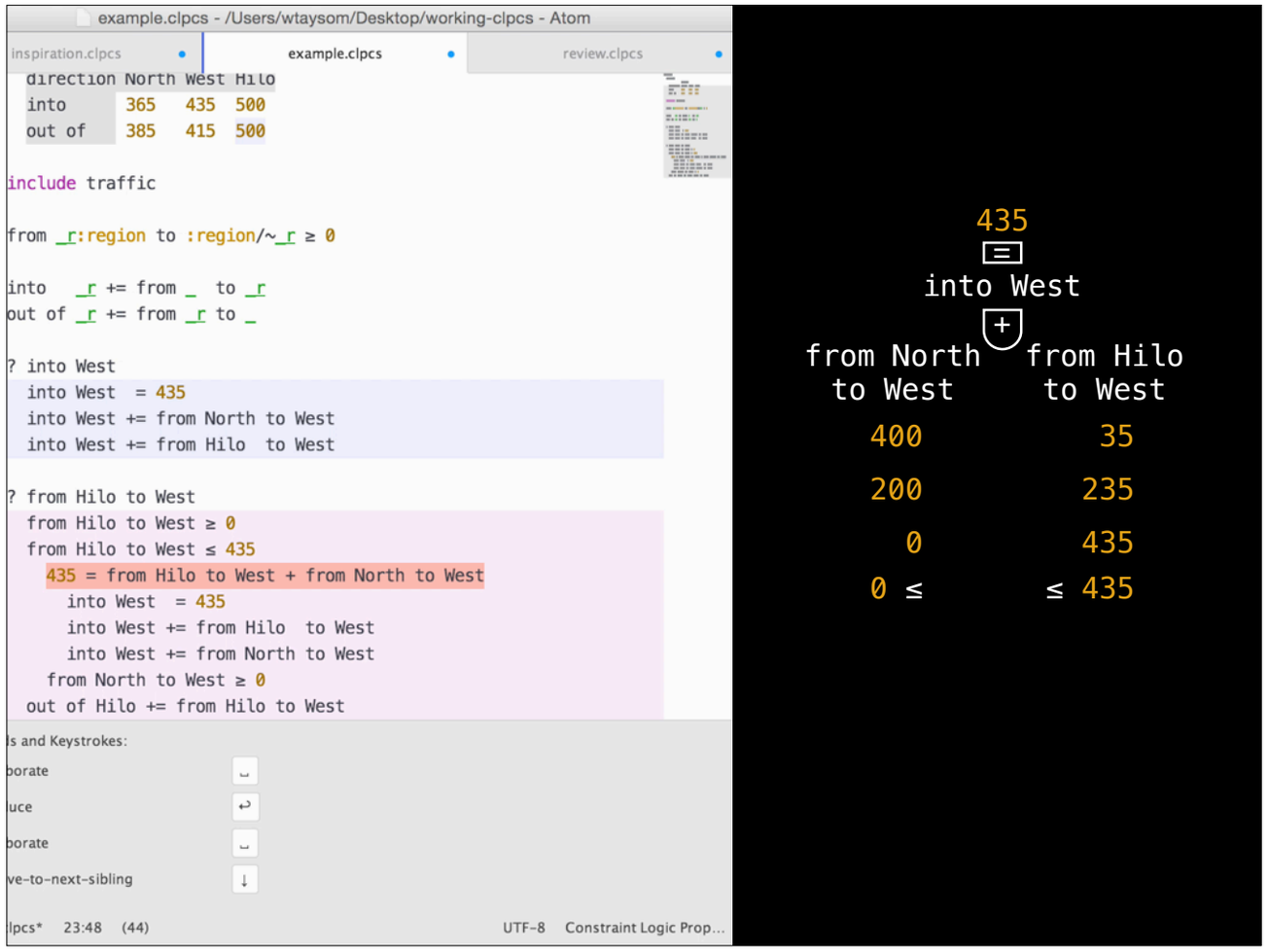
Take a look at this chart showing the relationship of `into West` to other facts. It is equal to 435, and it is the sum of the two `from _ into West` facts.



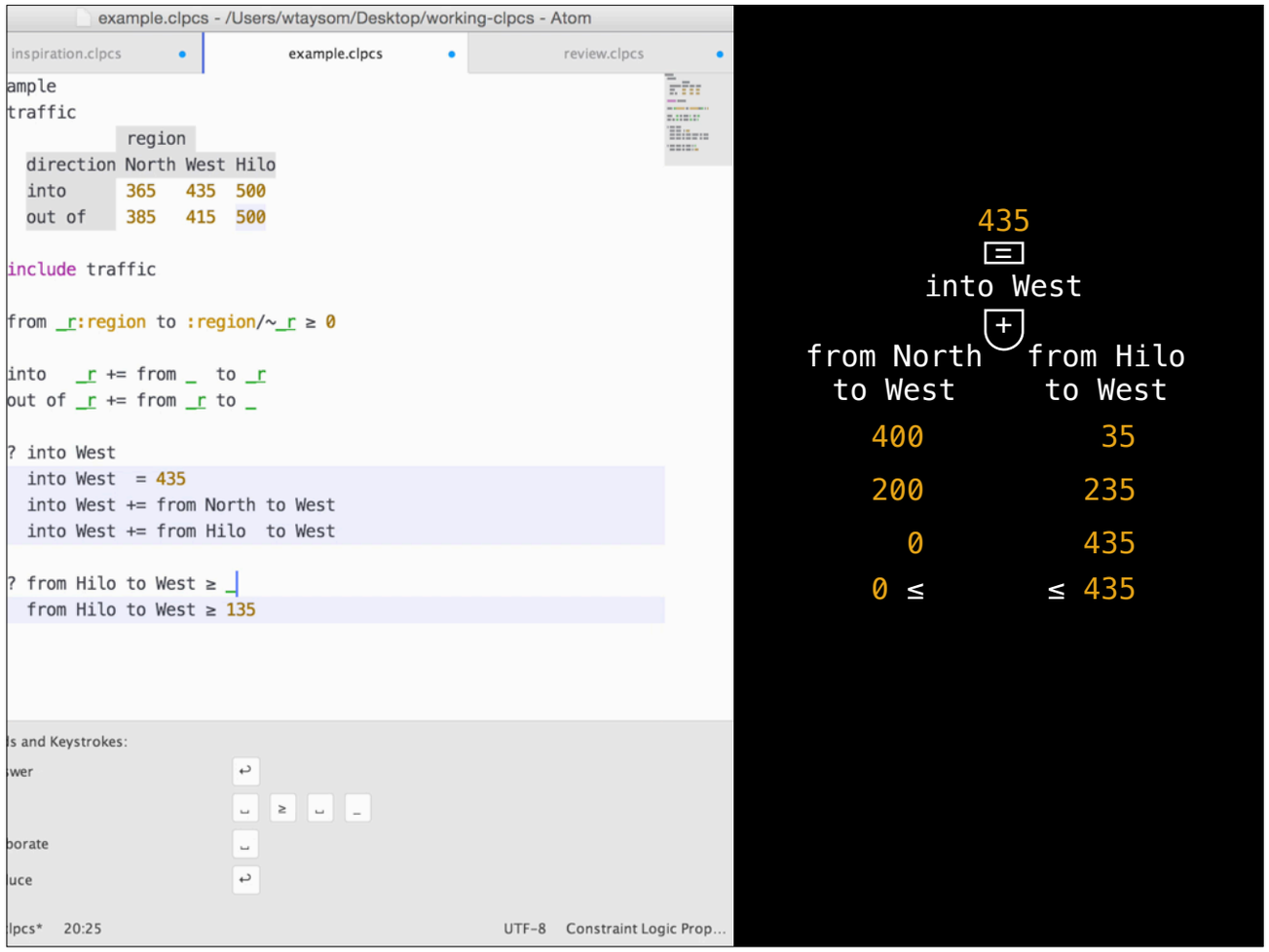
If we learn the value of one of the summands, we can infer the other. Propagation goes both ways. Numbers propagate, and bounds do too.



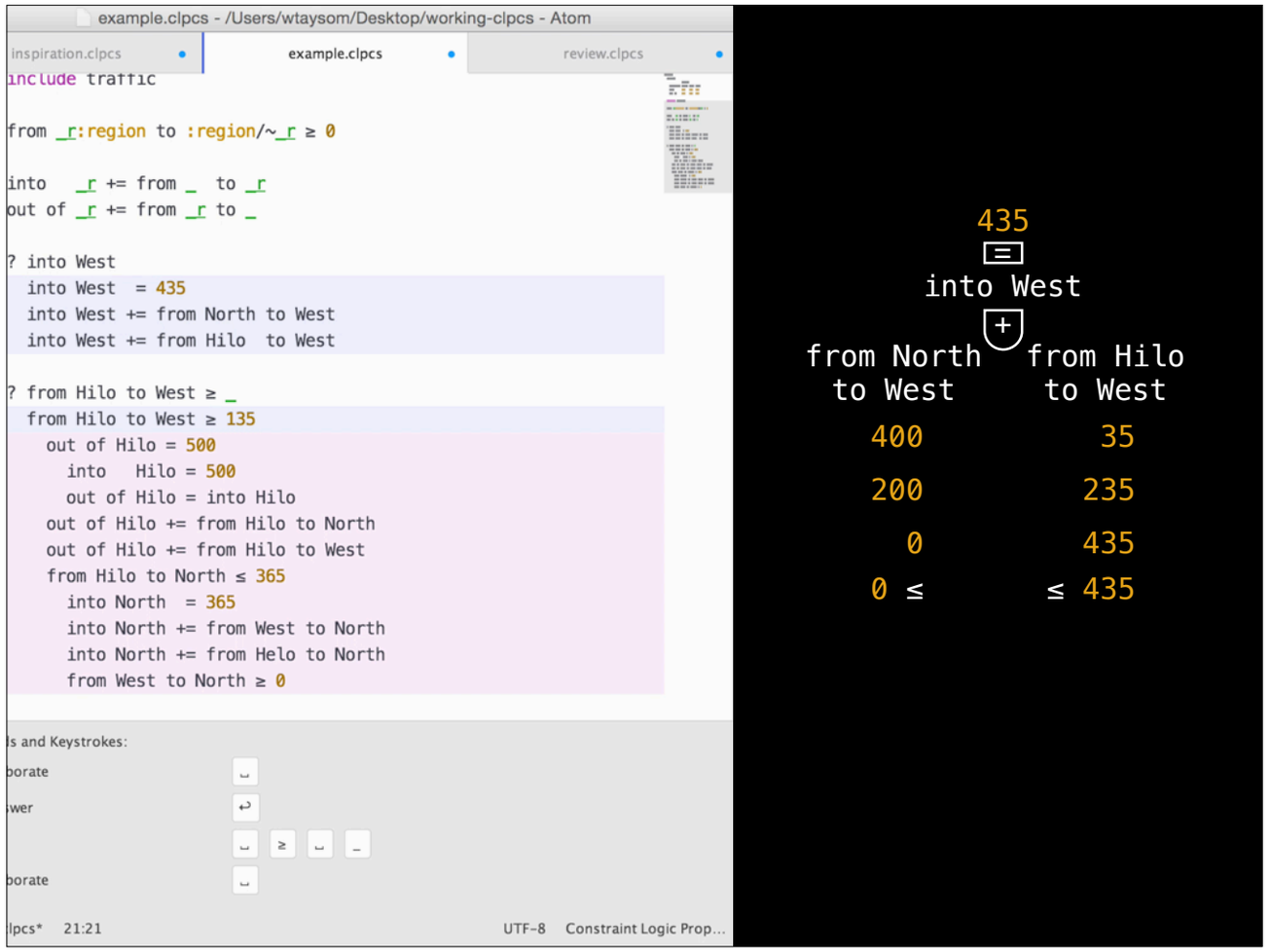
By propagating the inequality across the sum, we set an upper bound. Since addition is monotonic, a lower bound on one summand imposes an upper bound on the other.



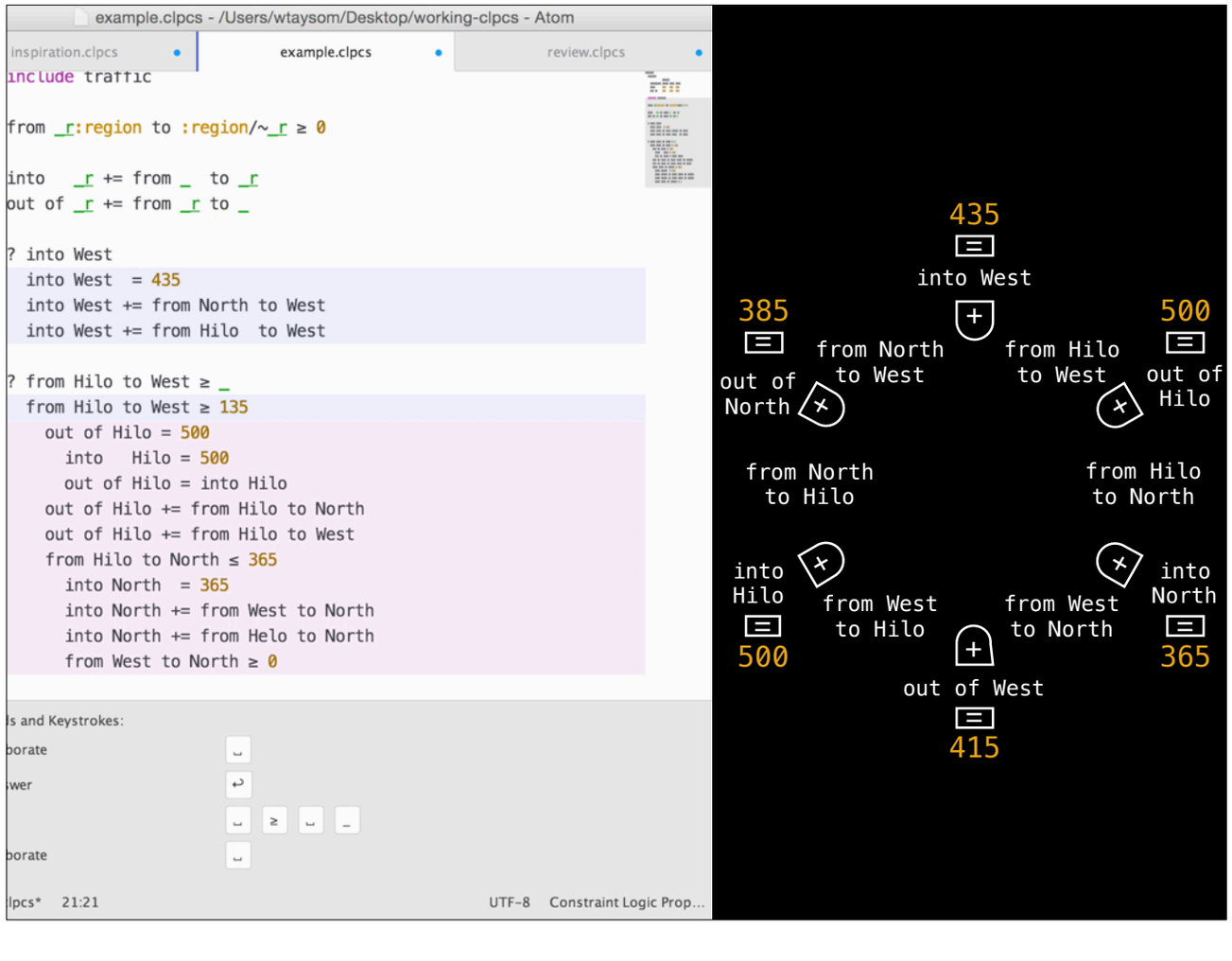
Expand to see the derivation.



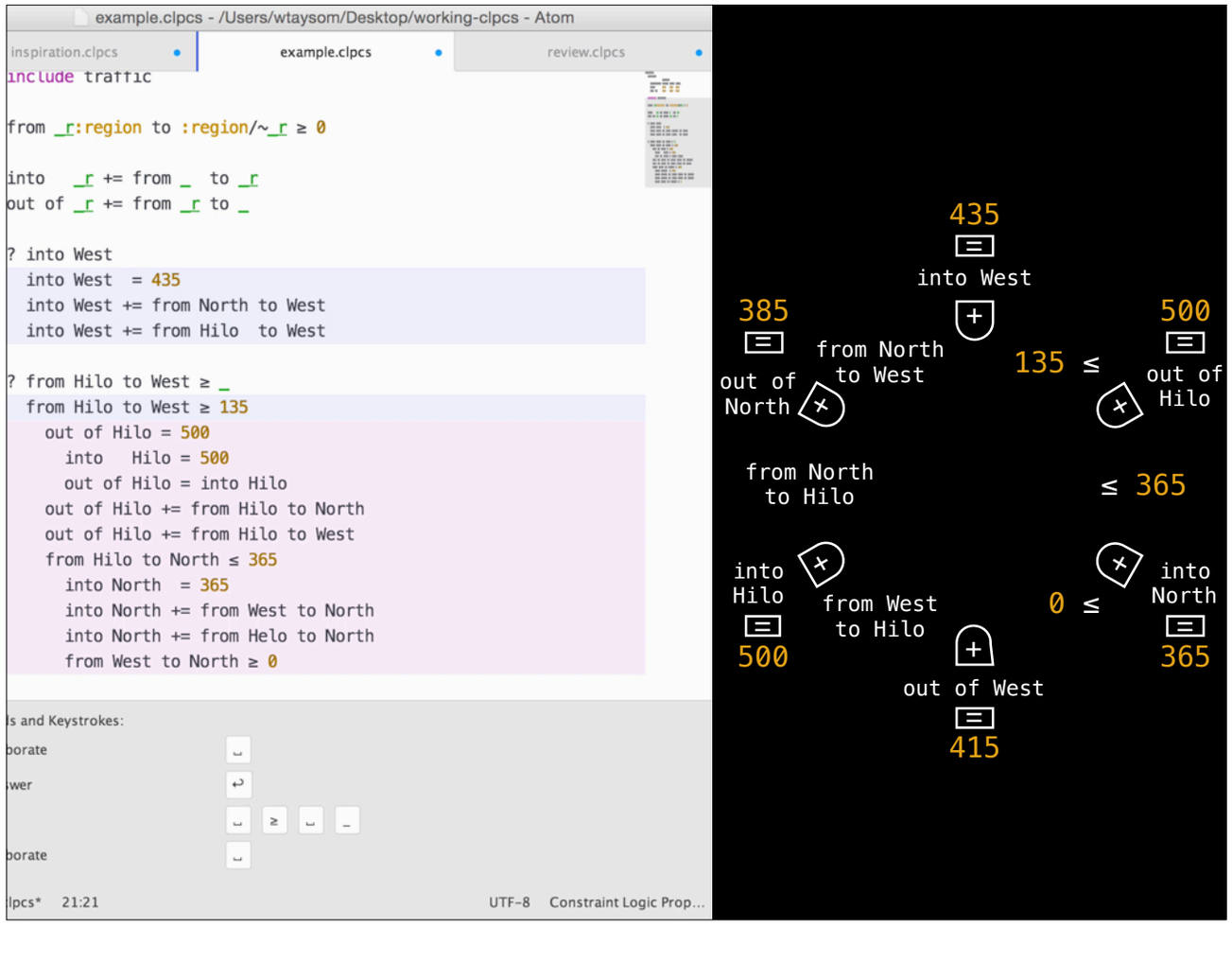
Having found an upper bound for `from Hilo to West`, can we improve its lower bound too?


example.clpcs - /Users/wtaysom/Desktop/working-clpcs - Atom
inspiration.clpcs
example.clpcs
review.clpcs
include traffic
from _r:region to :region/~_r ≥ 0
into _r += from _ to _r
out of _r += from _r to _
? into West
into West = 435
into West += from North to West
into West += from Hilo to West
? from Hilo to West ≥ _
from Hilo to West ≥ 135
out of Hilo = 500
into Hilo = 500
out of Hilo = into Hilo
out of Hilo += from Hilo to North
out of Hilo += from Hilo to West
from Hilo to North ≤ 365
into North = 365
into North += from West to North
into North += from Helo to North
from West to North ≥ 0
ls and Keystrokes:
borate
wer
borate
lpcs* 21:21
UTF-8 Constraint Logic Prop...
435
into West
from North to West
from Hilo to West
400
35
200
235
0
435
0 ≤
≤ 435

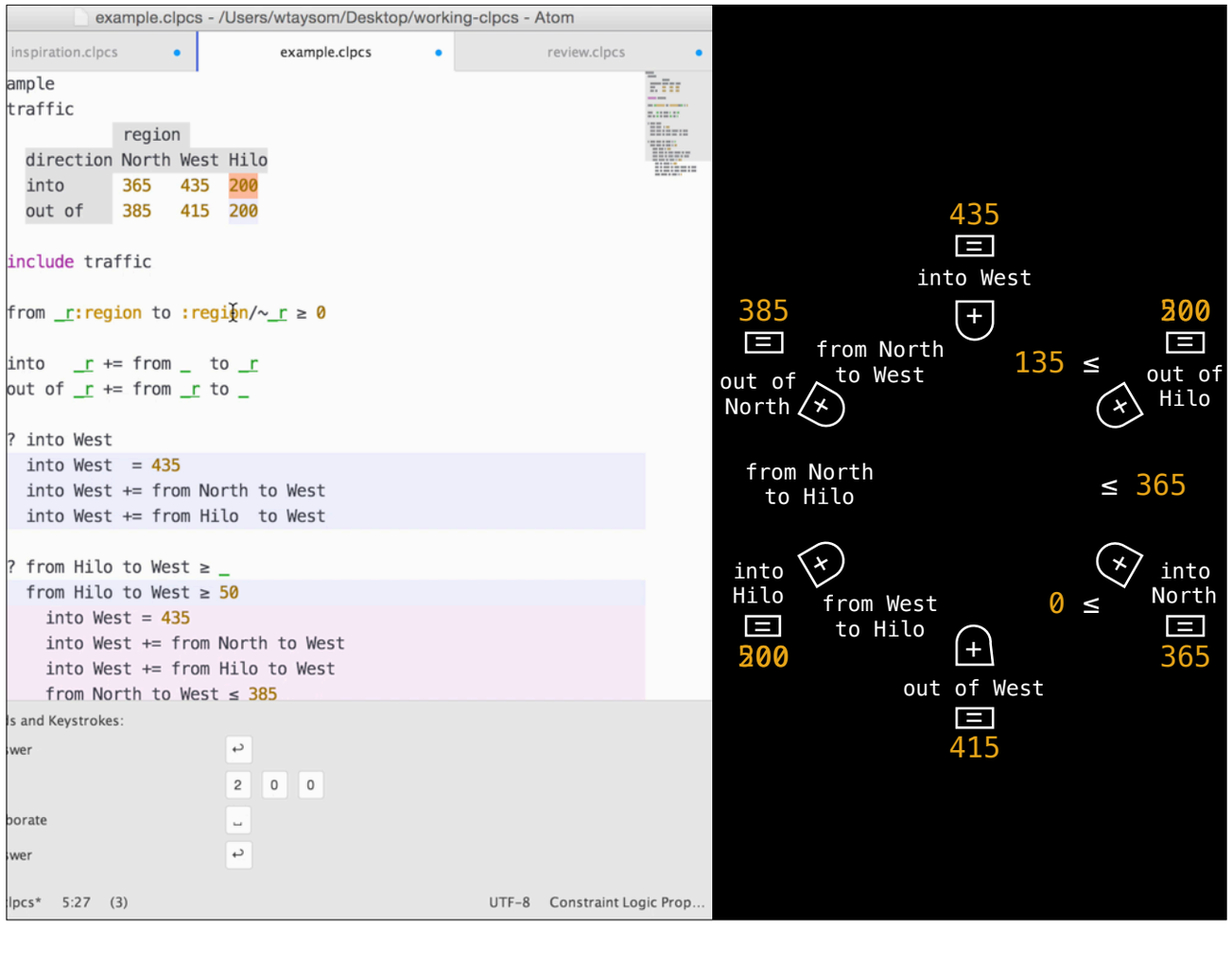
How?



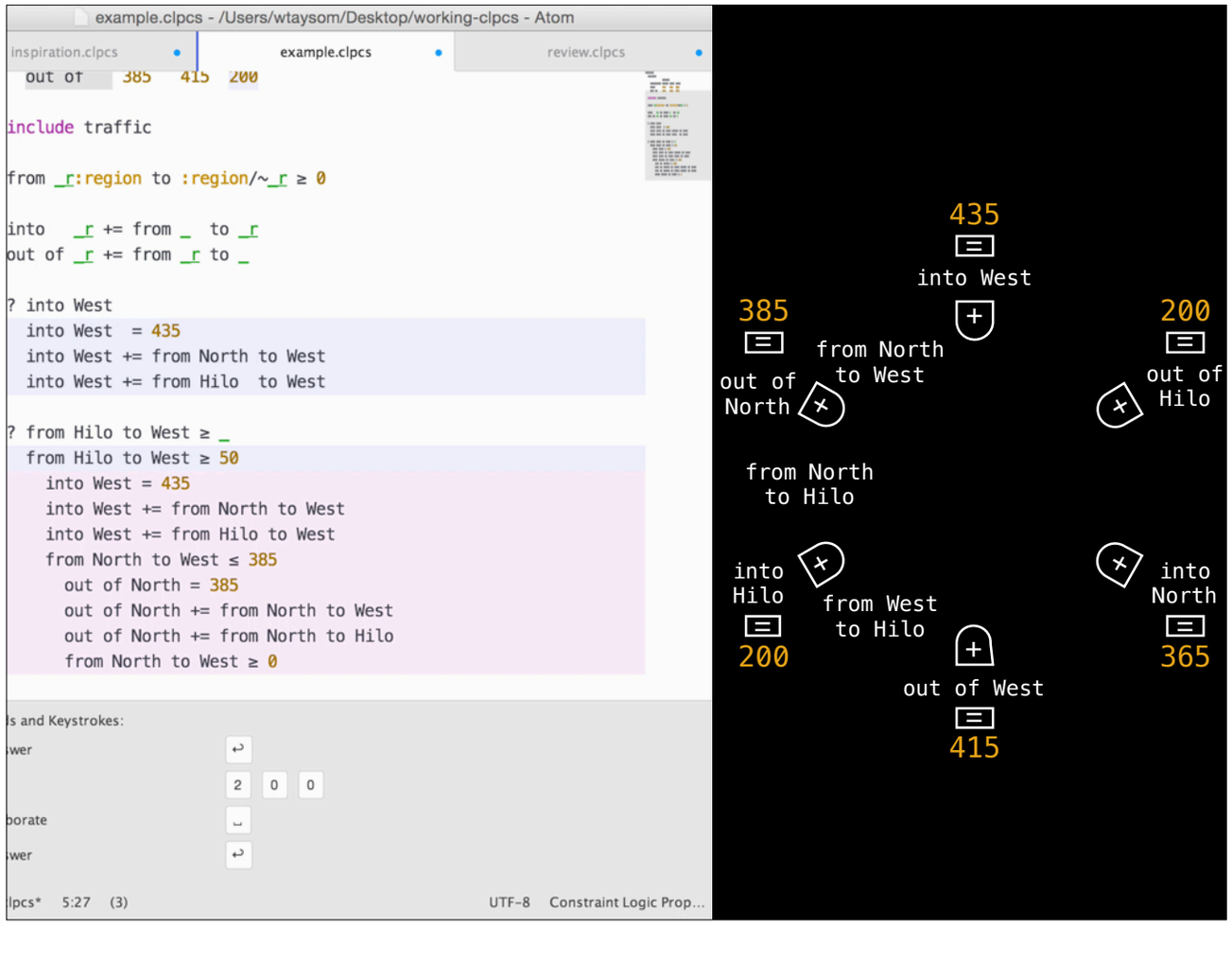
Let's add these facts to our chart: `out of Hilo`, `into North`, and the rest.



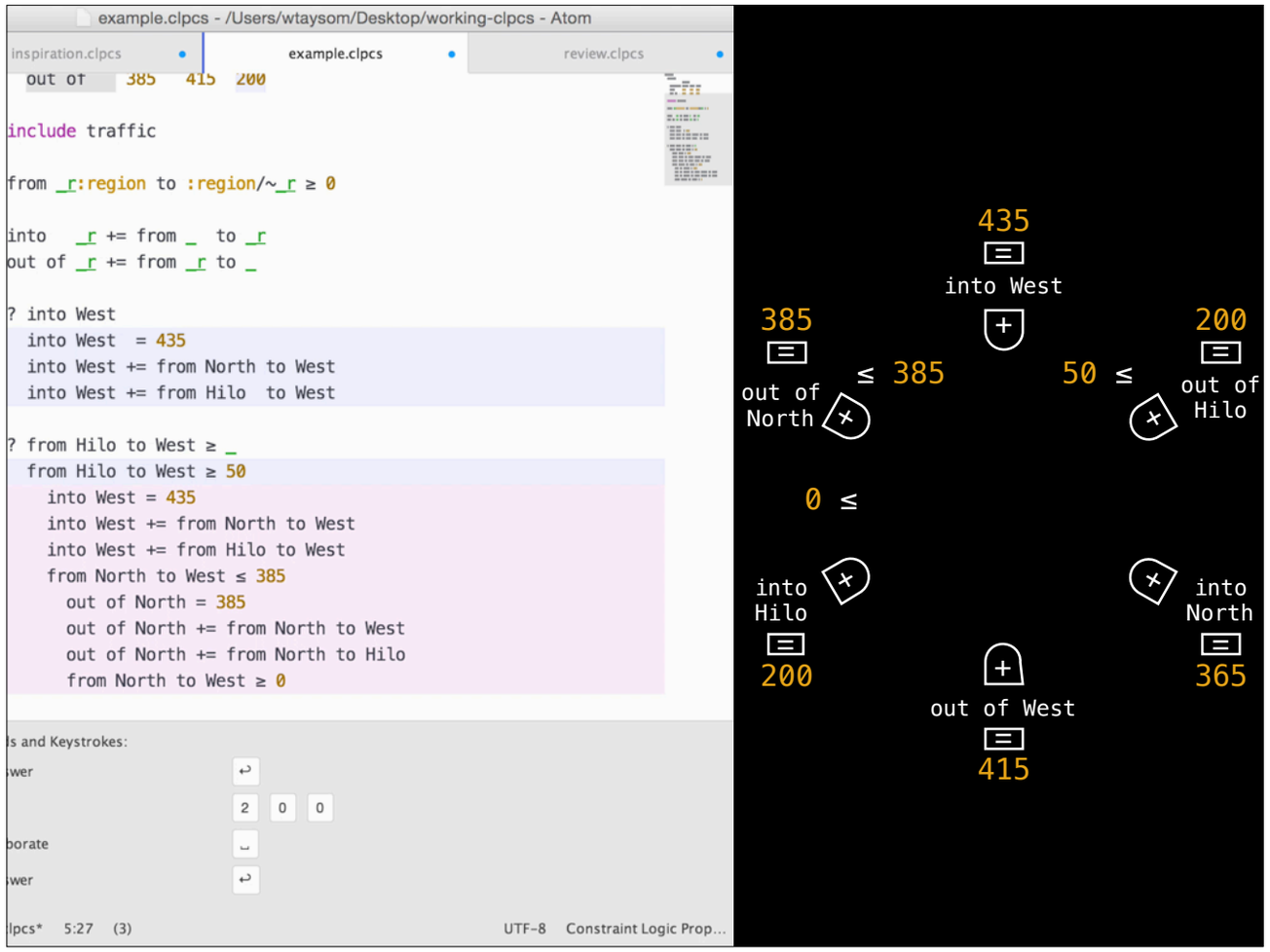
Where does the lower bound of 135 come from? It follows from there being an upper bound on `from Hilo to North`, which in turn comes from our zero bound on `from West to North`.



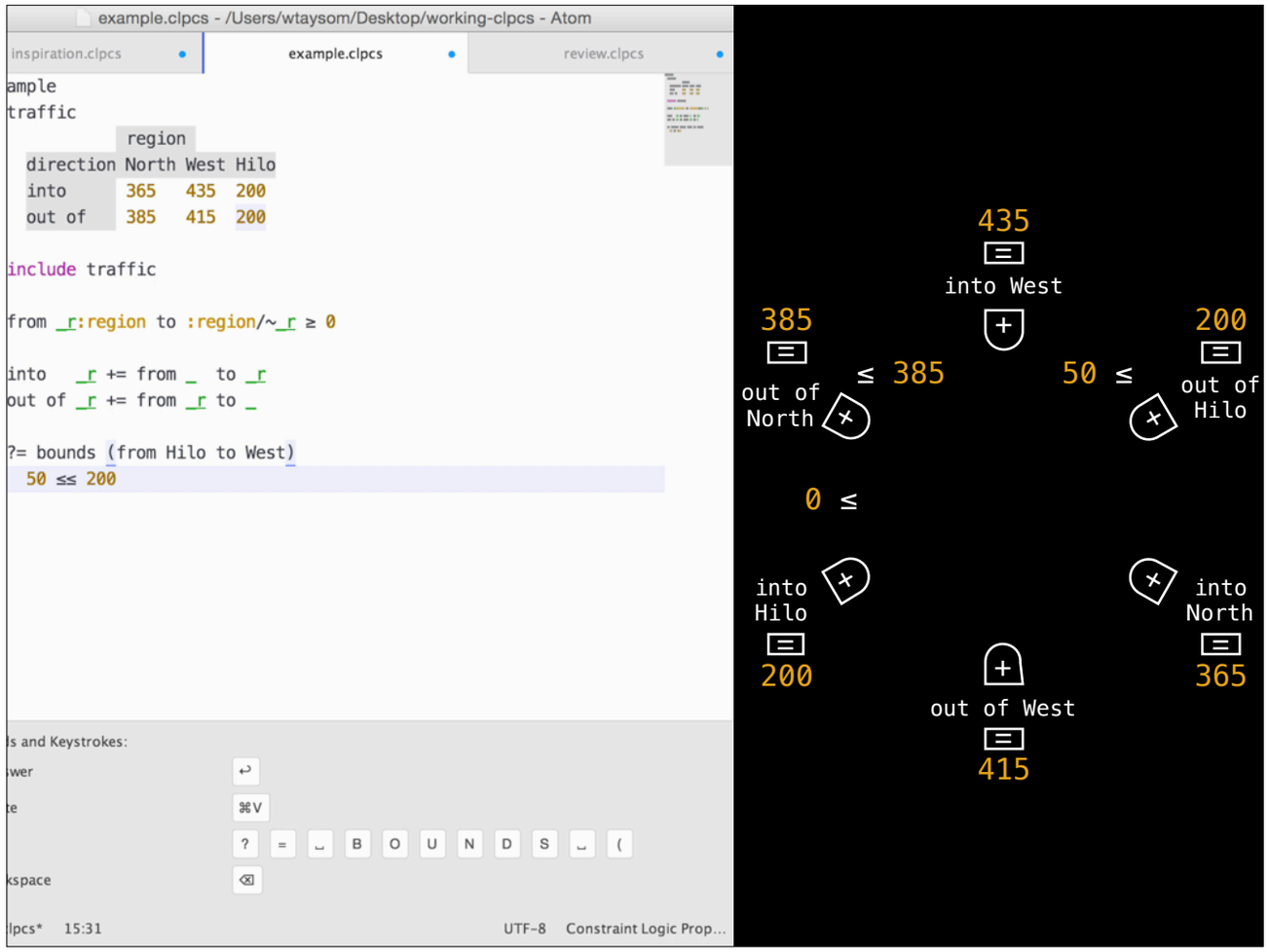
What if the traffic into Hilo were different? Say, 200.



Not only does the lower bound drop, `into West` becomes a tight constraint and `into Hilo` becomes a loose constraint. Back in the chart, switching `into Hilo` invalidates the previous inference and updates `out of Hilo`.

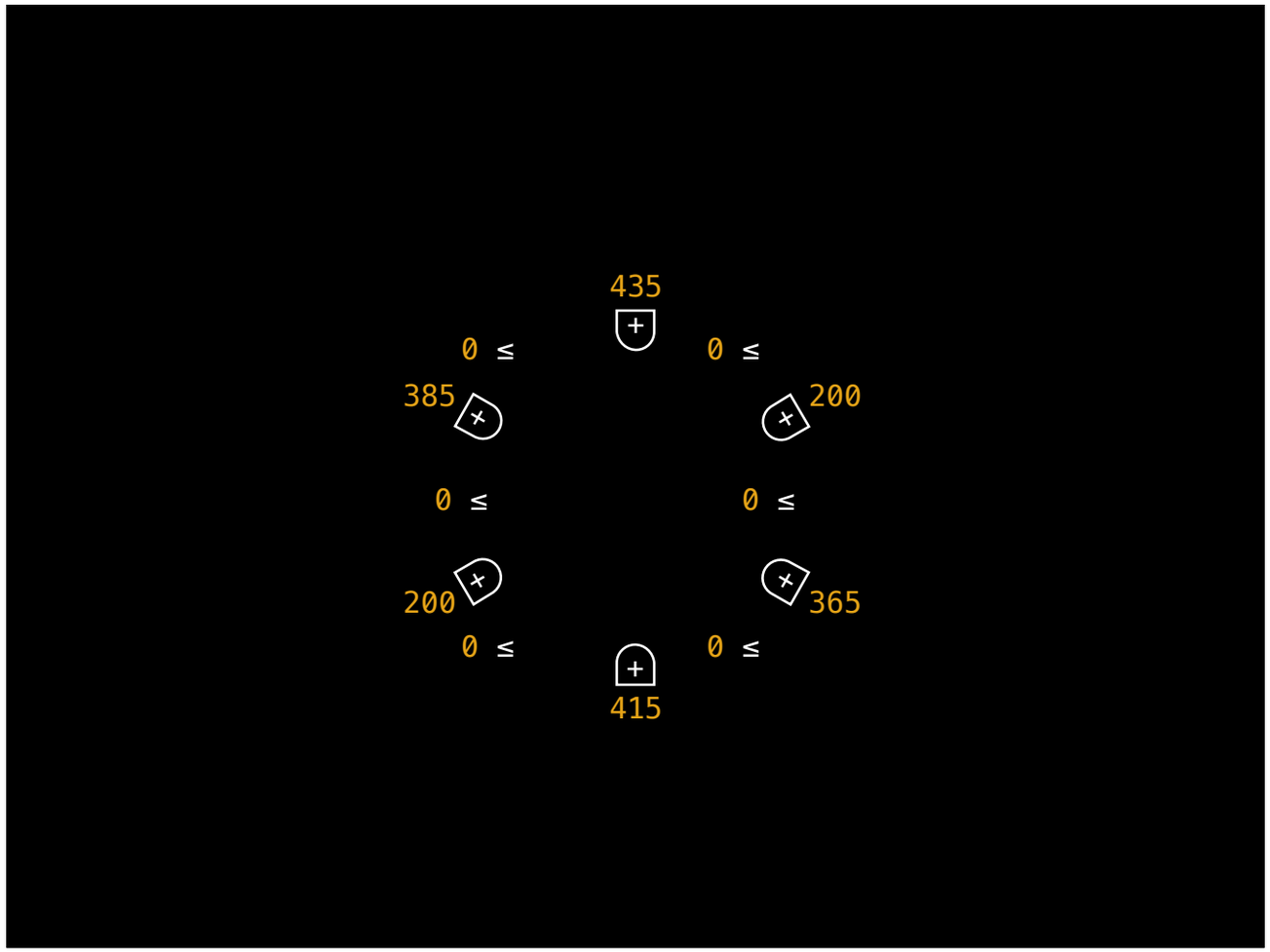


Our new bound is not as strong – having come around from the other side.

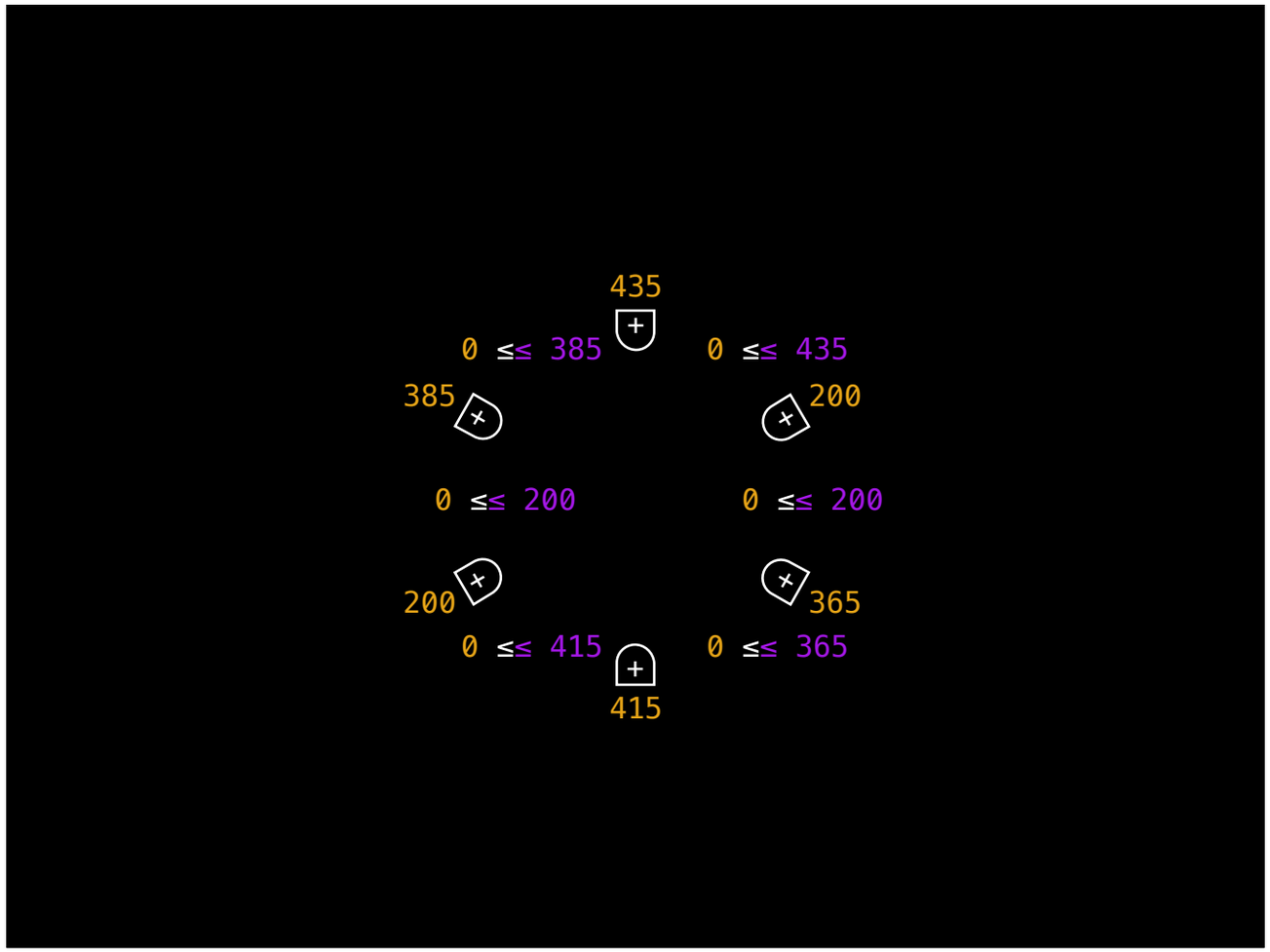


Let's ask about both the upper and lower bounds at once: the closed interval from 50 to 200. Instead of working backwards, let's see how the zero lower bounds might propagate forward.

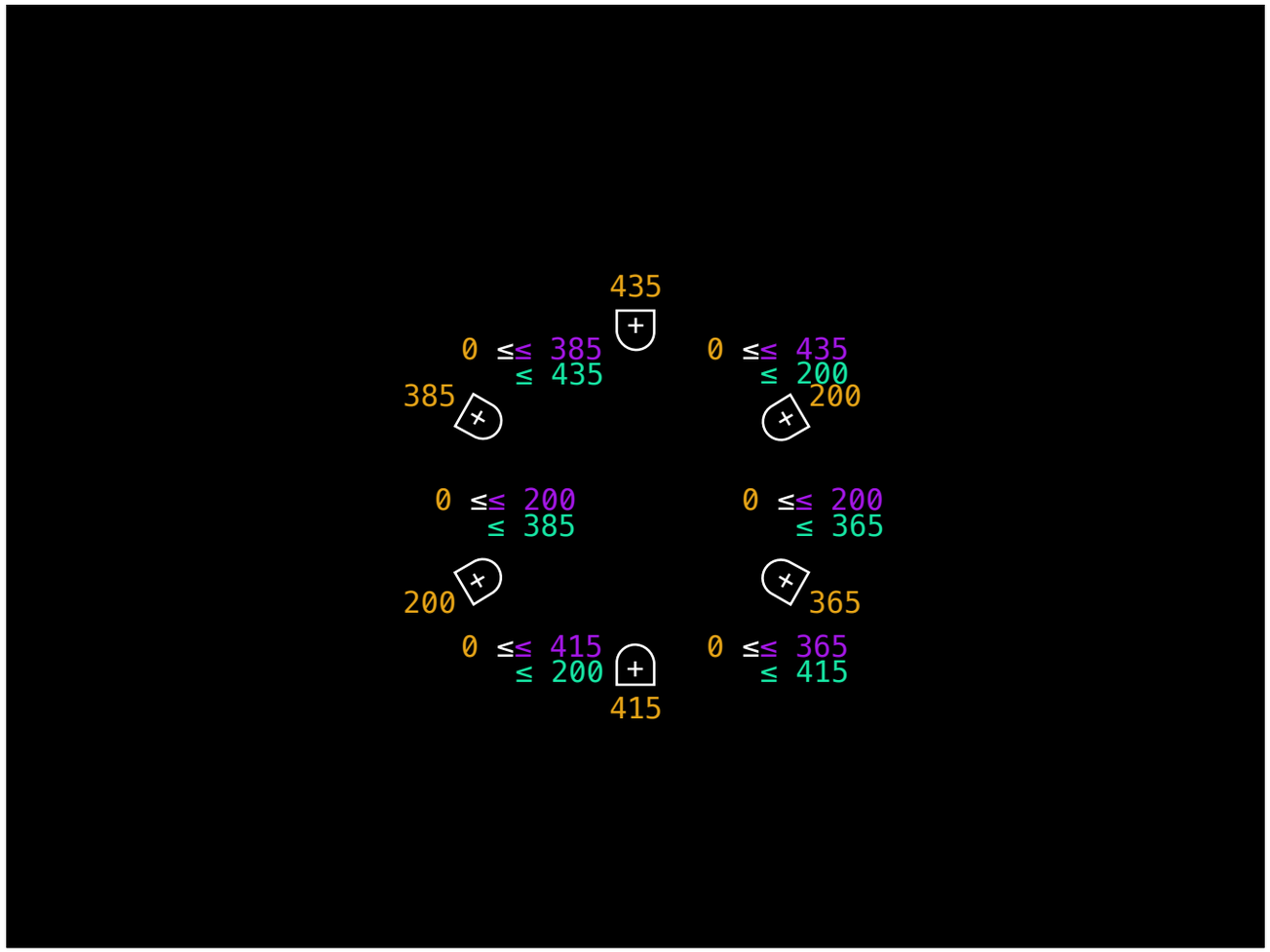
Start by stripping down to the essential relations.



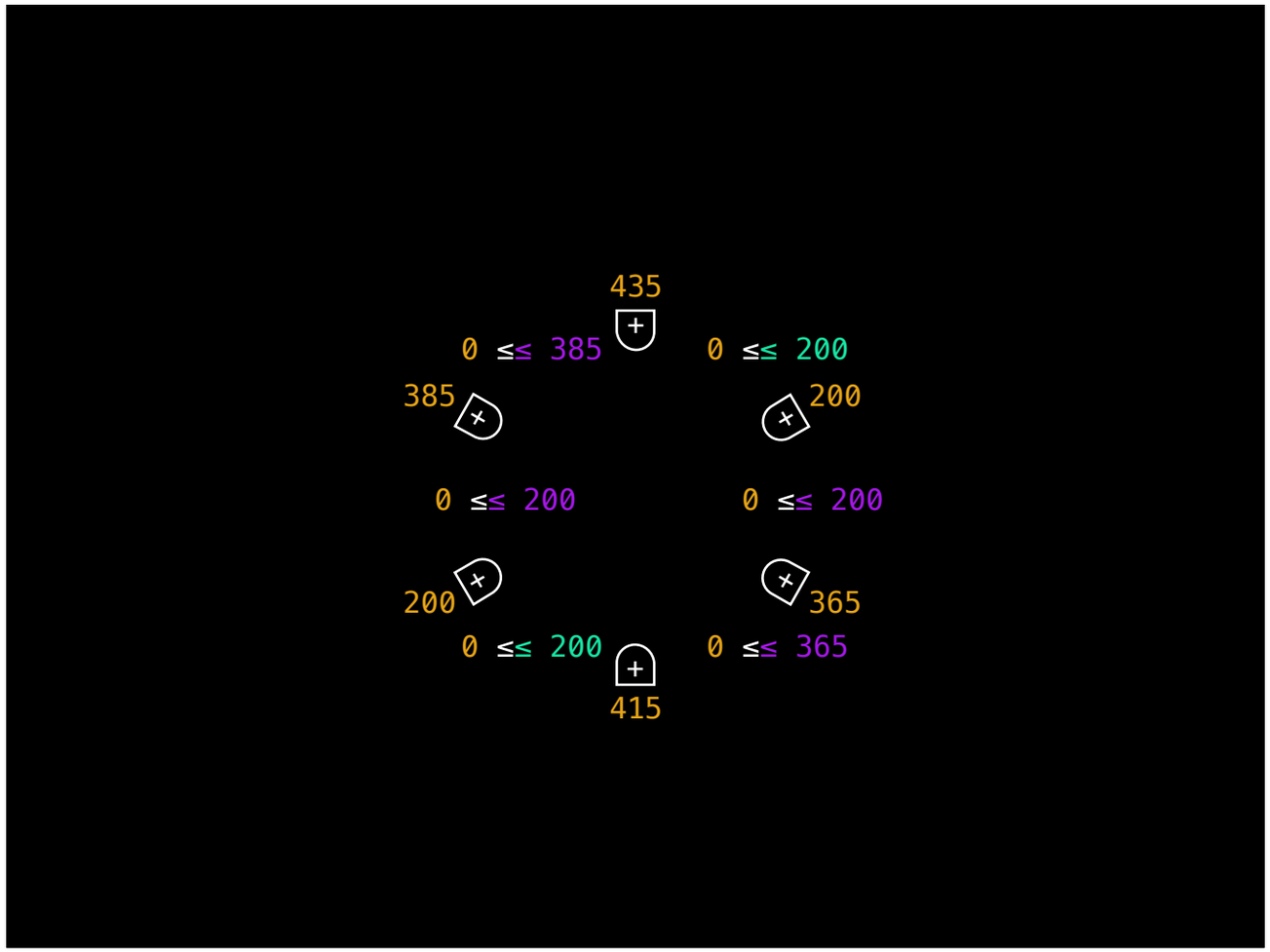
Add in the constraints.



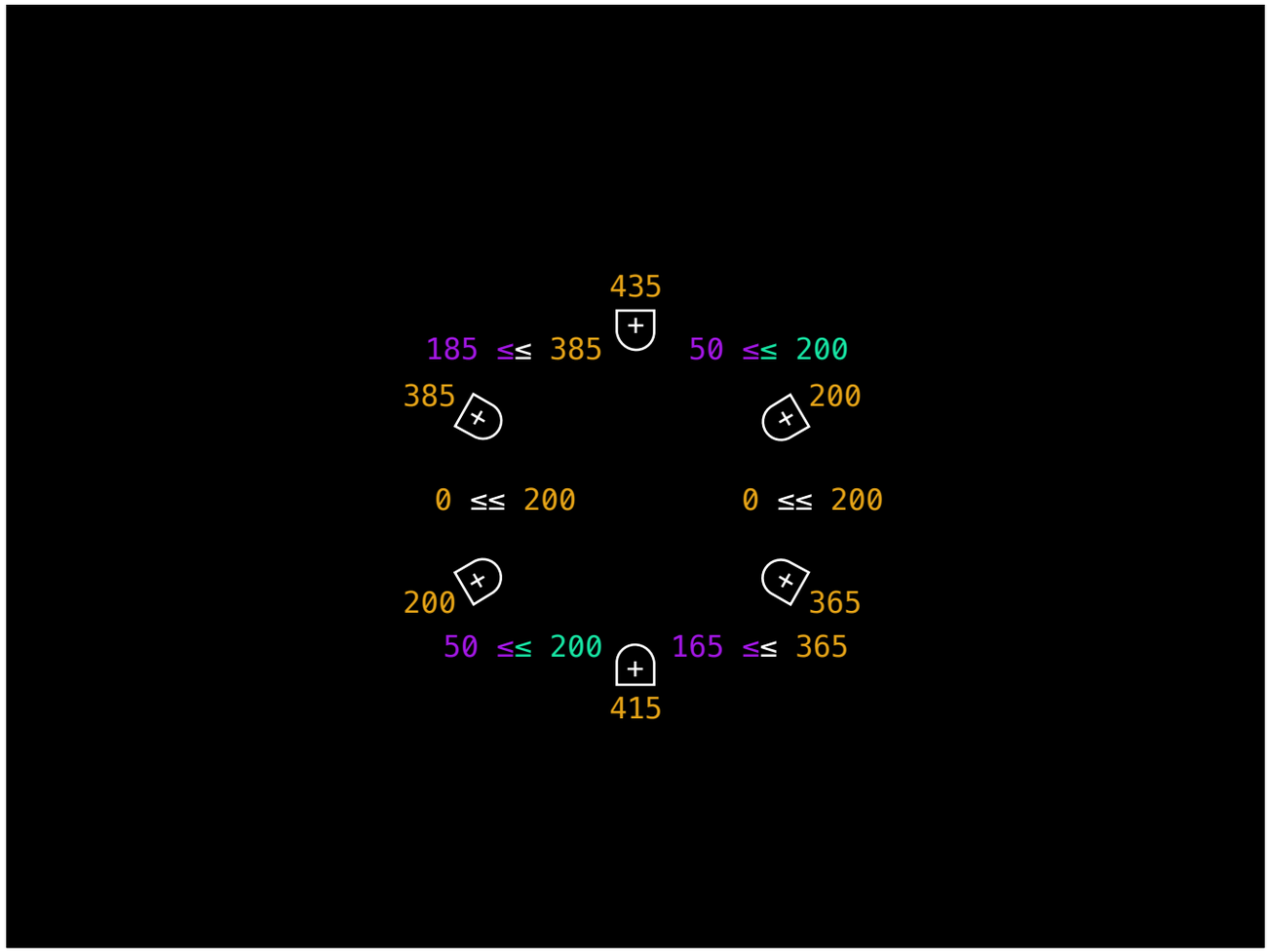
Propagate clockwise,



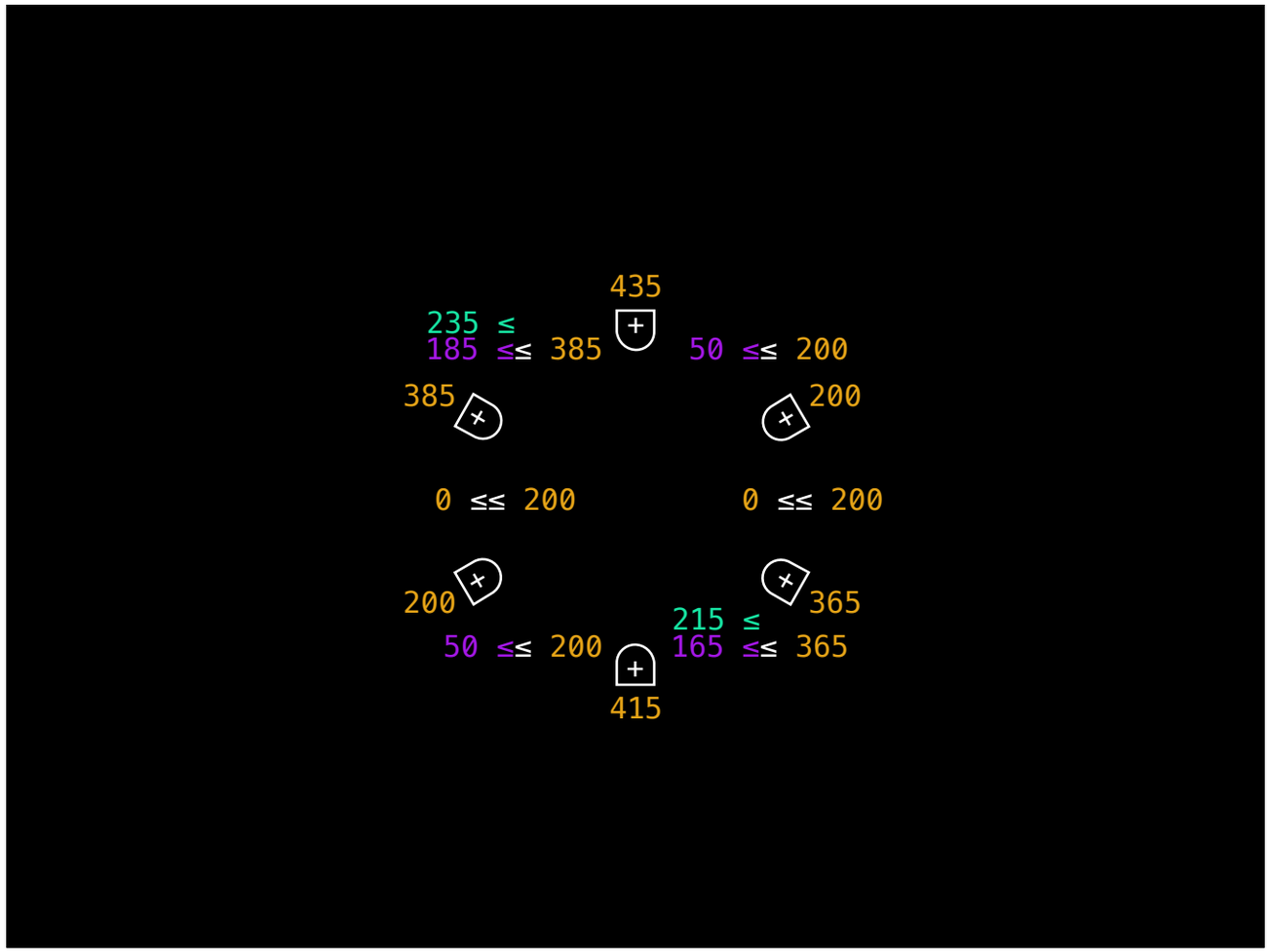
propagate counterclockwise.



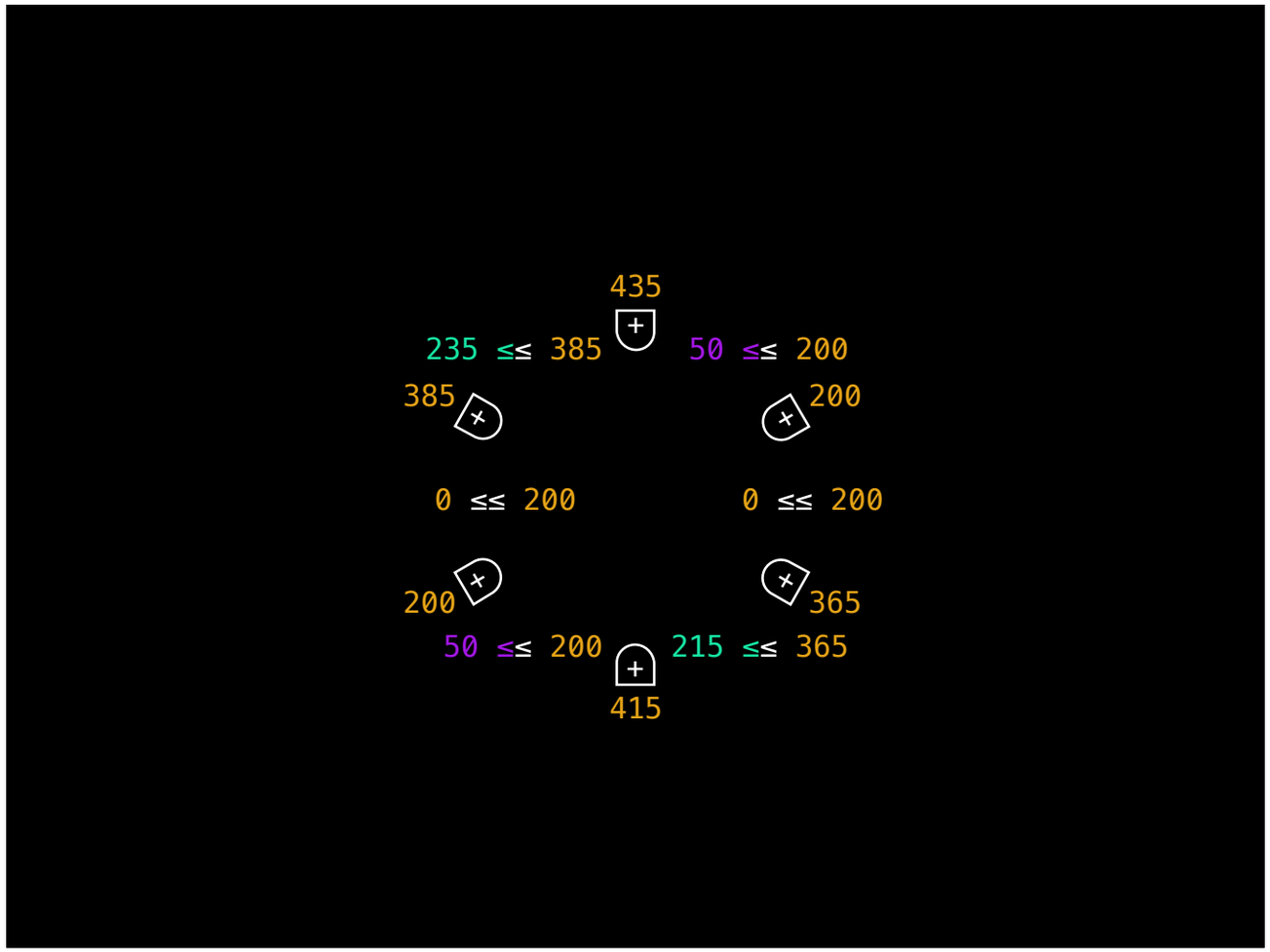
Reduce by taking the better bound in each cell.



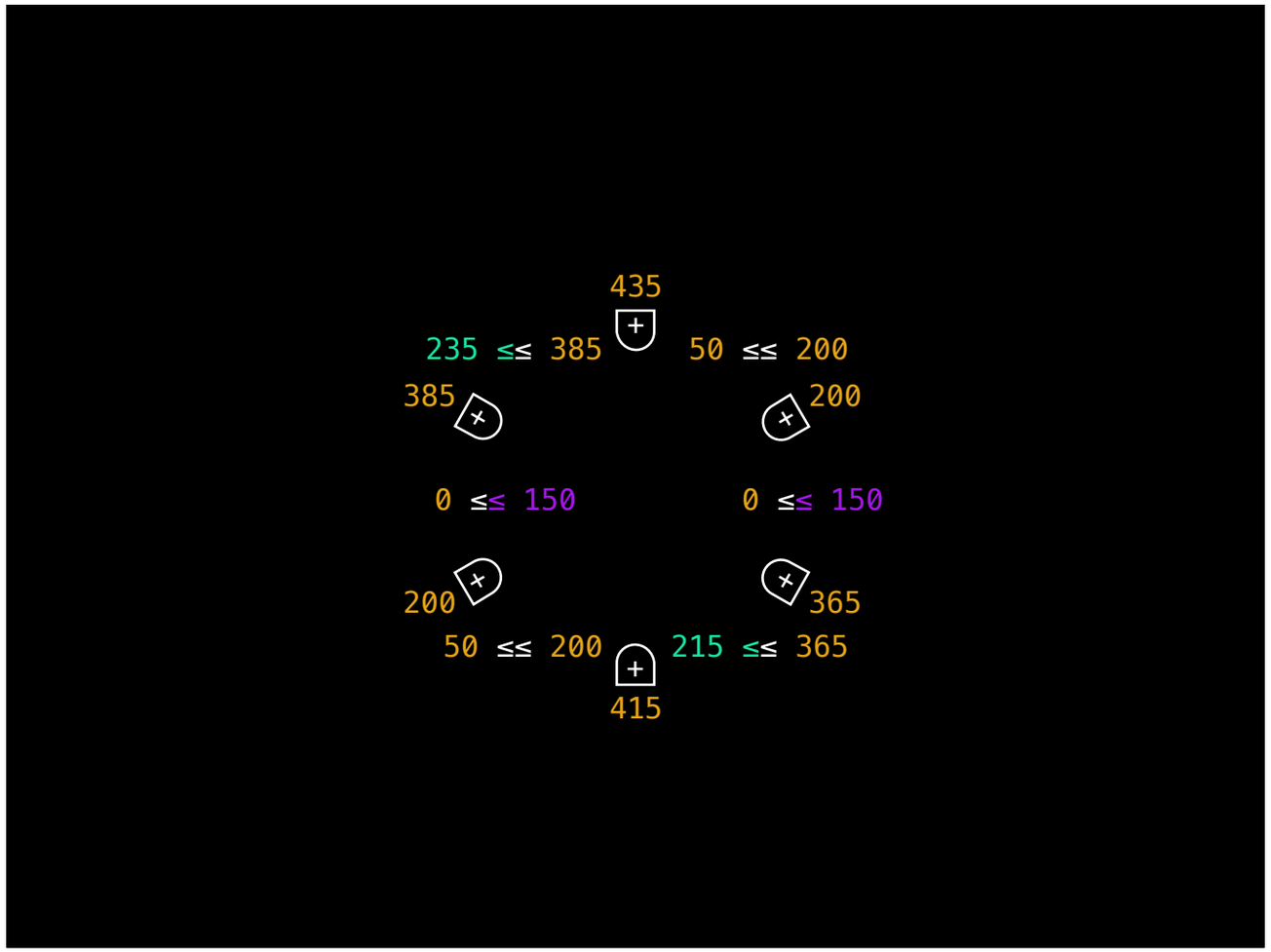
Another go. Propagate clockwise.



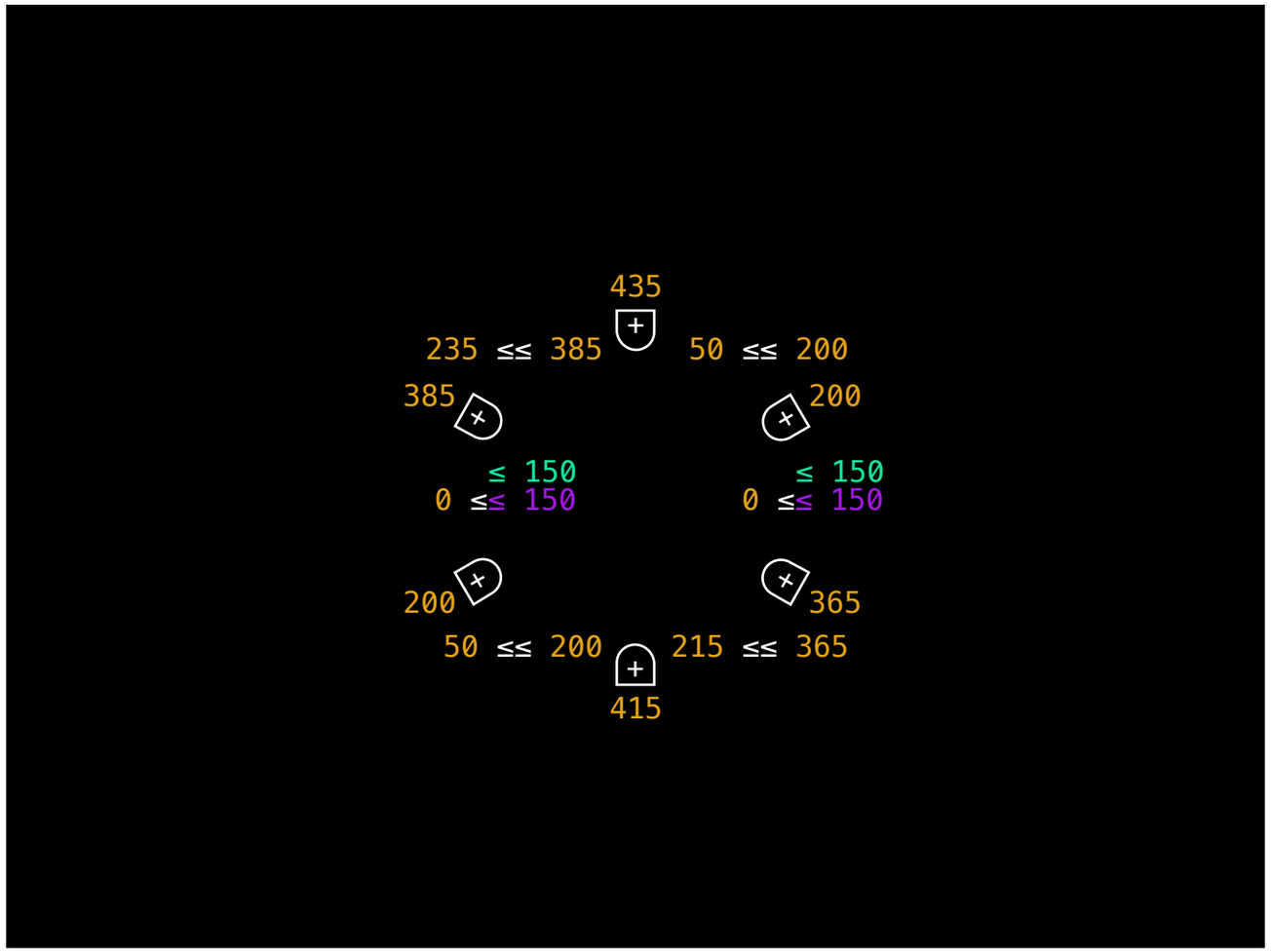
Propagate counterclockwise.



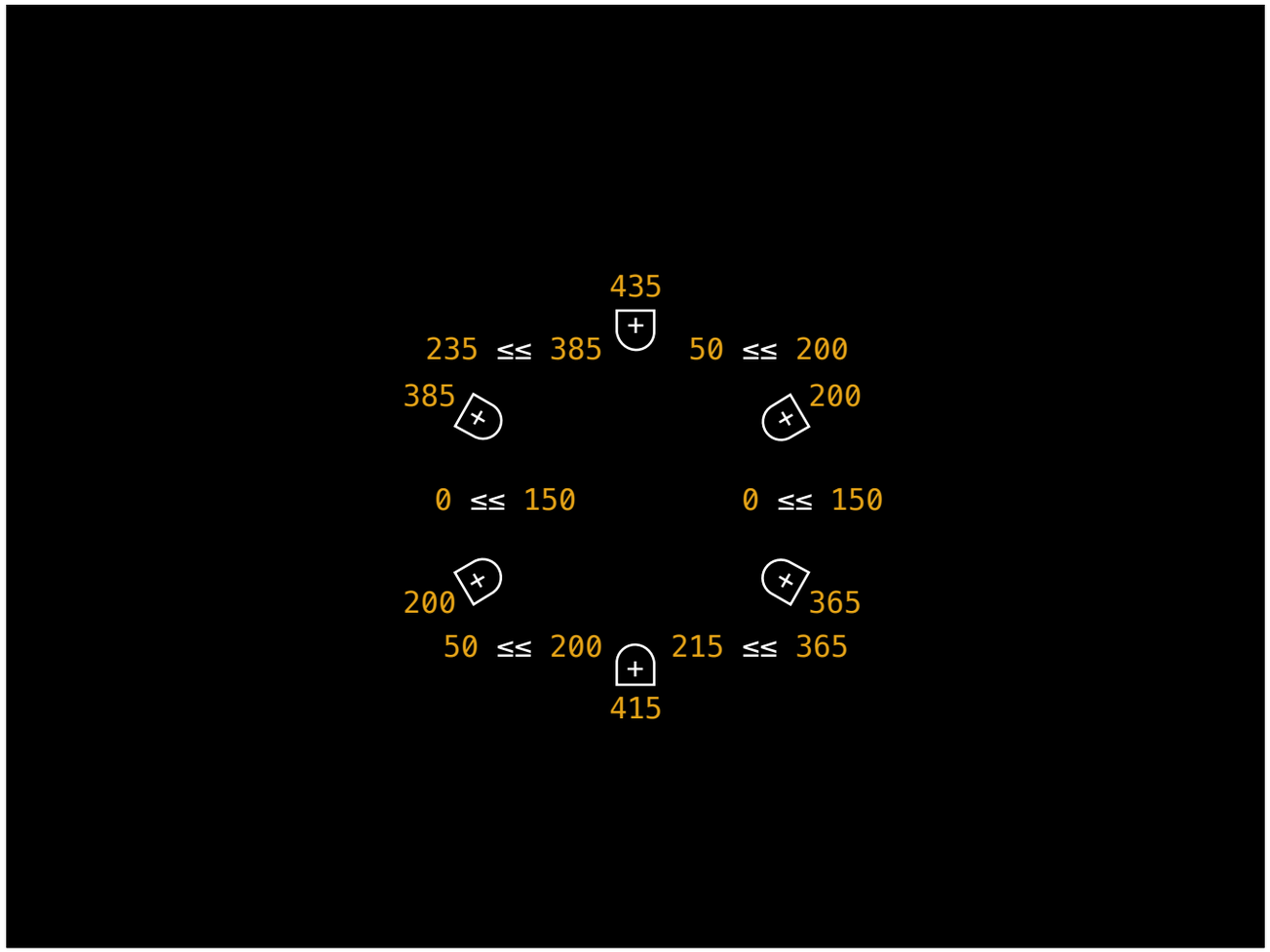
Merge again by picking the better bound.



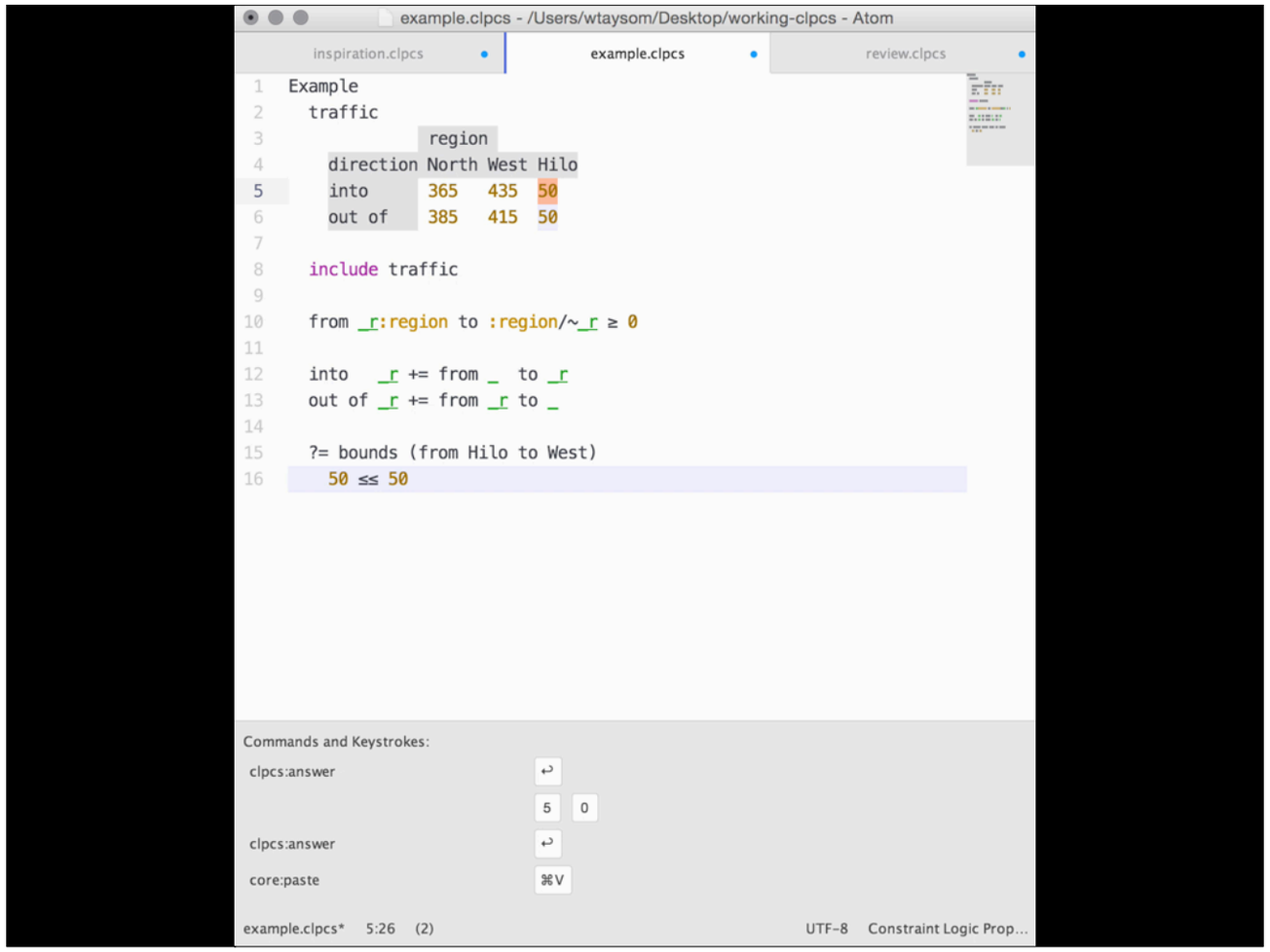
Again, again. Clockwise.



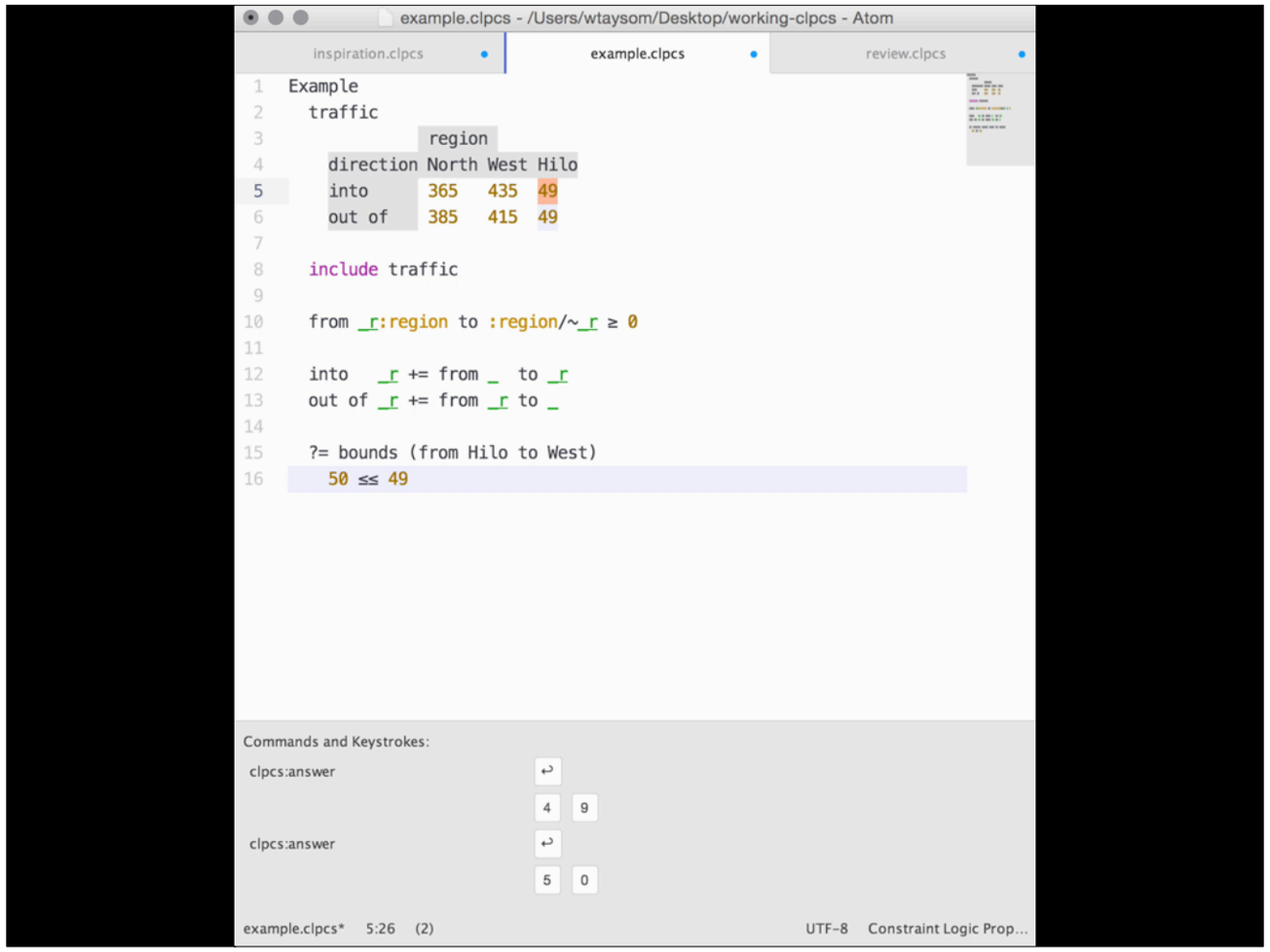
Counterclockwise.



At last the two waves cancel. And that is forward propagation. Let's bring back the editor.



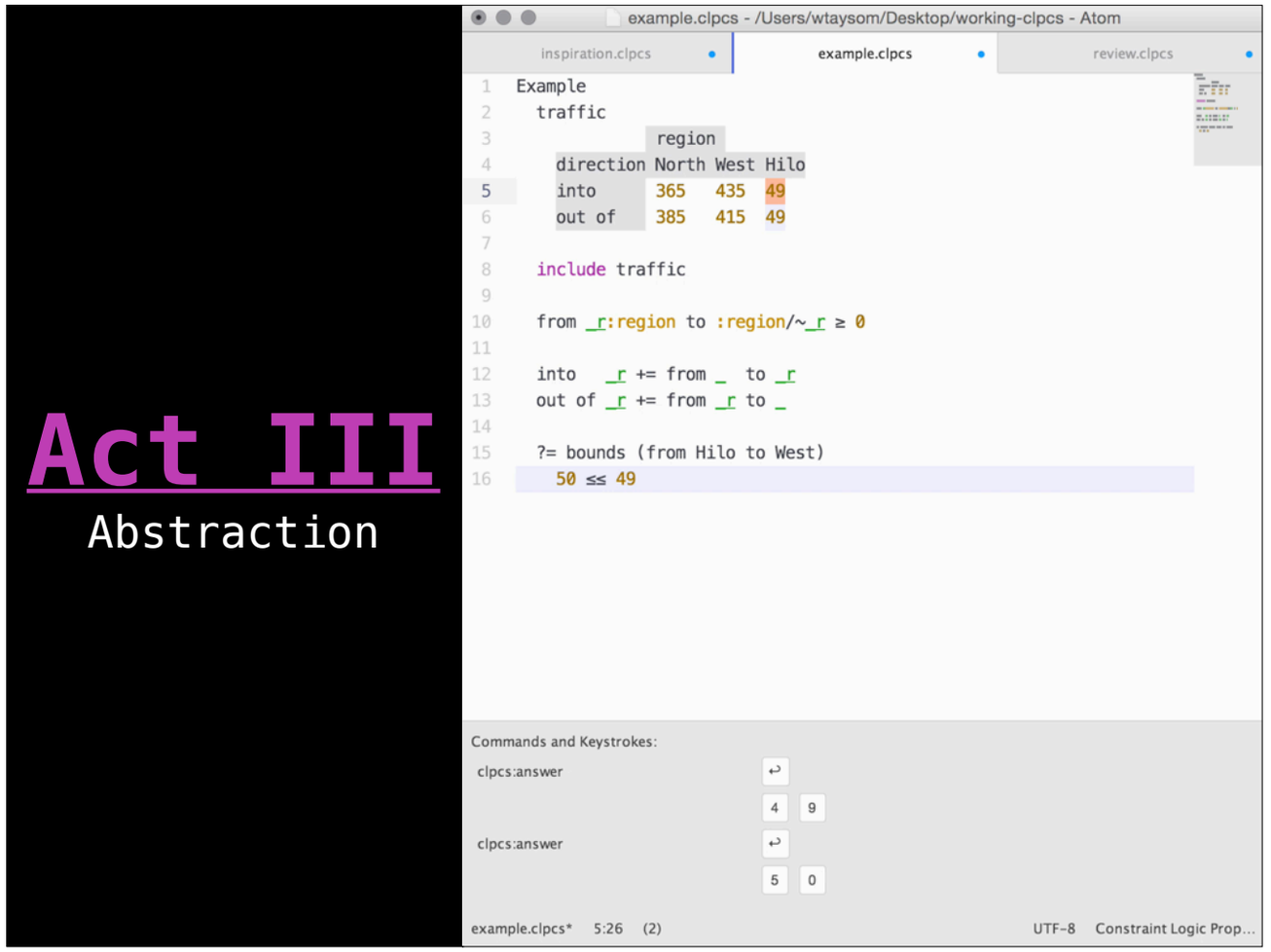
We get equality at 50.



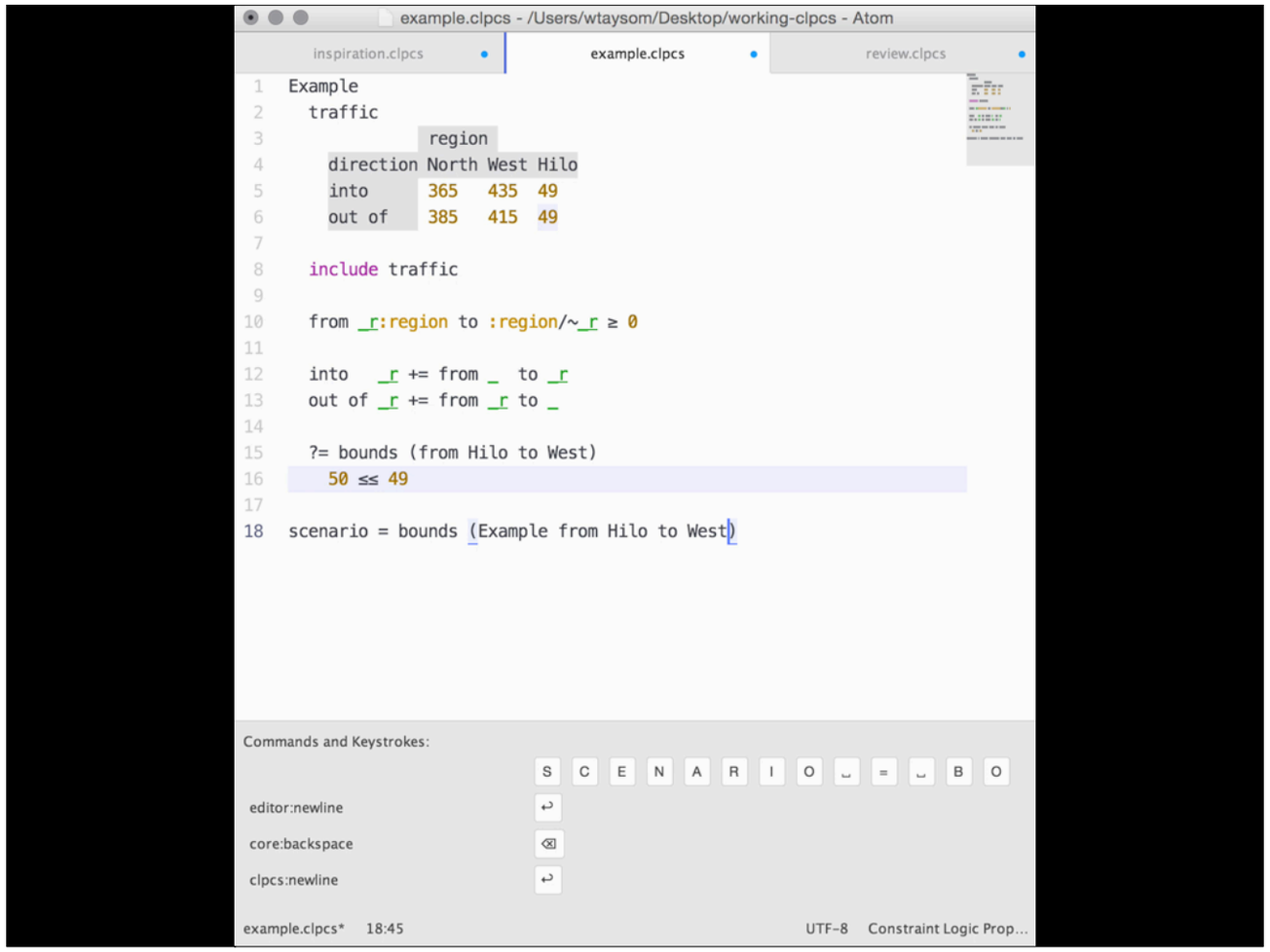
Below 50, we have conflict.

Act III

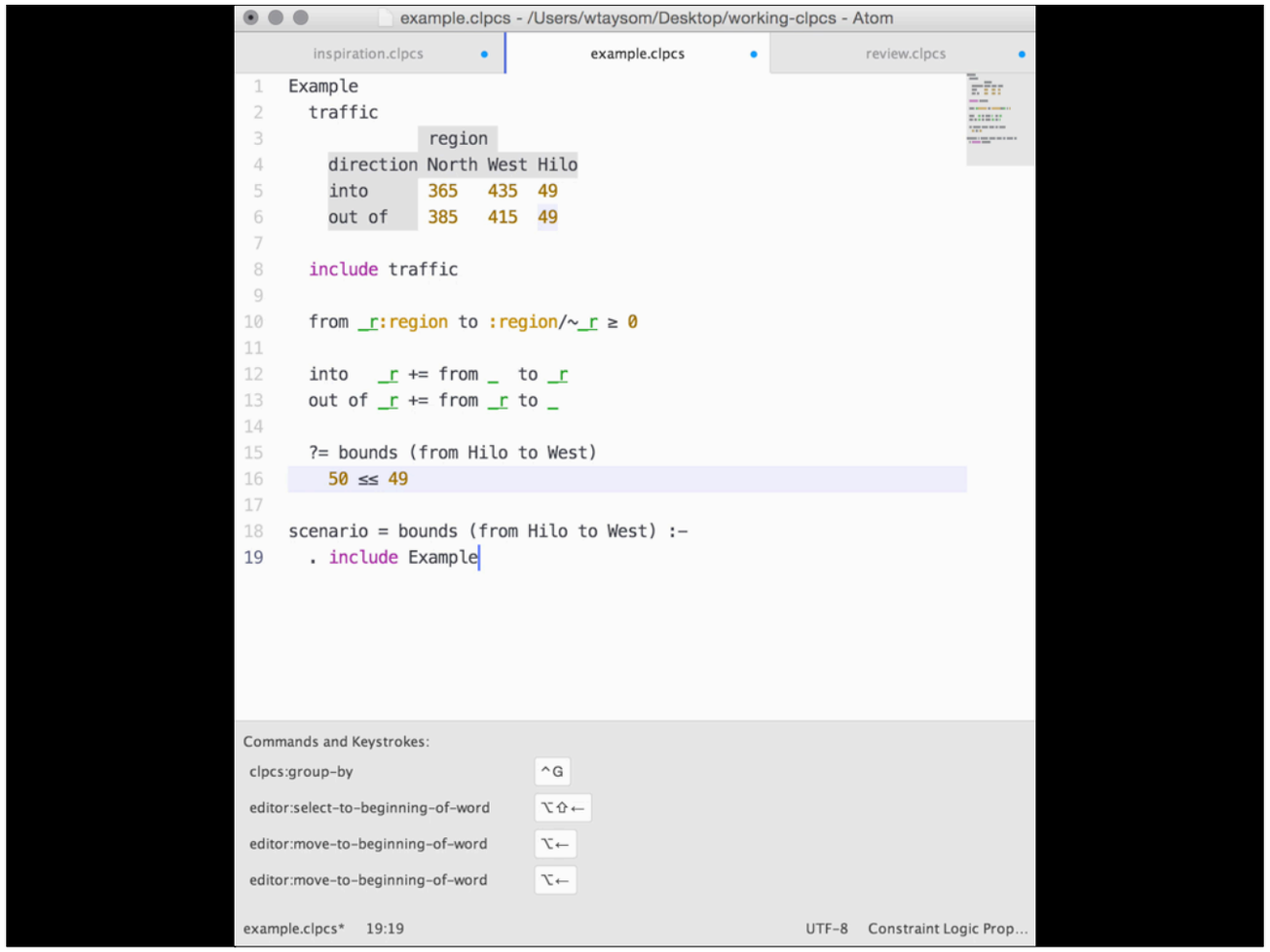
Abstraction



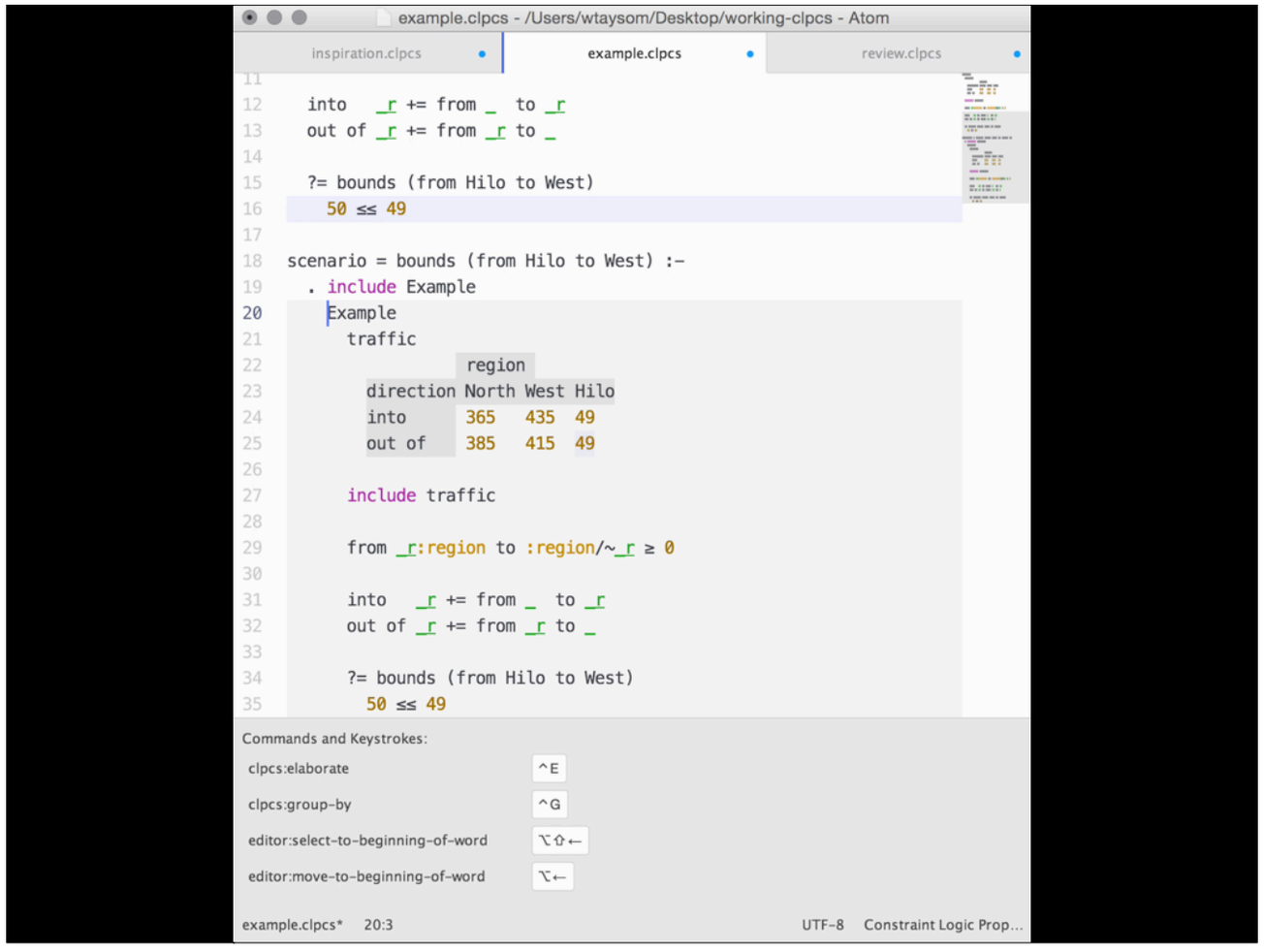
To see how these what-if scenarios relate, let's climb up the ladder of abstraction.



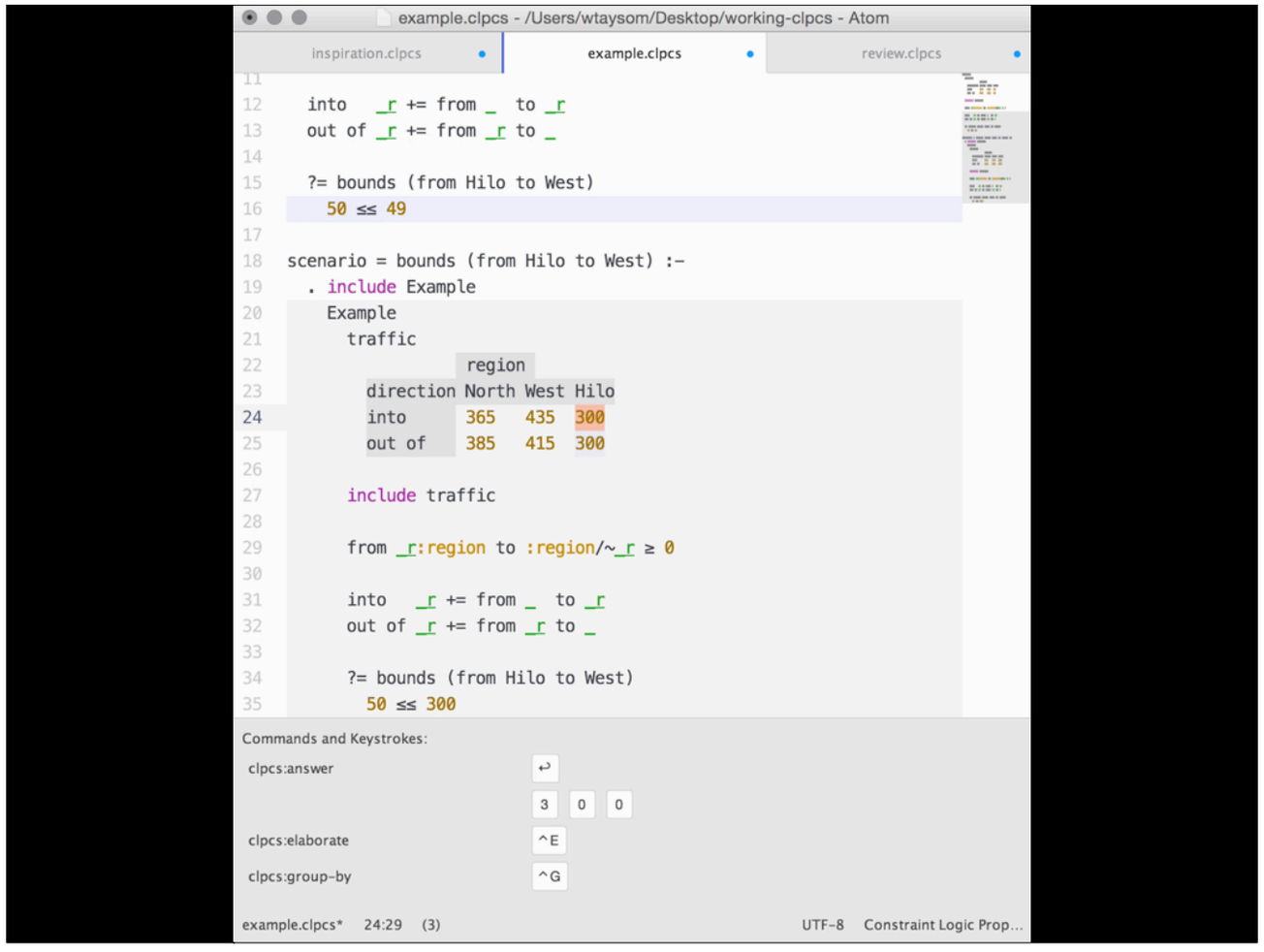
For our scenario, we're interested in the bounds on `from Hilo to West`.



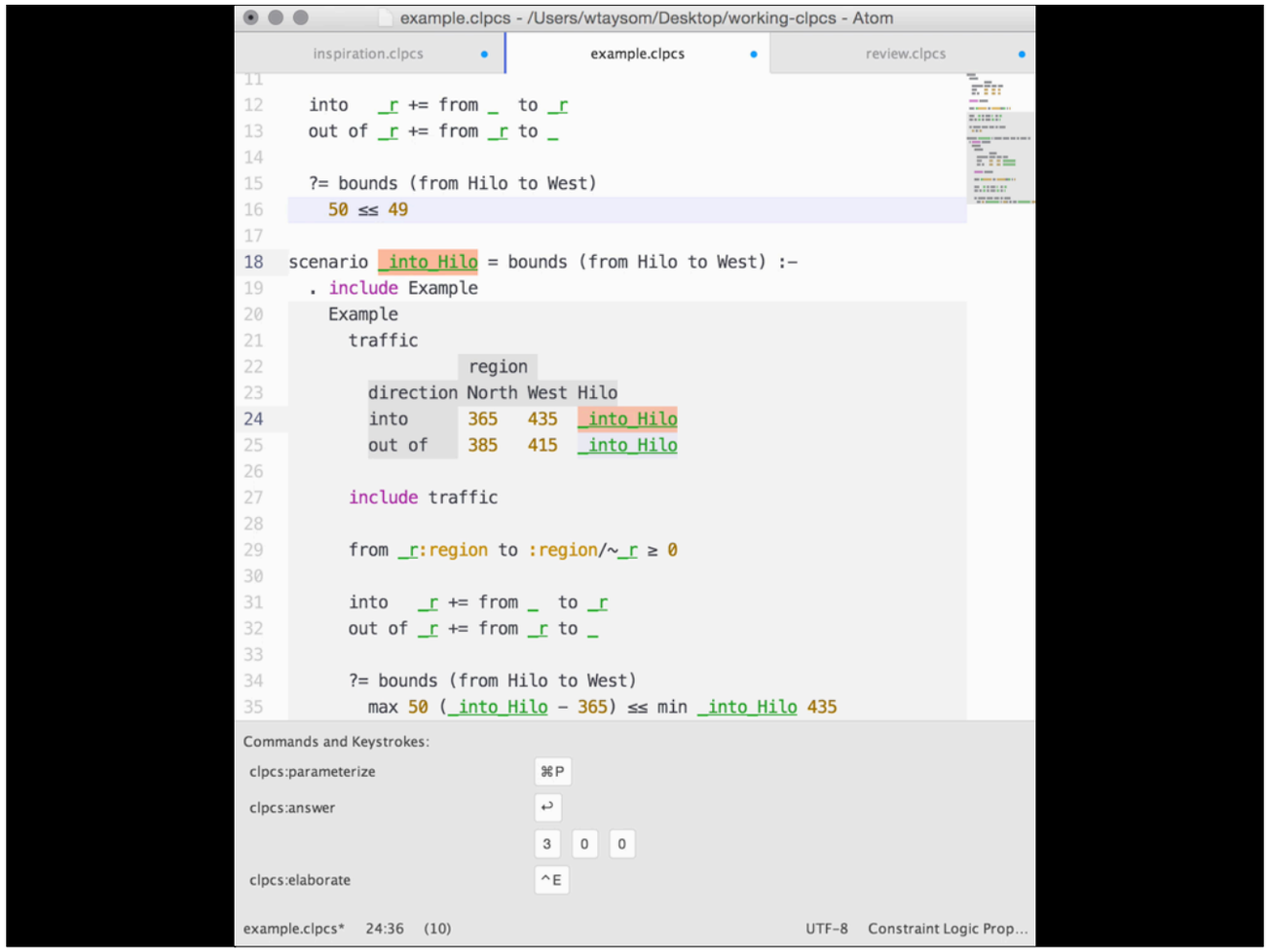
Include the `Example` module into the body of `scenario` for unqualified reference. The dot here allows a rule to have local facts. So within this rule all Example facts hold.



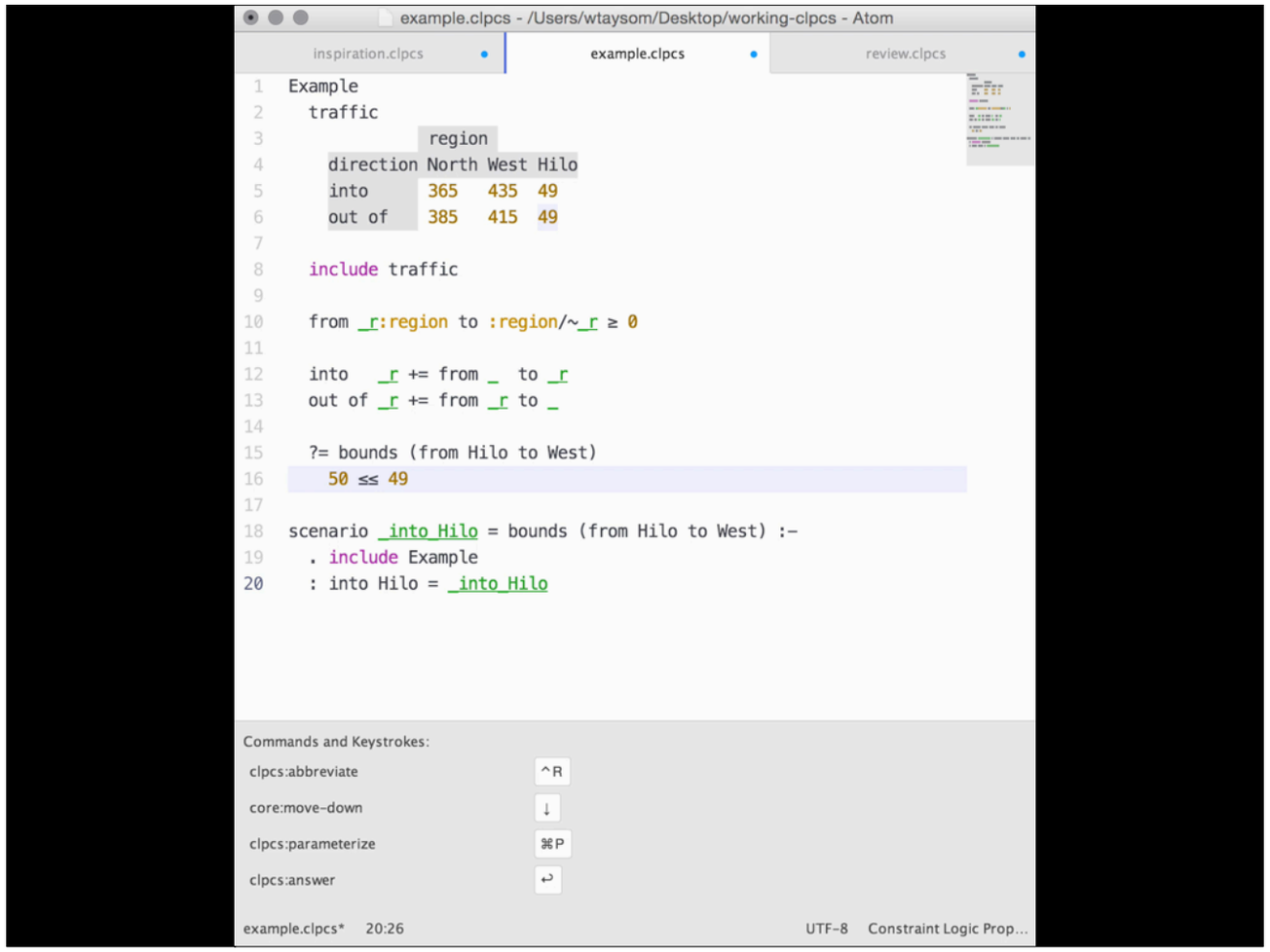
Take a look inside. Here we see the definition of the Example table. I use a gray background to suggest that the original is kept elsewhere.



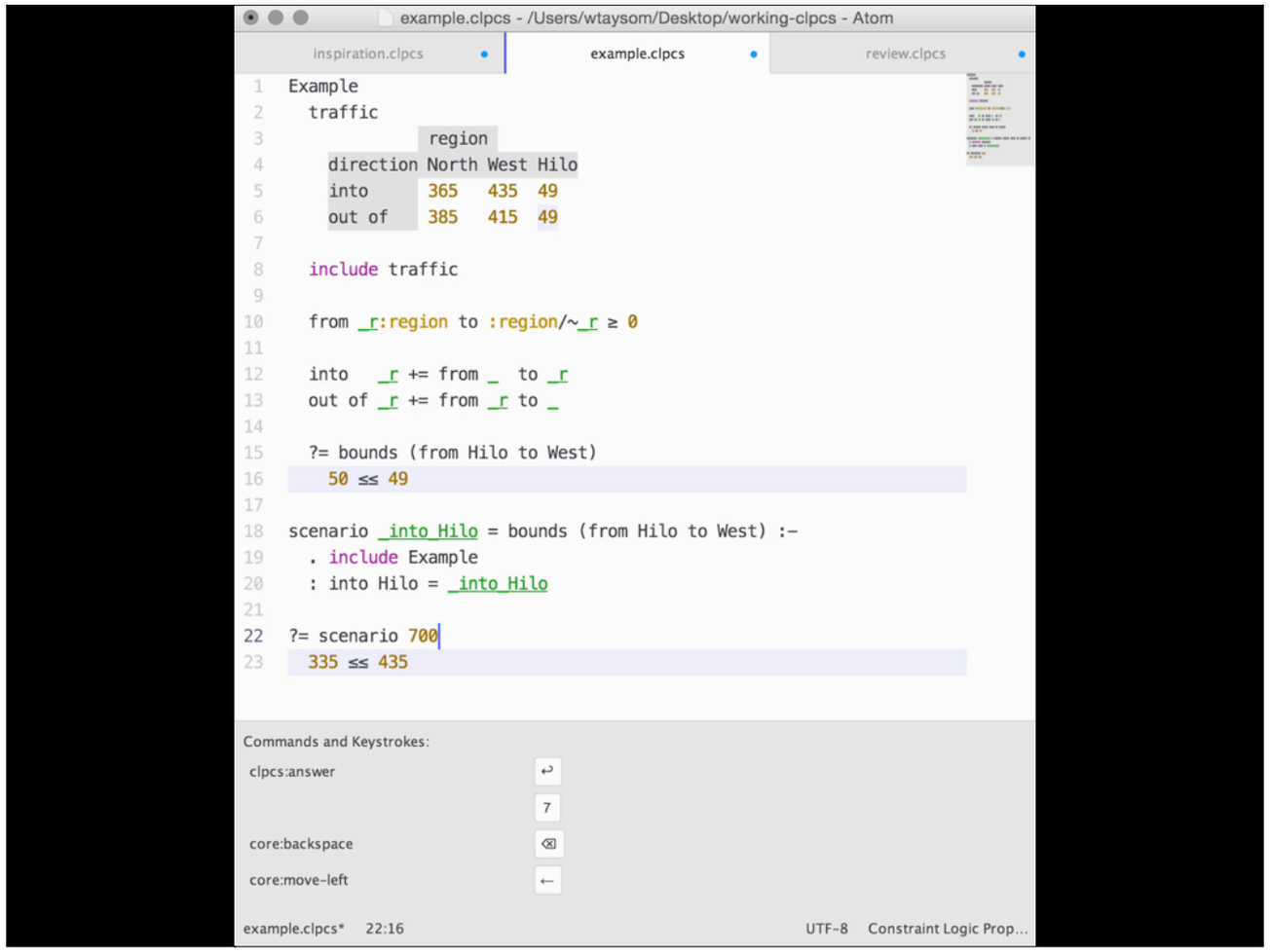
Beyond just viewing the included table, we can customize it. When we override the included version, the original remains unchanged.



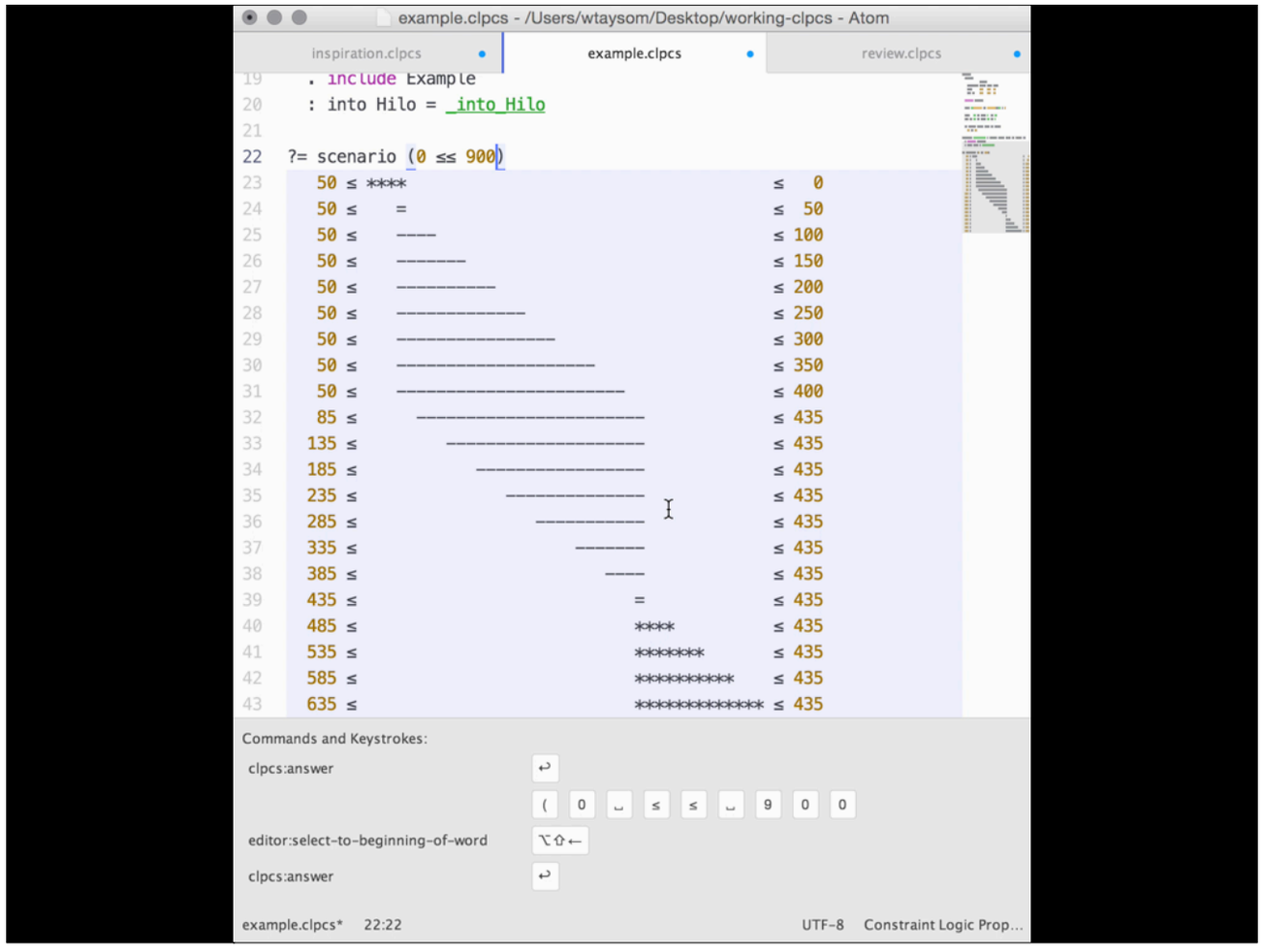
More importantly, we can parameterize over a term.



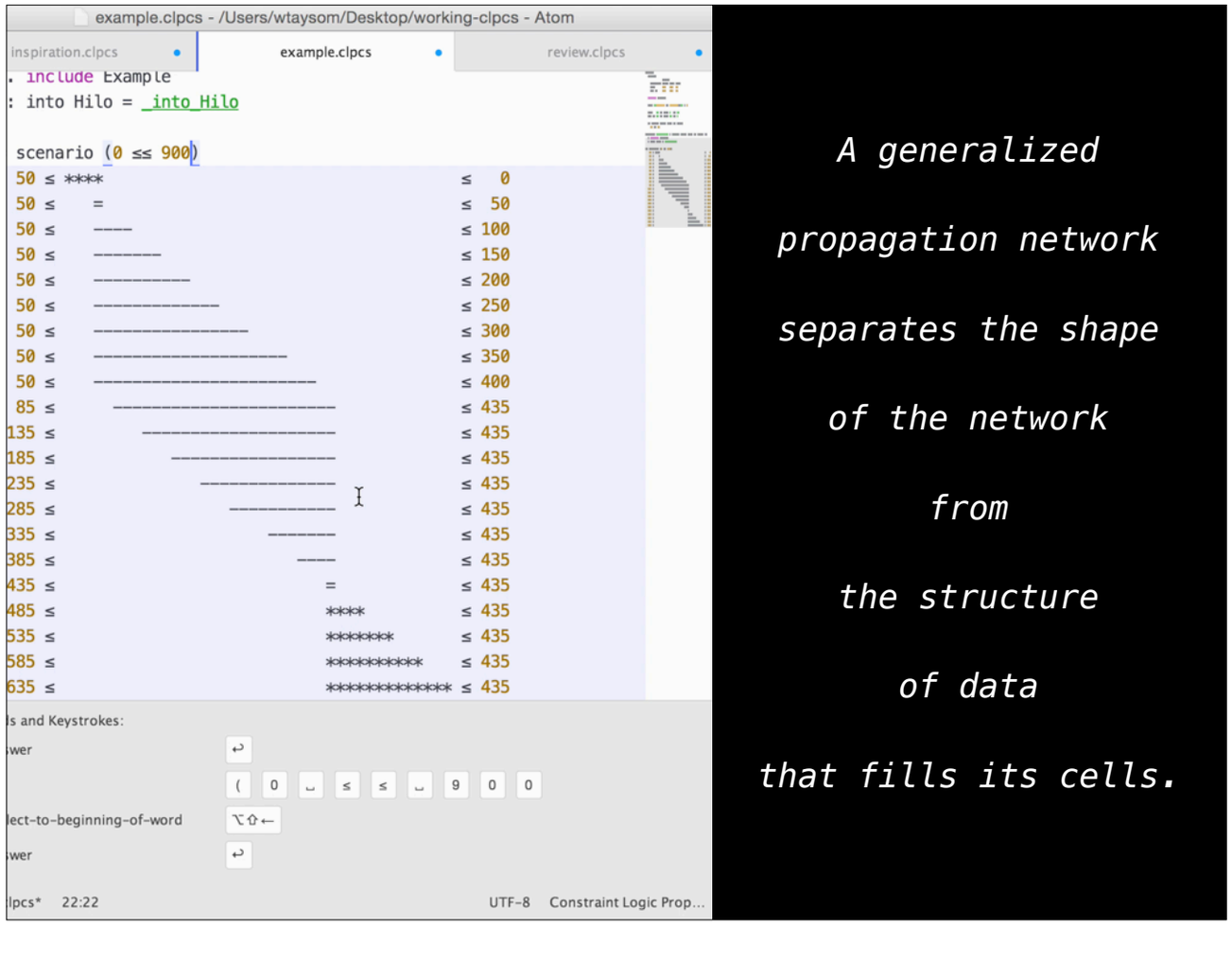
Close up the module. See that we're simply overriding a fact. Between an overriding diff and the fully woven version with all customizing patches integrated, some things are easy to see in the derivative, some are clearer in the integral. In the future, we freely switch between the two.



Try out some scenarios.



And finally, let's ask about a whole interval at once. The answer is an interval of intervals, charted crudely: dashes for standard intervals, stars for conflicted intervals, and equals sign for singletons. Although we defined the scenario with scalars in mind, we can apply it to intervals by mapping the scenario across a selection of values from the interval. Similarly, we can automatically lift other functors: sets, lists, probability distributions, time varying signals.



A generalized

propagation network

separates the shape

of the network

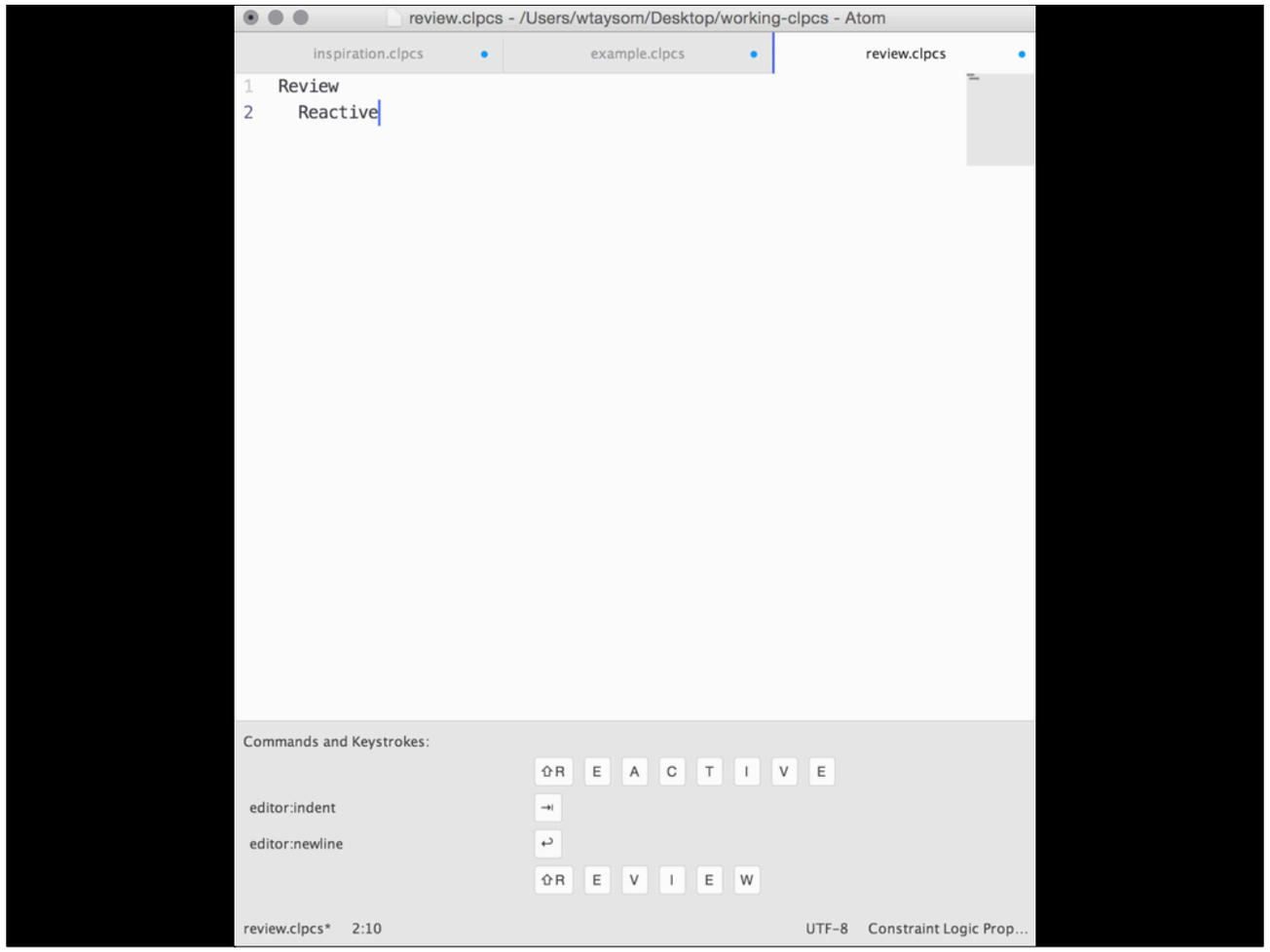
from

the structure

of data

that fills its cells.

A generalized propagation network separates the shape of the network from the structure of data that fills its cells. It offers a new kind of abstraction with more potential than I'm prepared to explore today. Instead let's review what we have explored.

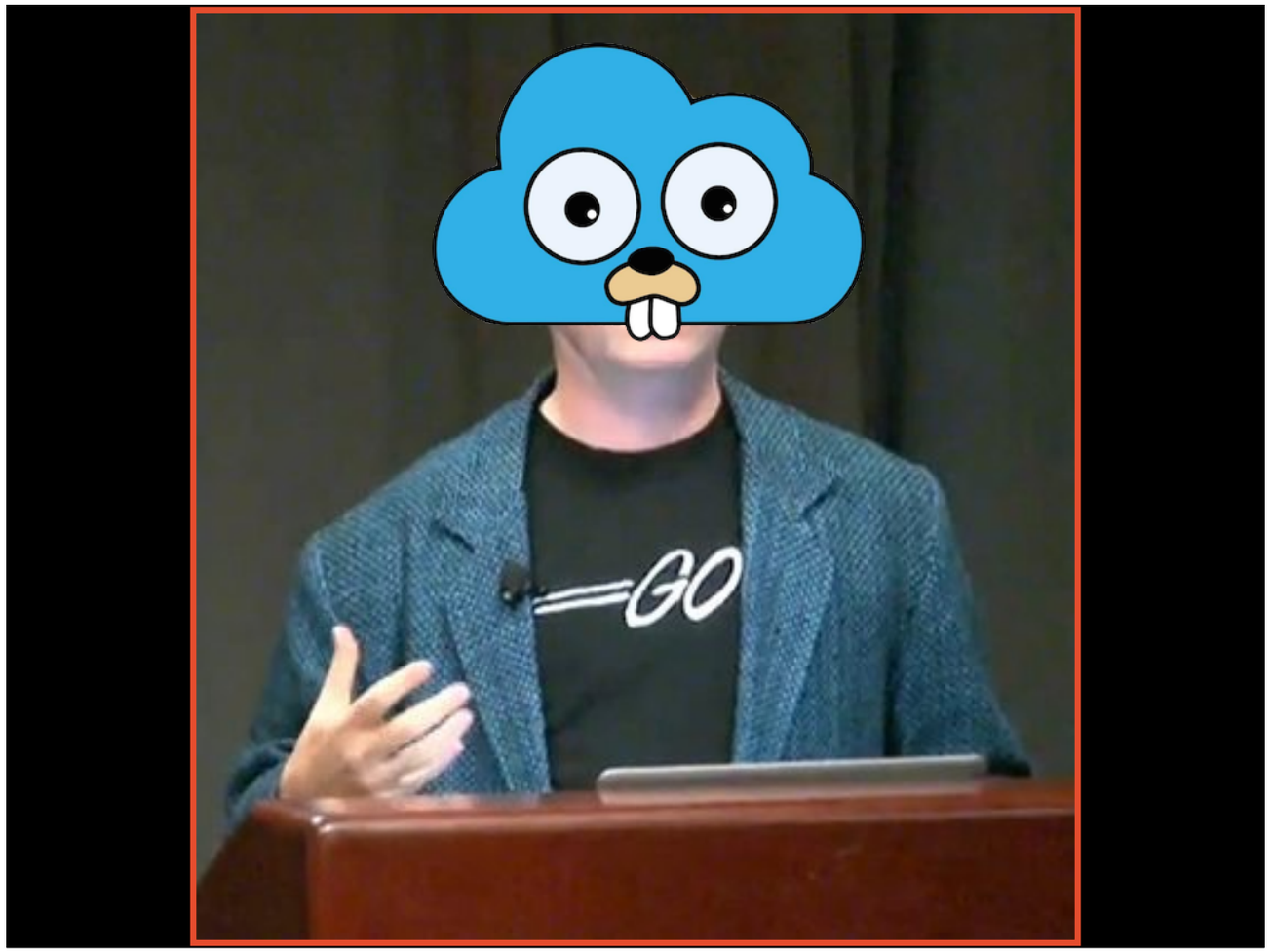


Being reactive is par for the course at this workshop. For every wibble, have related things wobble. Except we don't actually want every wibble to wobble.

While in the middle of typing a question or making a change, a system shouldn't interrupt. Every messenger app seems to experiment with sharing your keystrokes until they realize that people put their thoughts together only wanting to send after they hit return. So in the example, I marked uncommitted changes with a light yellow background.

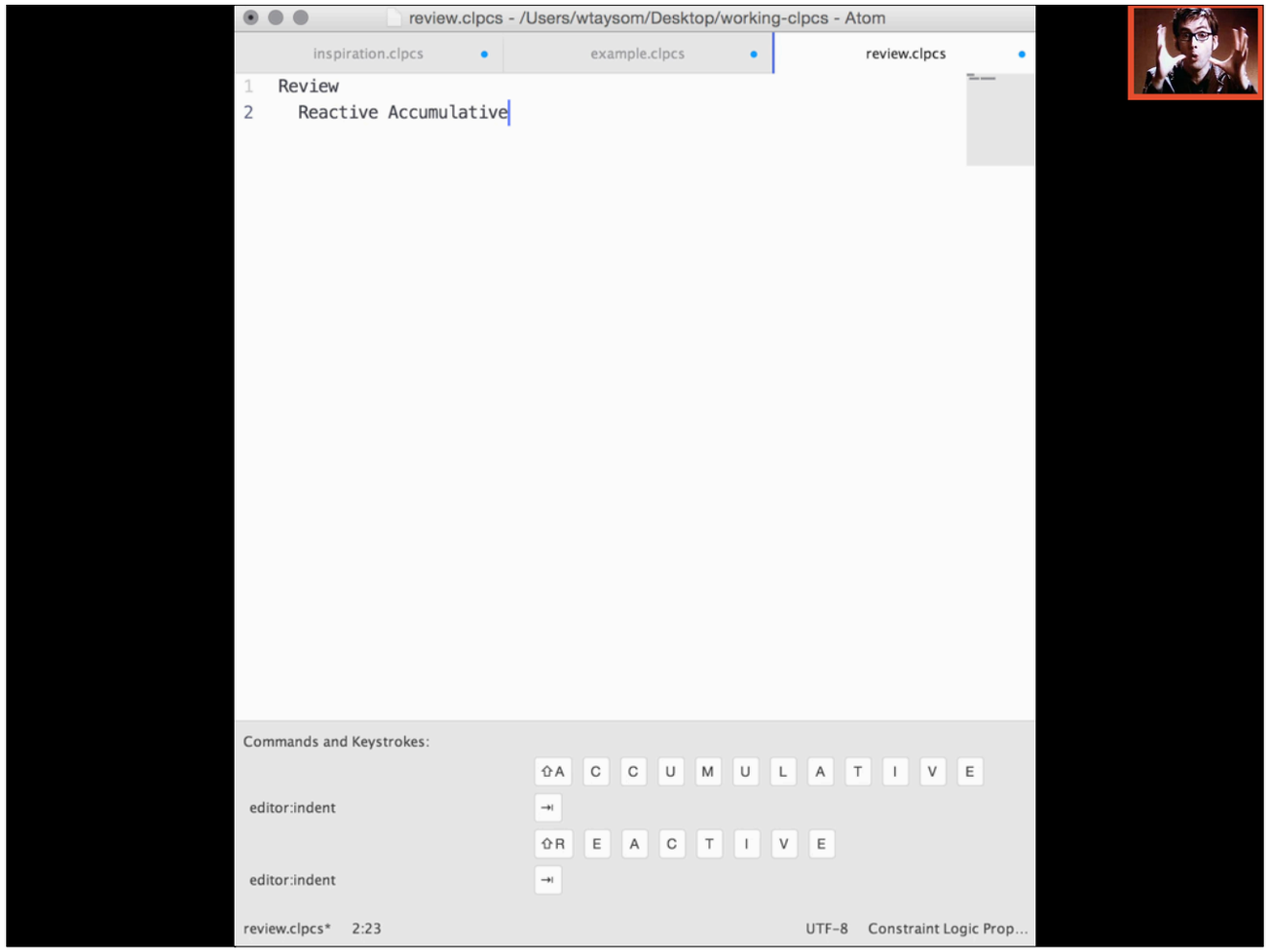
Some questions don't have an immediate answer. It takes time to think things through. Even with a datacenter of answers, it takes time to look them up. Build with these constraints in mind. First and always: keep the interface responsive. Process in the background. Report progress. Abort if asked. Be able to pause a process, inspect it, rewind.

Sounds like a list of operating system services. I'm not sure why process APIs remain so true to their 1970s roots.



Perhaps with the web, all effort drifted away up into the cloud.

A topic for another time.



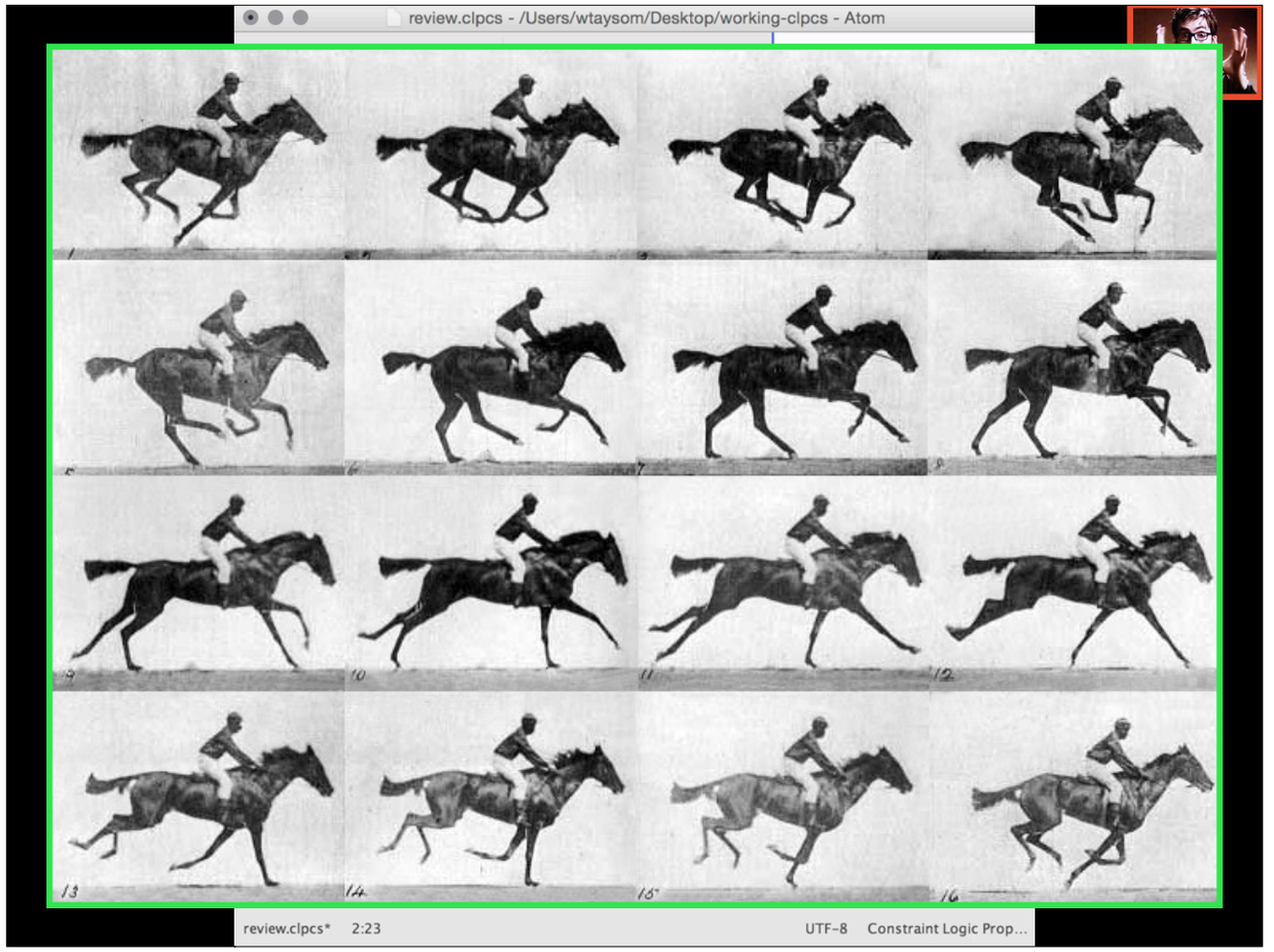
Time. A funny dimension. A good place to keep bugs. Functional programming goes to show you can get by without time some of the time. When you do need time, be explicit about it:

"IO using monads doesn't look like purely functional programming, and it shouldn't because it isn’t."

A basic spreadsheet is pure: every formula a pure function. But the people who make them are not. We build things over time, their structure at any moment being just a snapshot.

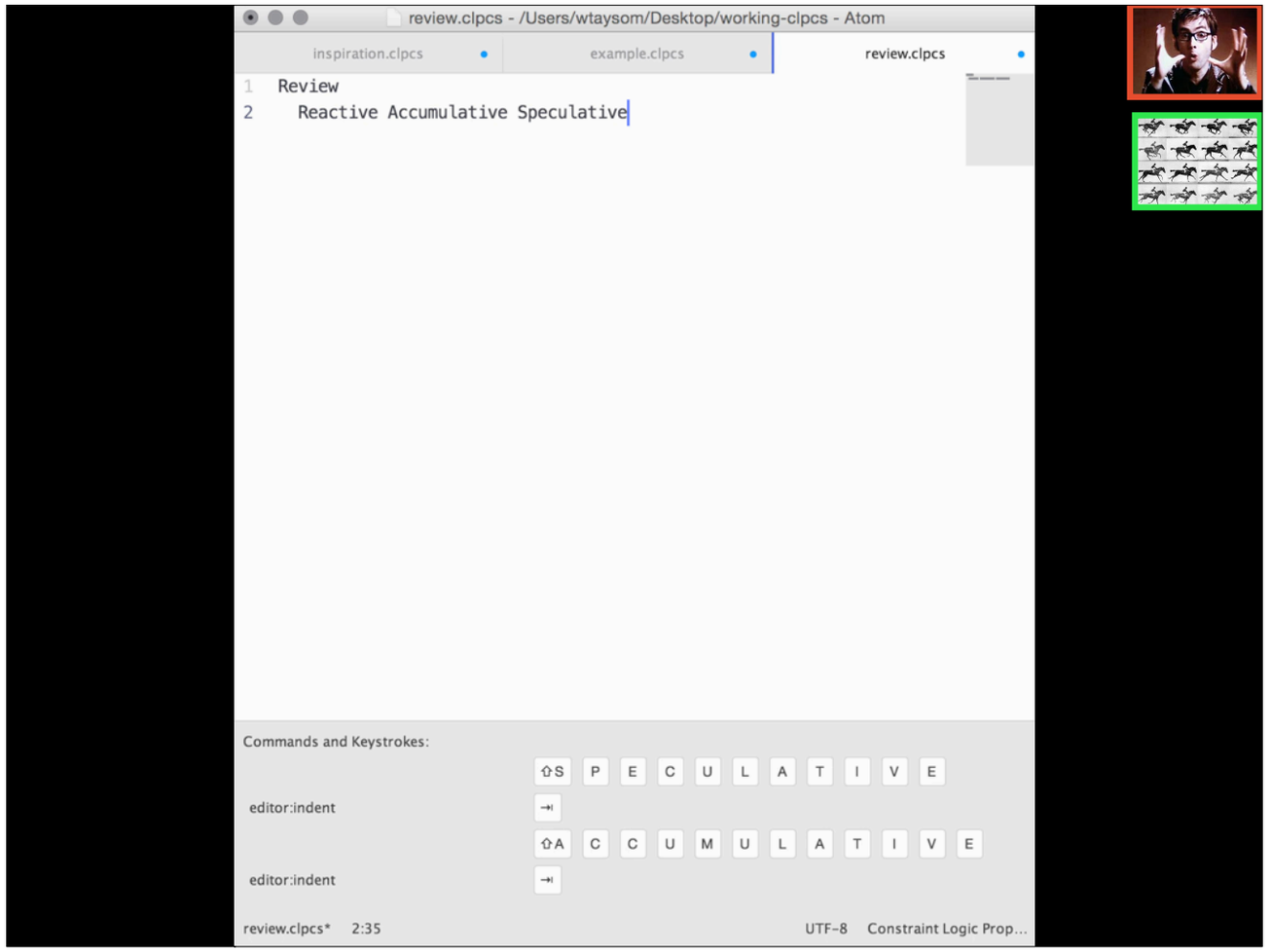
In order to work with time, we have to step out of time. I don't mean metaphysically, just that the order of events in a story doesn't need to match the order in which they were written. You can do it backwards: the resolved becomes unresolved, clues skitter away to their hiding places, foreshadows become regular back-shadows.

When reading, when interpreting, begin at the beginning, and go on till you come to the end: then stop. With structured programming, the key idea was to have control flow match code flow: first instruction to first line, last instruction to last line with some skipped and repeated lines in the middle. (Admittedly, that never quite worked out. The graphical user interface may be partly to blame.)



Still in our example, facts accumulated from top to bottom. A question asked early-on ignored spoilers that came later. Time unfolds strung along the vertical.

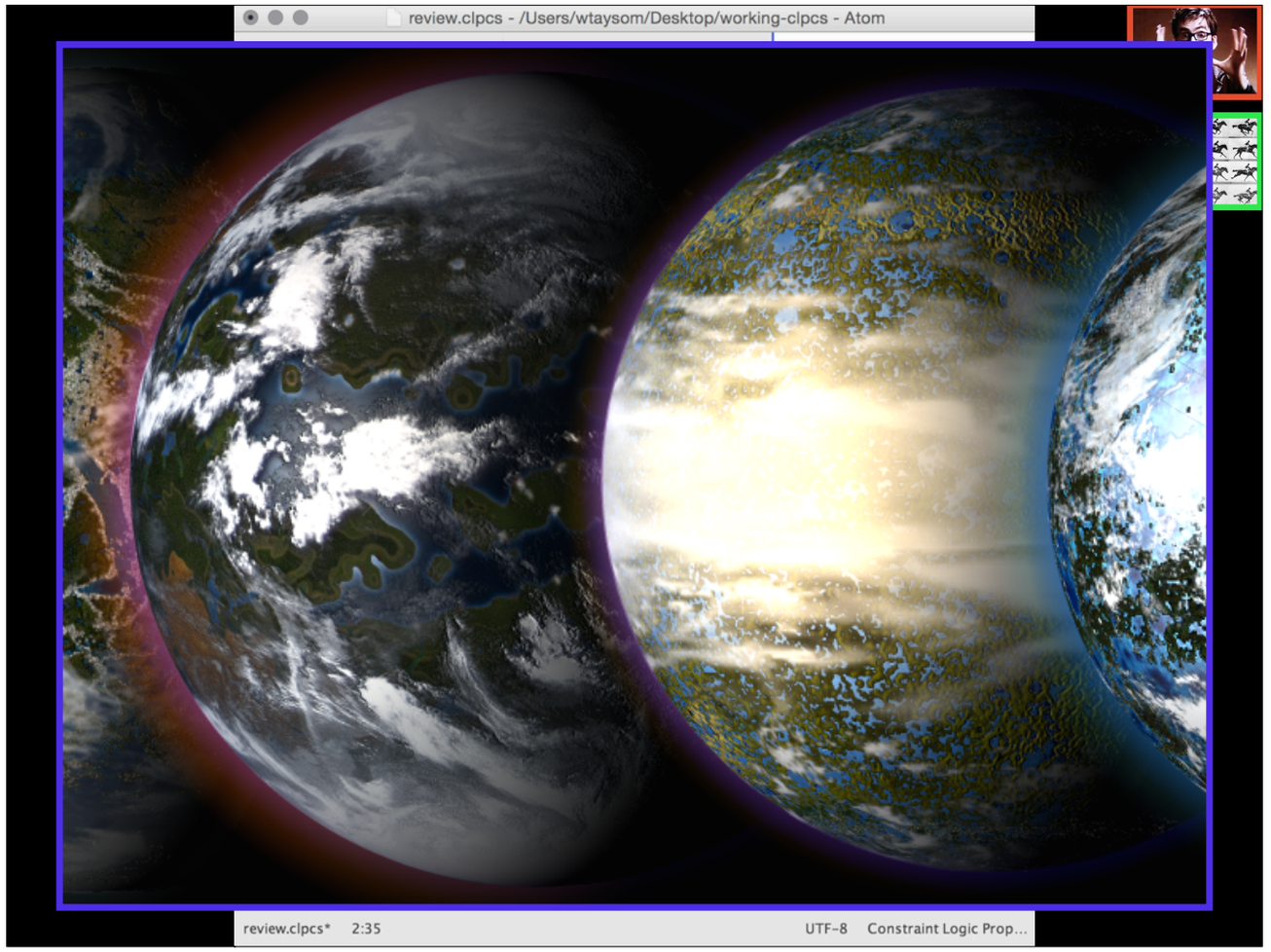
Ask a question twice, you'll get the same answer unless there are new facts written in the middle. If we quickly alternate between the two, the effect is to highlight the change. If we have a long list of answers, scrolling from one to the next gives us animation.



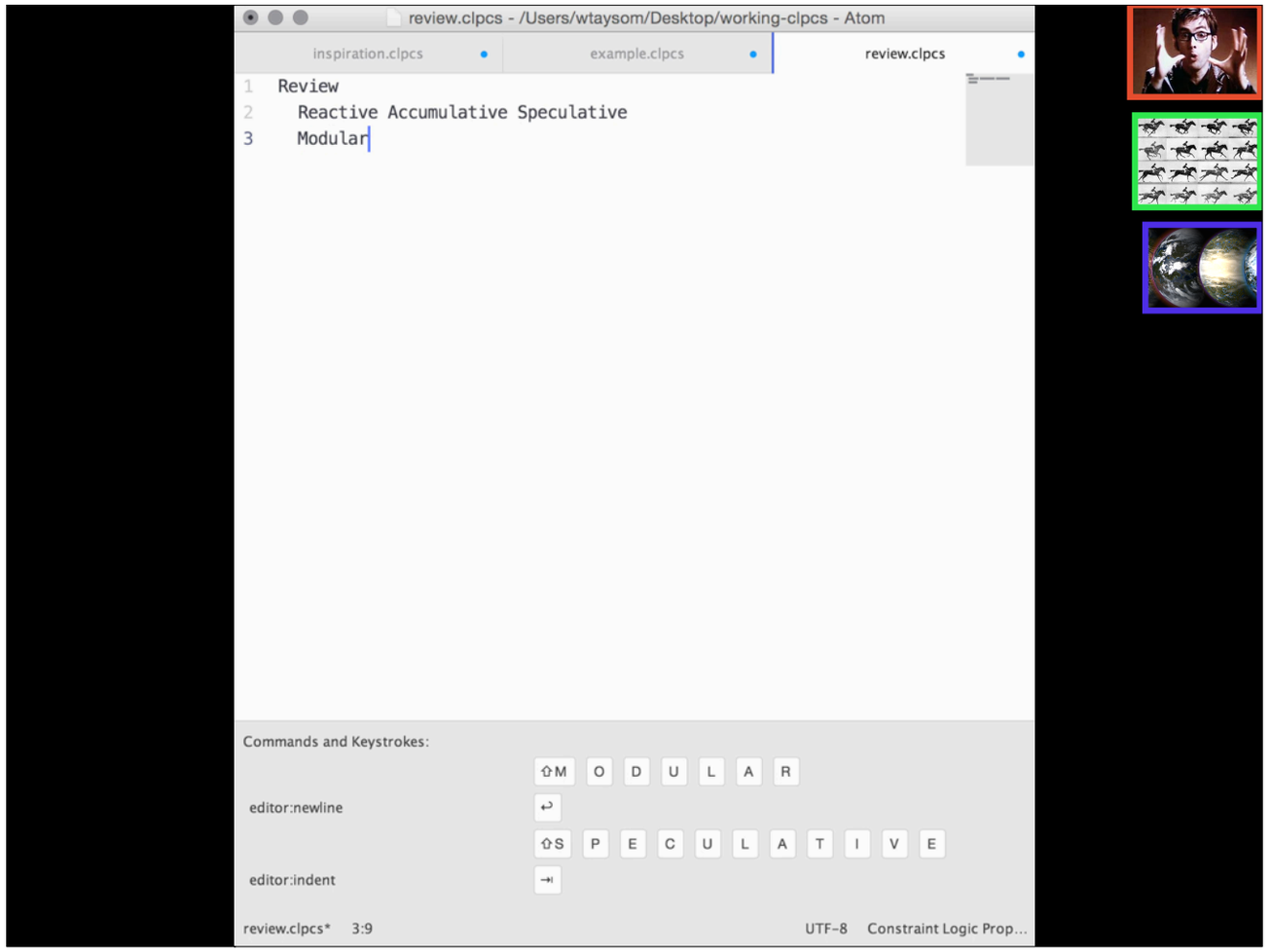
Analysts know some animator tricks. In our traffic table, we tried several values for `into Hilo` seeing changes in slow motion. In Excel to make transitions immediate, you start with a sheet. Make a copy. Update cells. Make another copy. Update cells. And so on. Switch between tabs in rapid succession:

the workbook becomes a flip book.

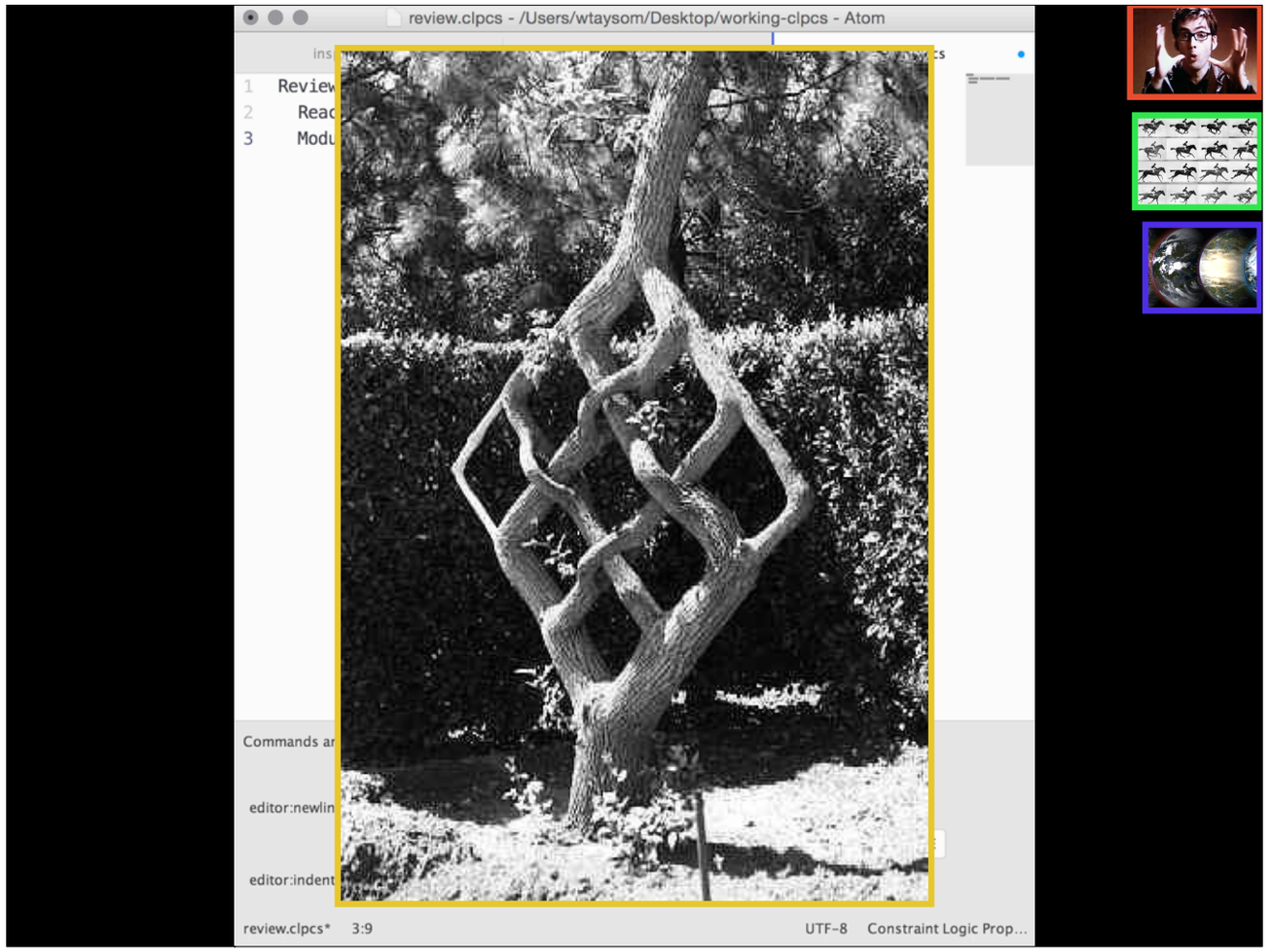
Of course the copies have no link back to the master, so it's easy to get out of sync. In the example, including a table made the link. Overriding customized it. Scope and variables abstracted over it.



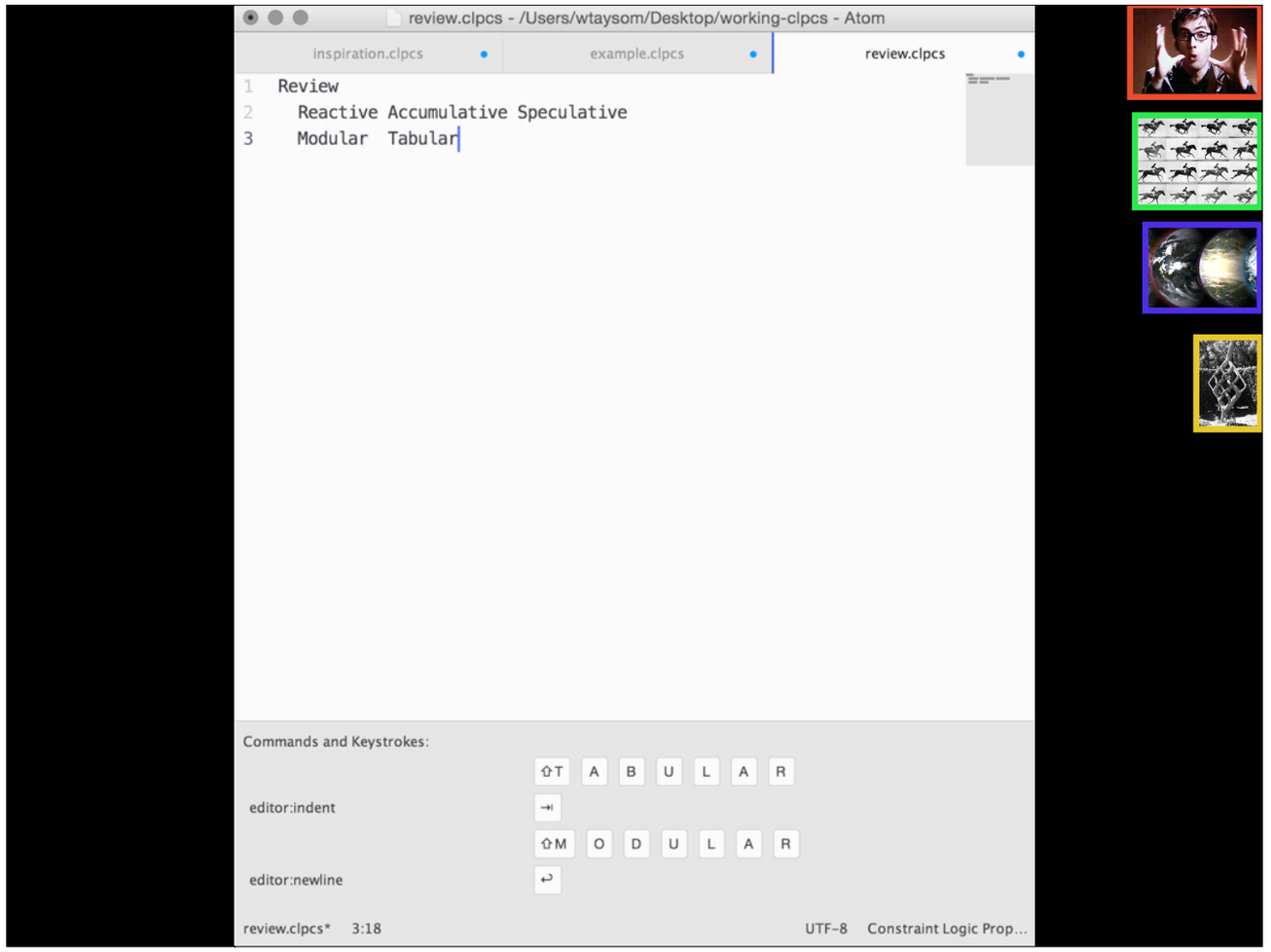
Then with auto-lifting propagation, we speculated over a range of possible worlds, possible futures. Layer on a user driven selection function: animation becomes interaction.



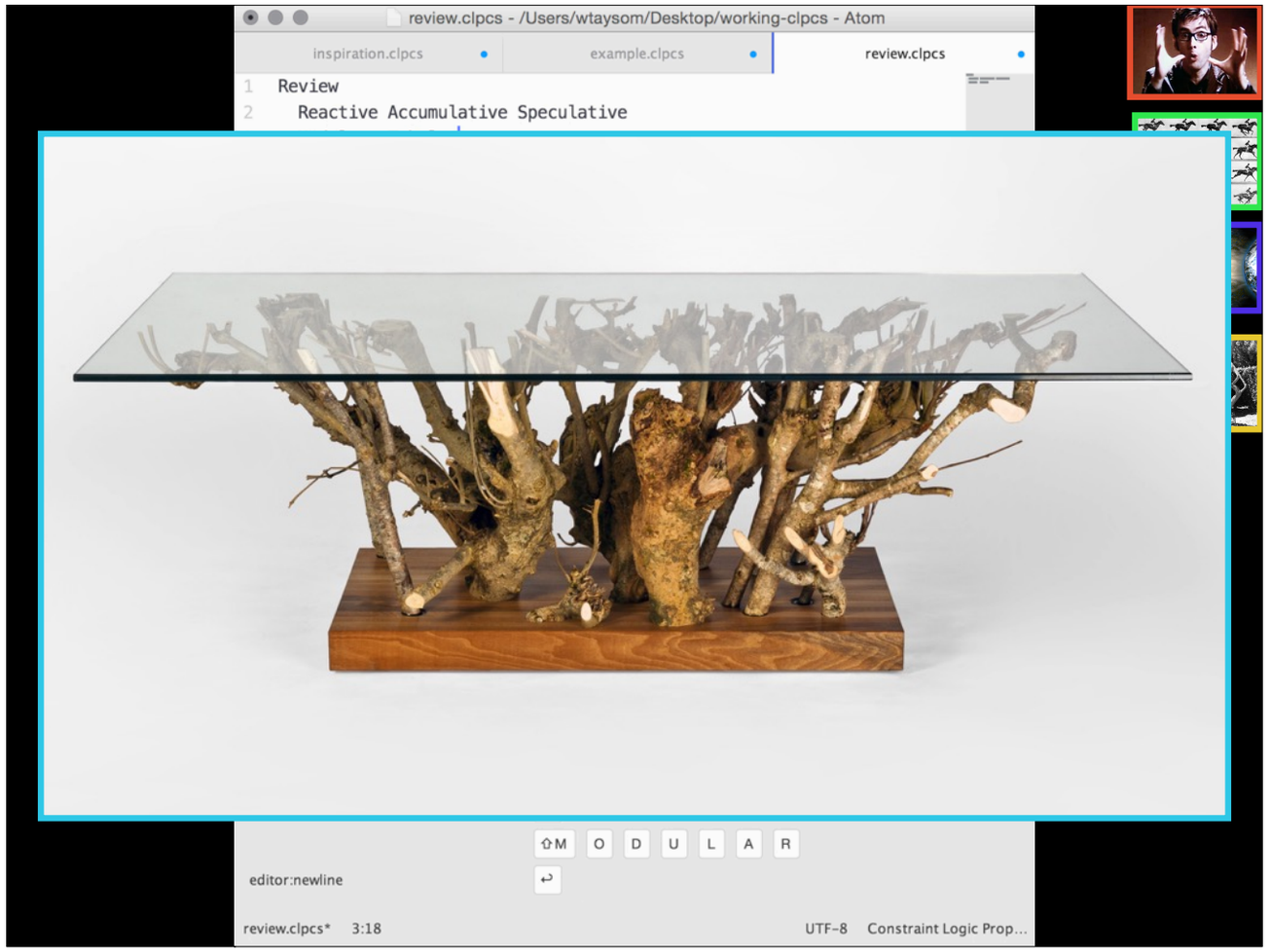
Convenient speculation comes from convenient modularity. In the example, we just indented for new scope.



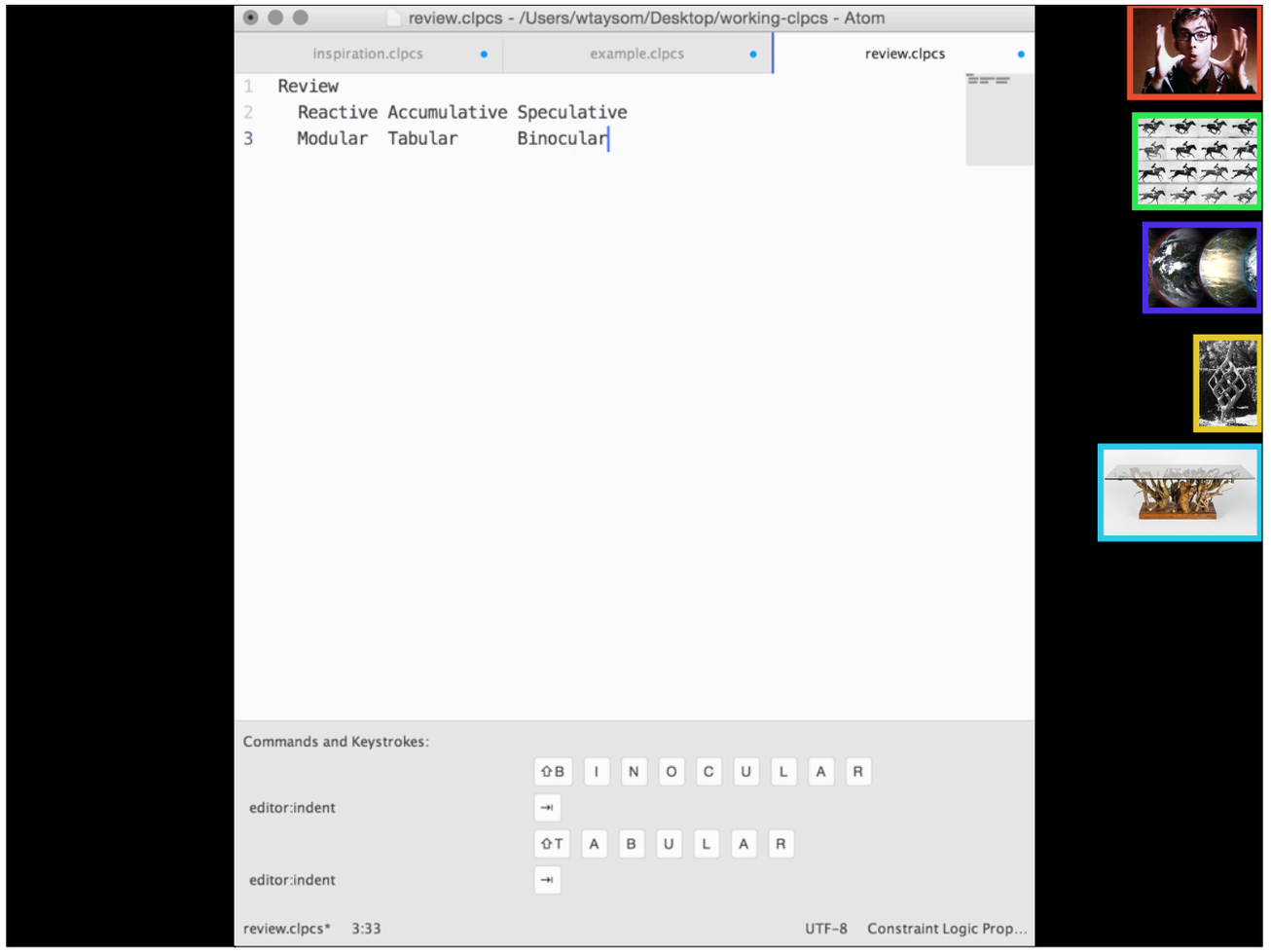
Includes graft branches: multiple inheritance. How's that for a diamond problem?



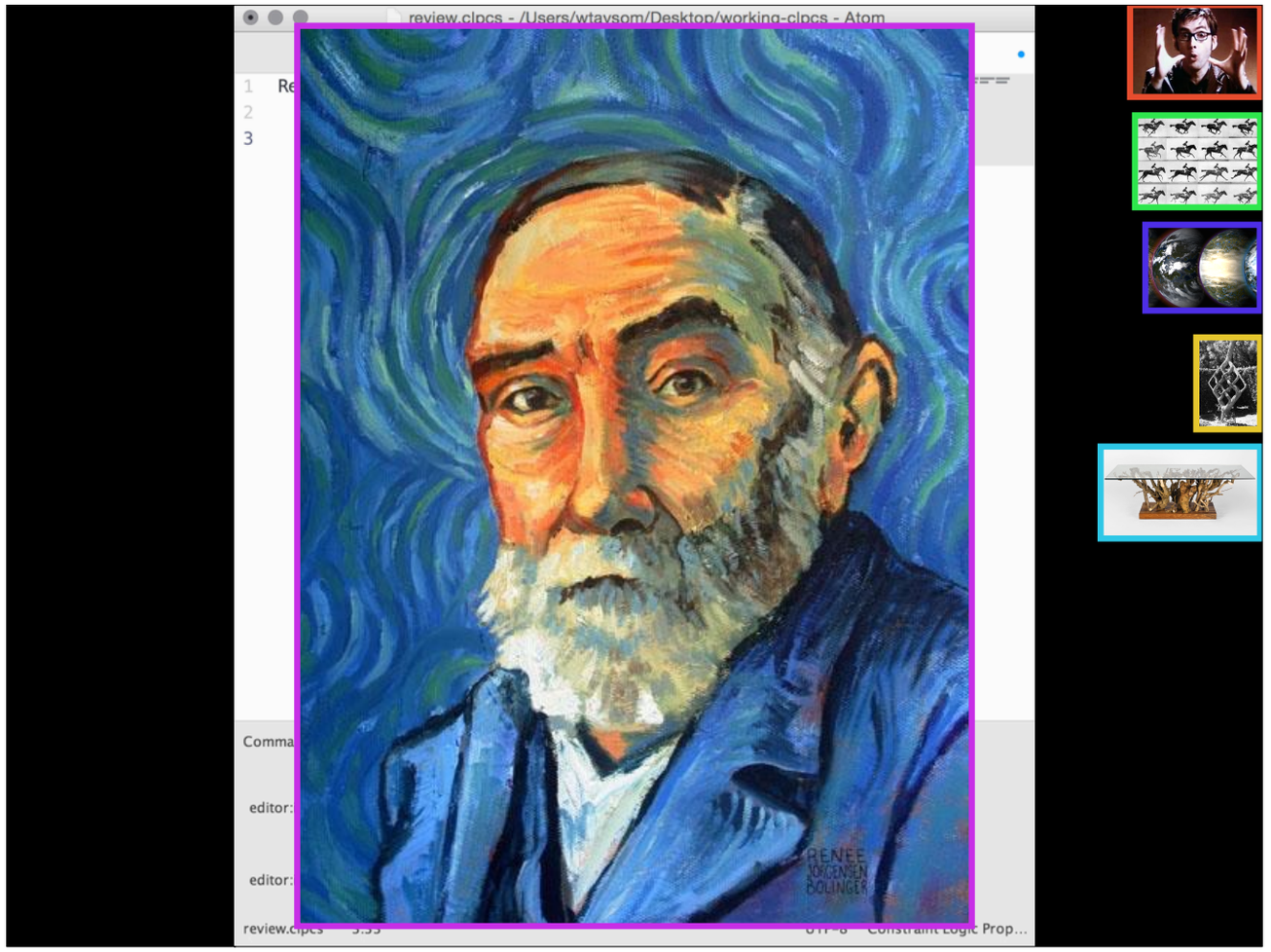
Excel isn't exactly a tool for dealing with tables. Each sheet is a grid. People may treat part like a table, you can mark a box of cells as a table with sorting and filters, but maintaining the illusion requires diligence. For instance, auto-fills let you see how each cell relates to others, but they hide the fact that a parameterized formula is being applied to a whole column. Other tools take tables more seriously, often at the cost of freeform flexibility.



Moreover, not all data is tabular. We have trees too. The example showed trees and tables trying to coexist with a tree understood as a grouped table.

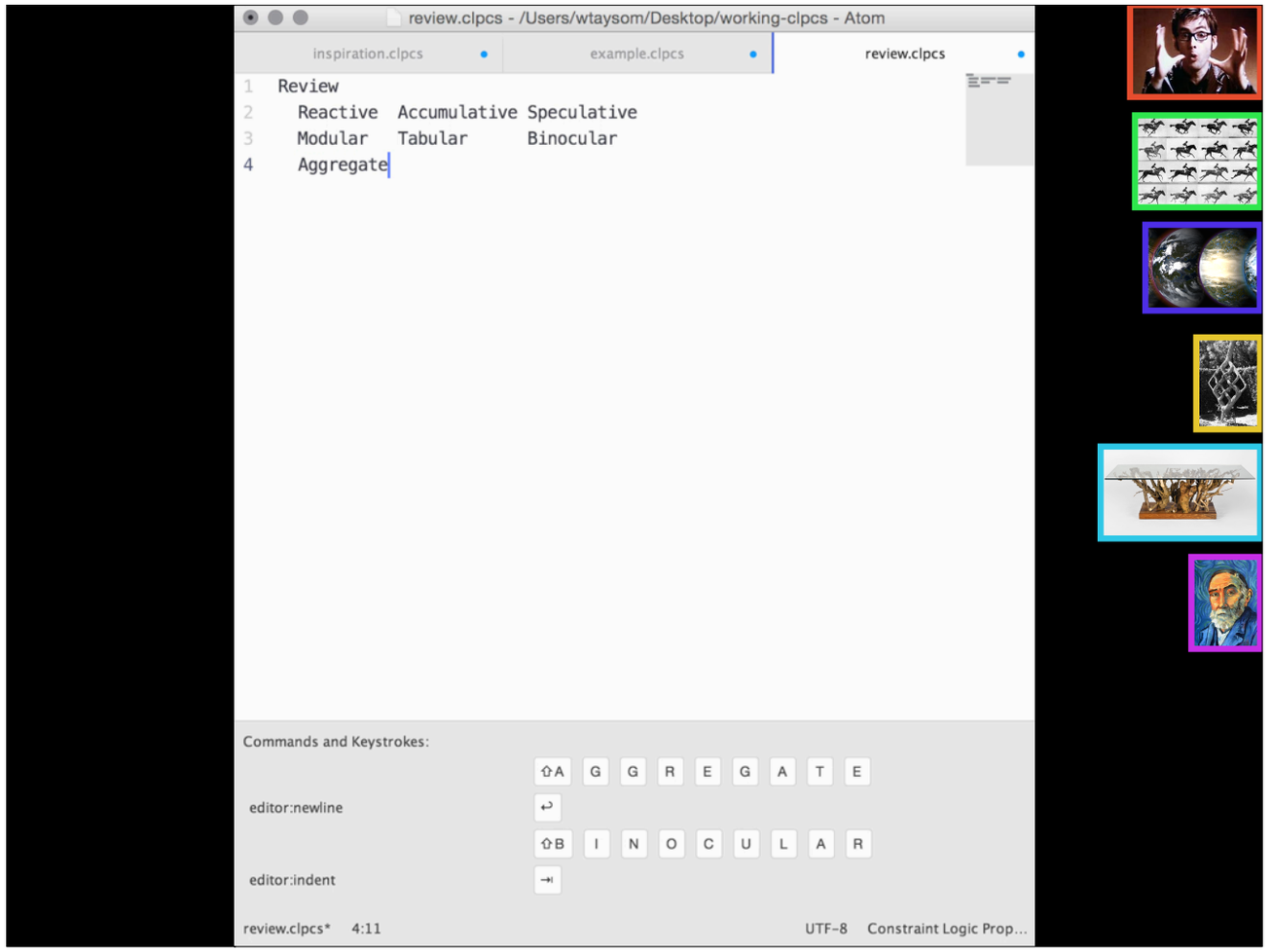


There's more than one way to look at a thing. We've seen many examples: group-by, more or less syntactic sugar, expanding an included table. One favorite is the convention of spreadsheet cells. Enter a formula: the sense of what you want. Hit return.

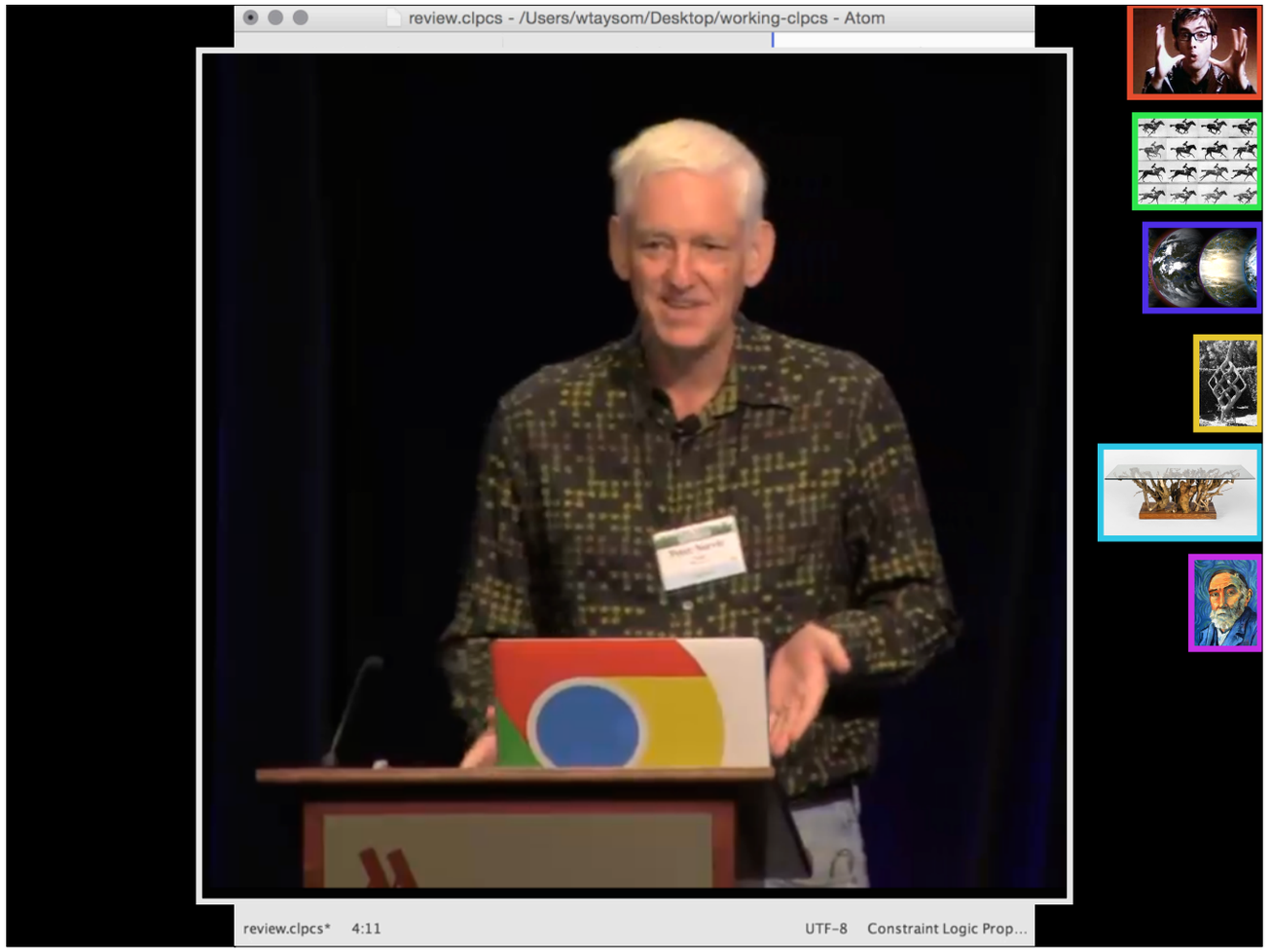


See a calculated value: the reference that sense expresses in context.

It's like binocular vision: from two slightly offset planar projections, we experience a 3D scene. Any representation that emphasizes some aspect does so by obscuring another. With multiple representations, we get to see around obstacles.



Last year at SPLASH,



Peter Norvig remarked that mathematical specifications don't naturally translate into C++.

Program Optimization: Crossing the Math/CS Chasm

$$\begin{aligned}\mathbf{e}_{\text{best}} &= \text{argmax}_{\mathbf{e}} p(\mathbf{e}|\mathbf{f}) \\ &= \text{argmax}_{\mathbf{e}} p(\mathbf{f}|\mathbf{e}) p_{\text{LM}}(\mathbf{e}) \omega^{\text{length}(\mathbf{e})}\end{aligned}$$

- A few lines of math
- Tens of thousands of lines of C++

No surprise there. However, with a good declarative language, the translation can be even better than the math.

Declarative Languages for AI, ML Applications

- DYNA (Jason Eisner)
  Probabilistic Context Free Parser:

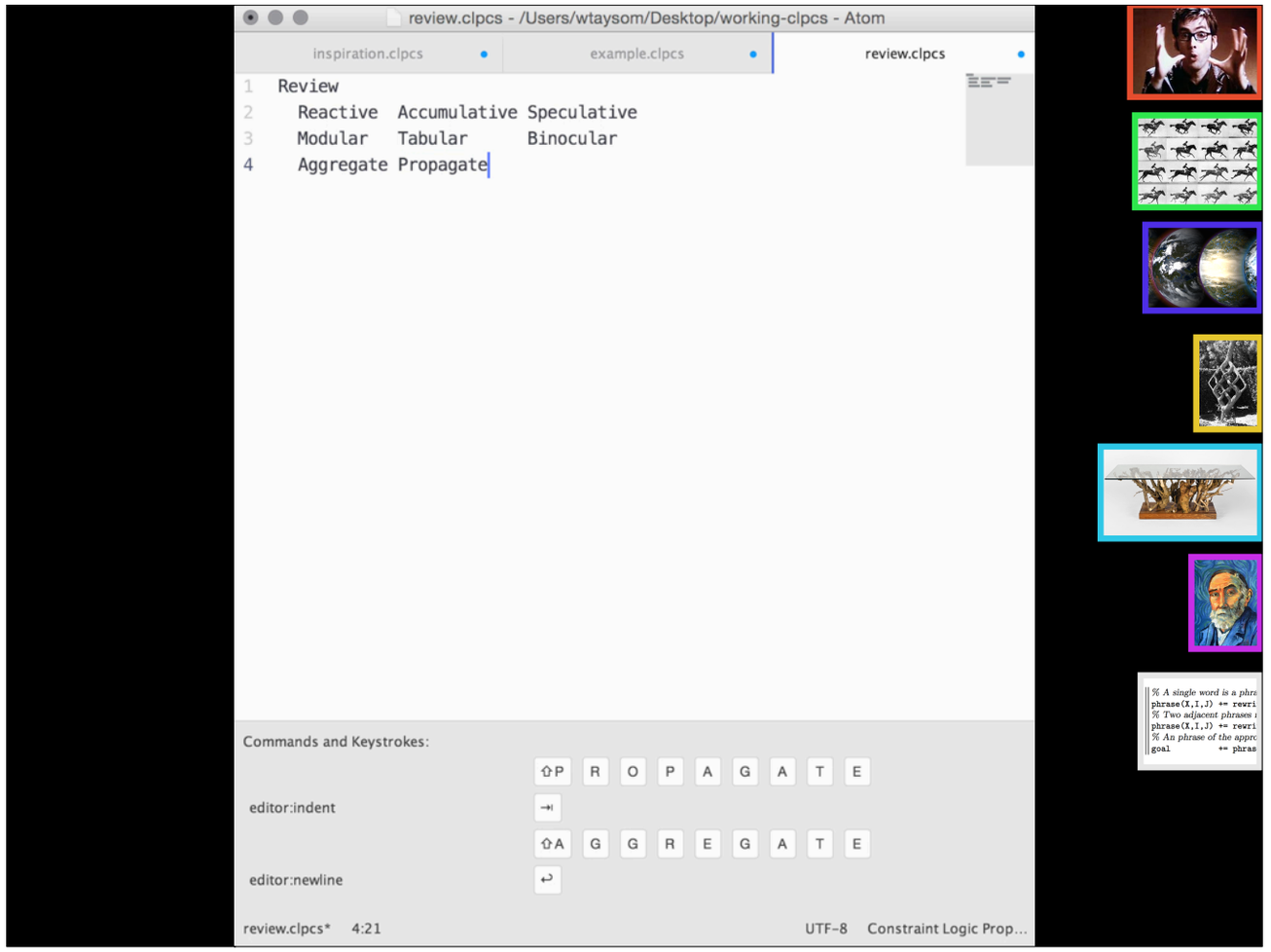
```
% A single word is a phrase (given an appropriate grammar rule).
phrase(X,I,J) += rewrite(X,W) * word(W,I,J).
% Two adjacent phrases make a wider phrase (given an appropriate rule).
phrase(X,I,J) += rewrite(X,Y,Z) * phrase(Y,I,Mid) * phrase(Z,Mid,J).
% An phrase of the appropriate type covering the whole sentence is a parse.
goal          += phrase(start_nonterminal,0,length).
```

```
rewrite("S","NP","VP")=0.9
word("spring",5,6)=1
start_nonterminal="S"
```

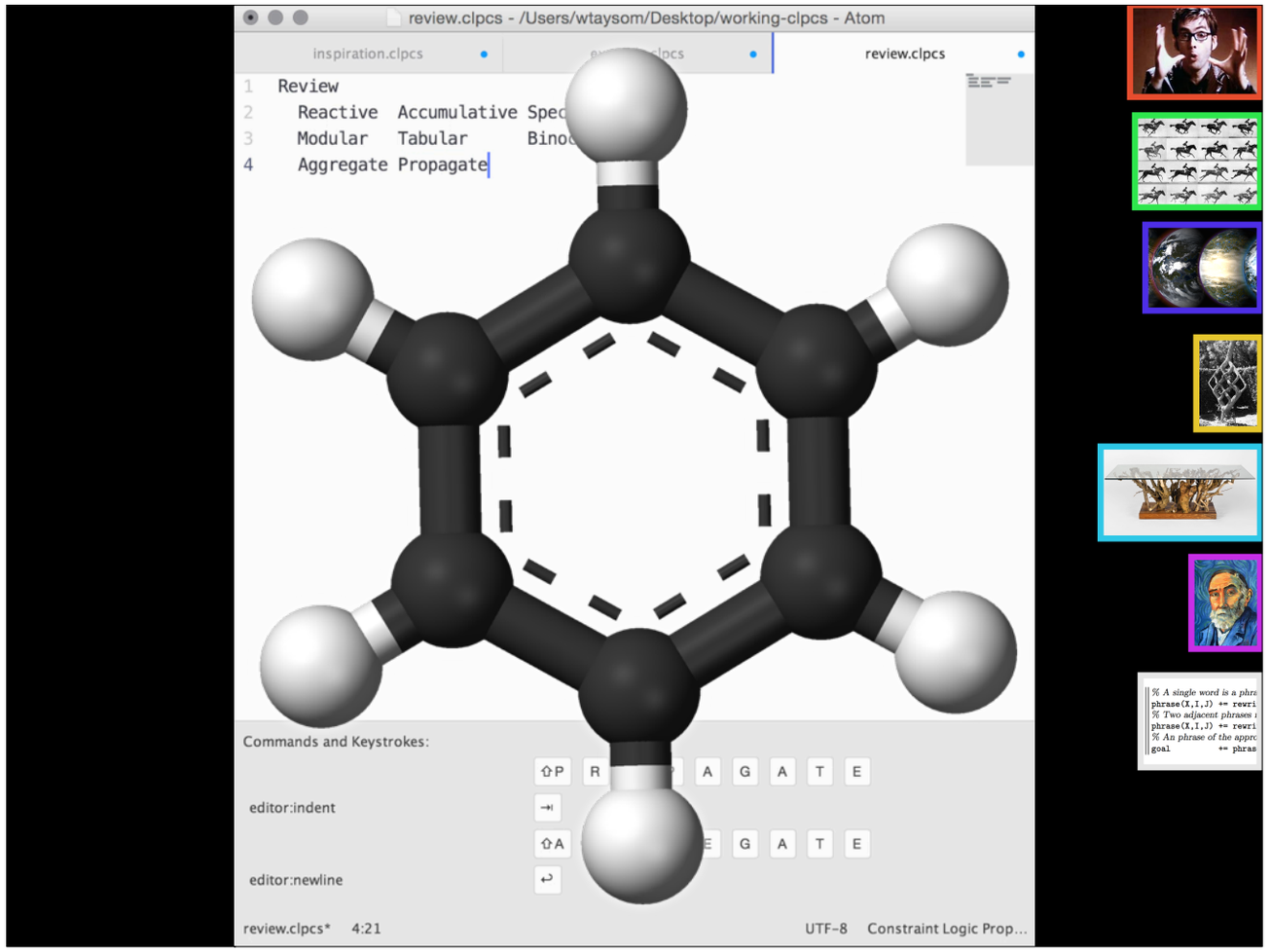
At seeing this slide, I was awestruck: notation practically perfect in every way. Ever since, I've been pondering why I reacted so strongly.

```
% A single word is a phra
phrase(X,I,J) += rewri
% Two adjacent phrases
phrase(X,I,J) += rewri
% An phrase of the appro
goal          += phras
```

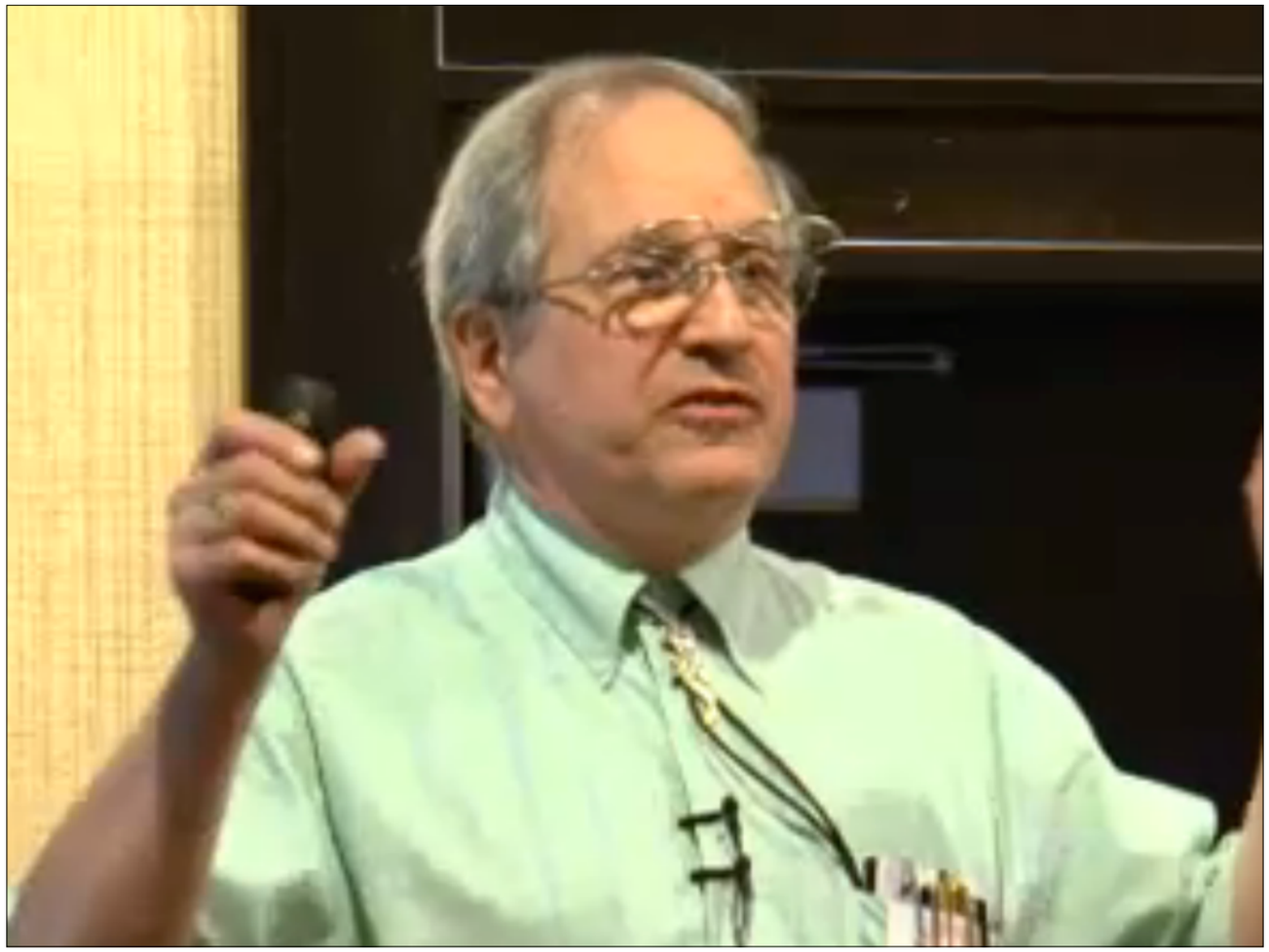
Earlier I said that aggregators fold over all unifying substitutions. By listing those substitutions, we enumerate all the terms in a family of sums. Unification gives us an unfolded intermediate representation. Group-by the lefthand side to see an expansion of each sum. The beauty comes from having both intensional and extensional views with regular unification serving as the lens.



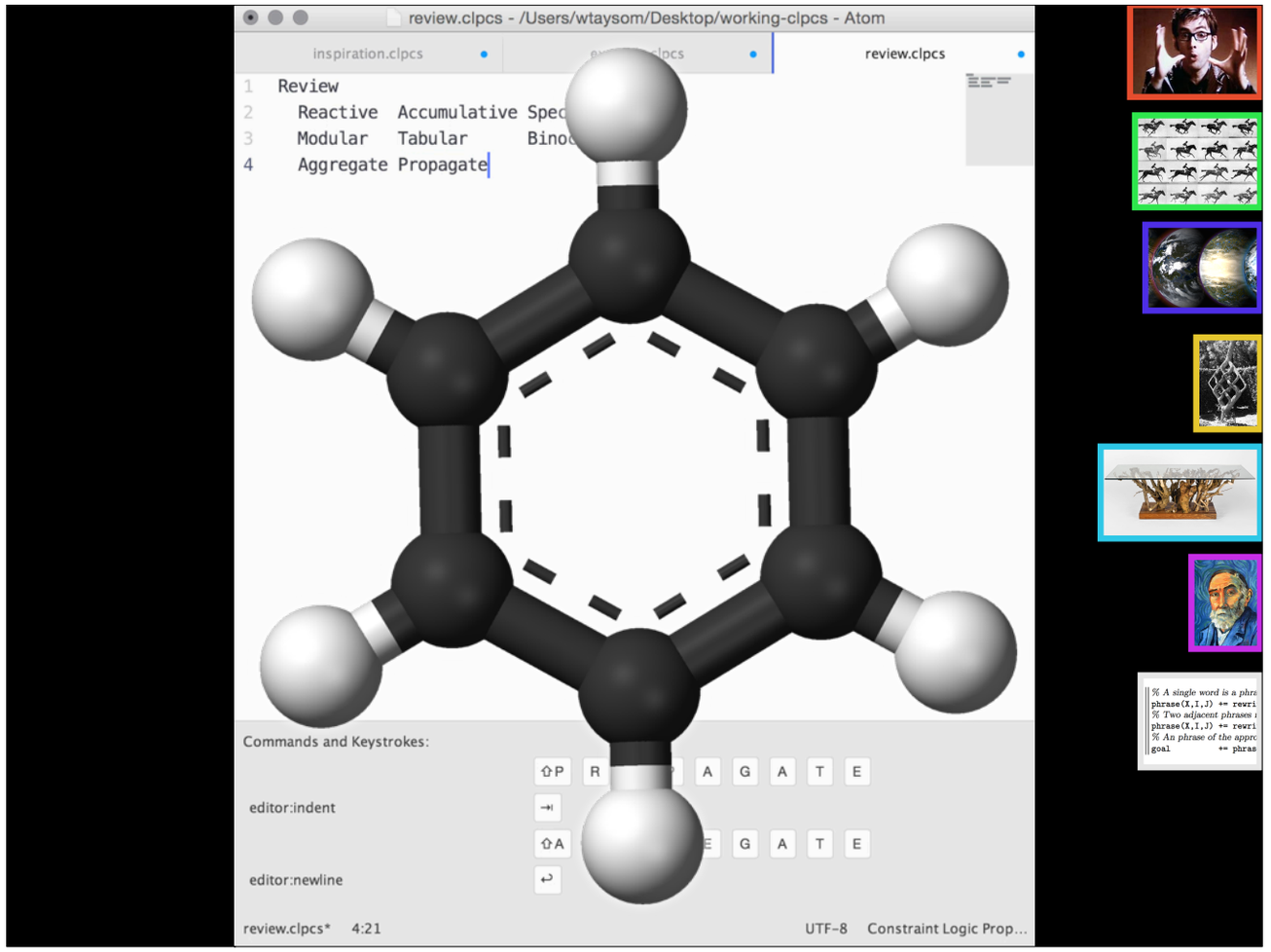
An aggregator puts each aggregate at the hub of a network with terms spreading out as spokes.



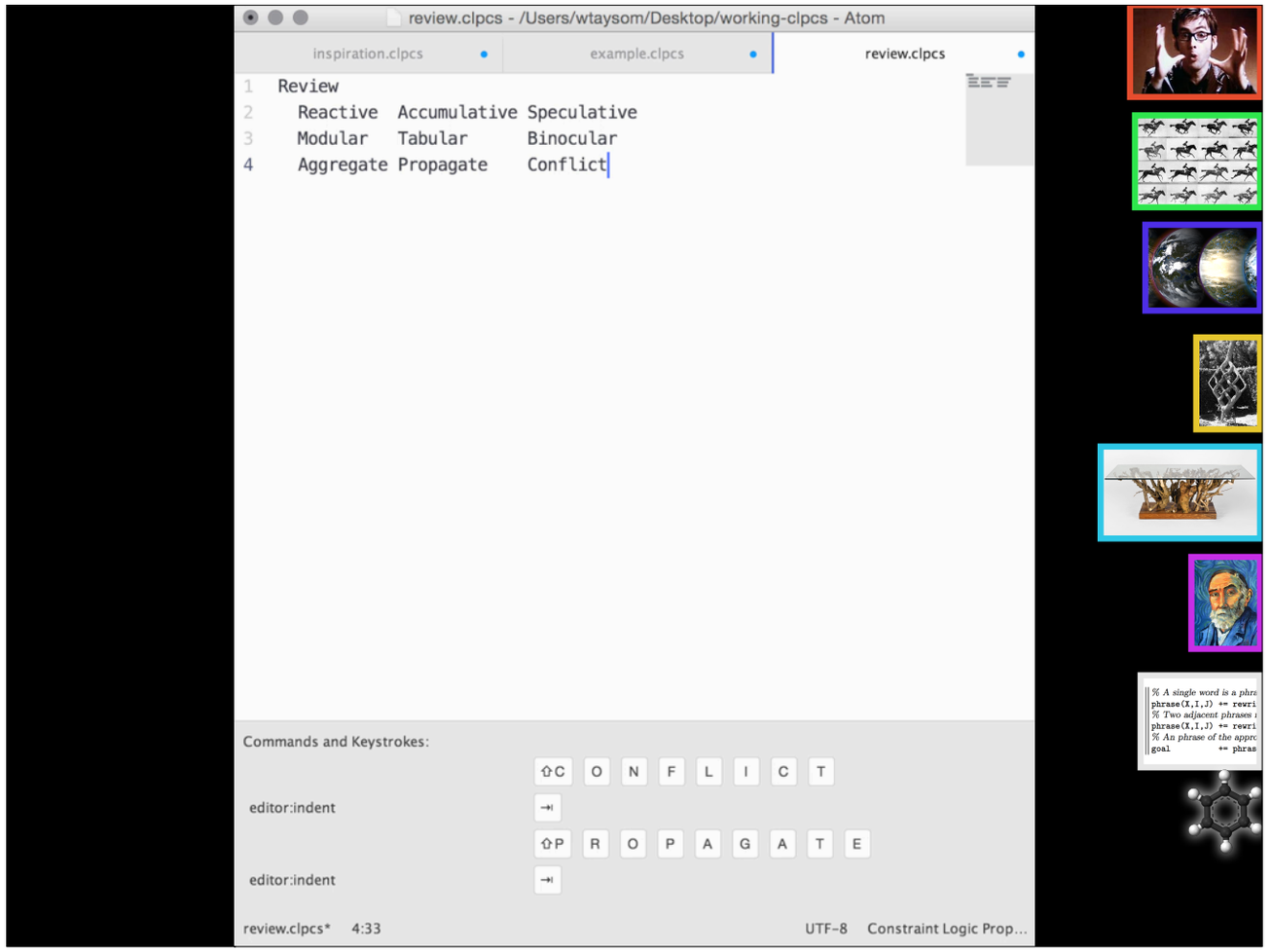
In our example, these spokes joined other hubs forming a benzene shaped ring. We propagated numbers and bounds along the network.



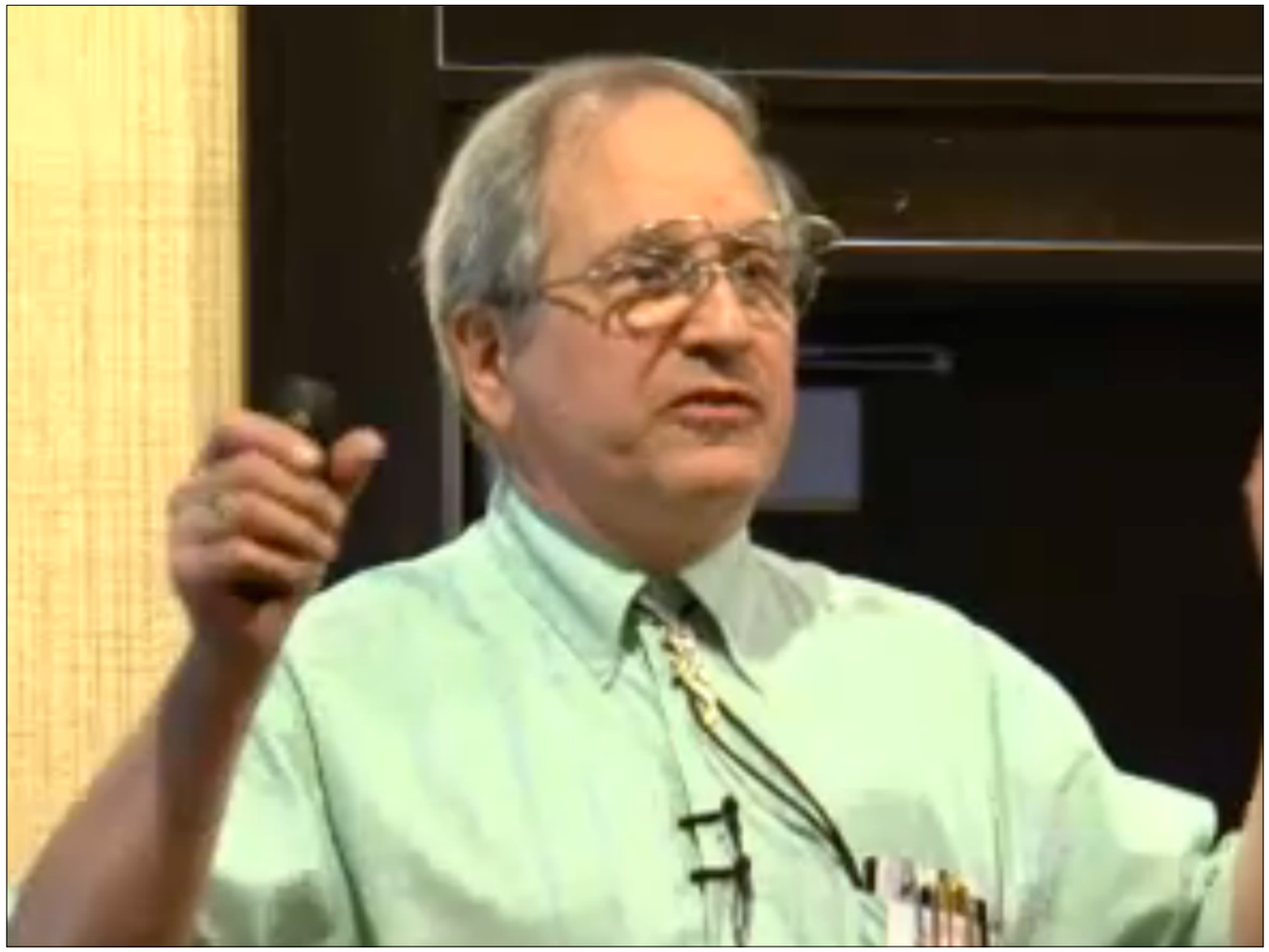
When Gerald Sussman talked about propagation at Strange Loop in 2011, he propagated derivatives, matrices, voltages. "The value here is that I'm allowed to dynamically change the meaning of all my operators, add new things that they can do." One new thing was to have a propagated value track its journey through the network, its province.



I've long used a weak form of this in Object Oriented settings. Proxy an object so that all its methods return a proxy whose methods all return a proxy, and so on. By the end, you have an image of data downstream from the original object, good for measuring how tangled a system is. These tracer doodads are good for view maintenance too: collect all the metadata necessary to map a view-update back to its model. Back-propagation makes tracing even better: given a wrong value find causes.



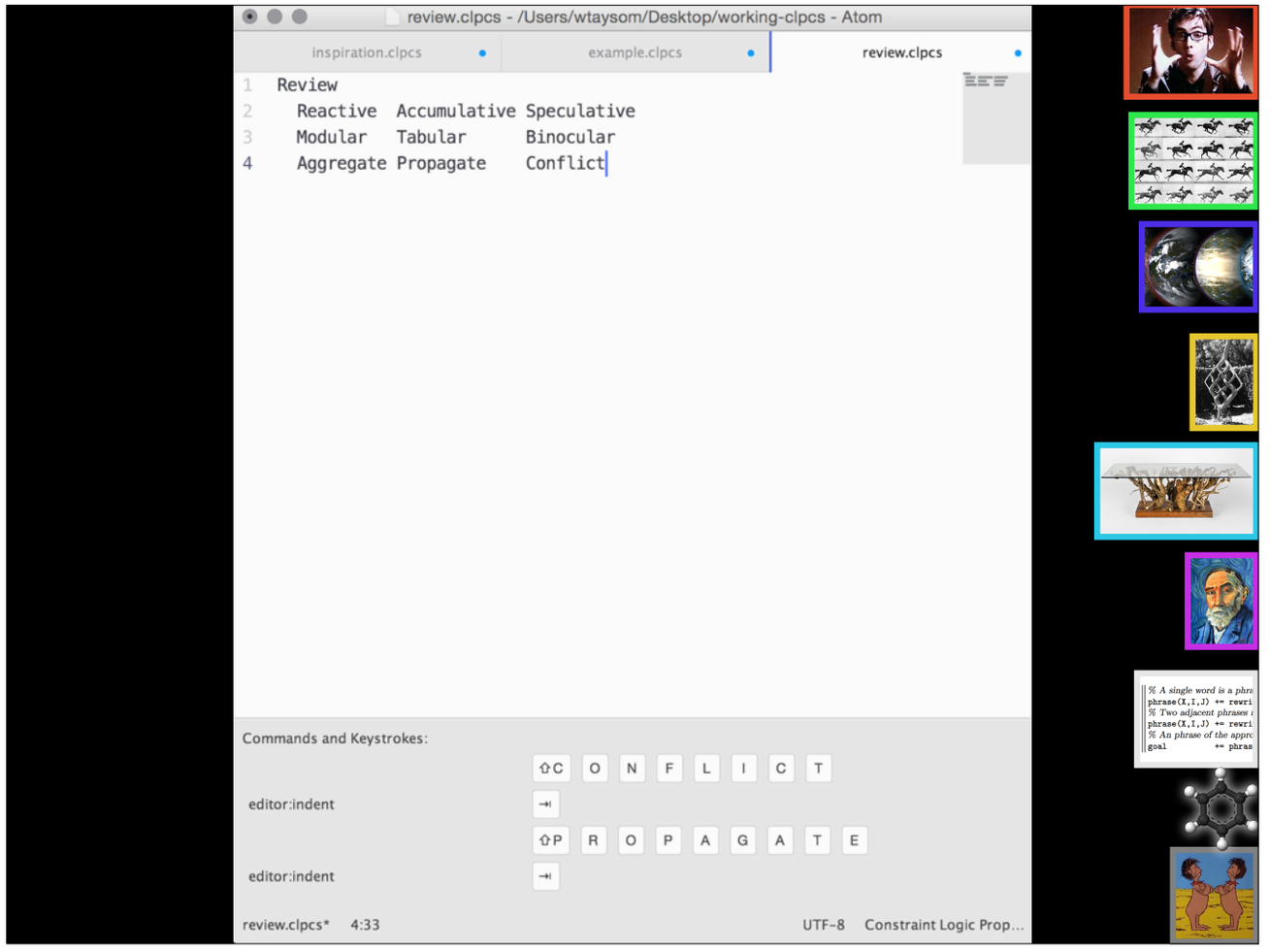
Sussman went on to say, "I can pay a belief in correctness for tremendous flexibly. ... The problem facing us as computer engineers is not correctness but evolvability." The whole purpose of thinking things through is to go from muddled, confused, contradictory ideas to clear, orderly, consistent ones.



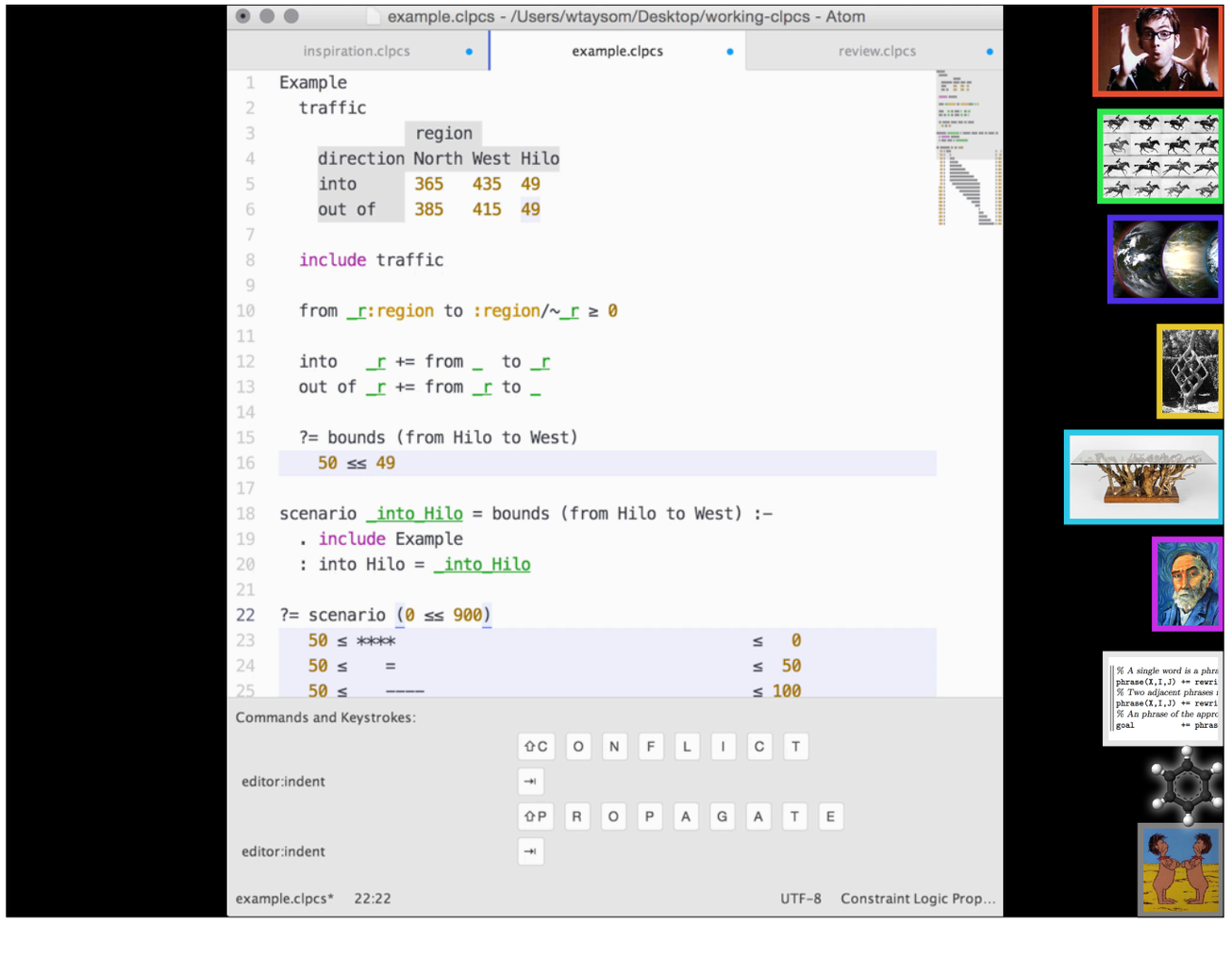
Sussman went on to say, "I can pay a belief in correctness for tremendous flexibly. ... The problem facing us as computer engineers is not correctness but evolvability." The whole purpose of thinking things through is to go from muddled, confused, contradictory ideas to clear, orderly, consistent ones.

The degree to which our software tools can model confusion matches their usefulness in conceptualizing, brainstorming, coming to understand. A perfect language, one that only compiles correct programs, is a poor fit for a world full of inconsistency and conflict.

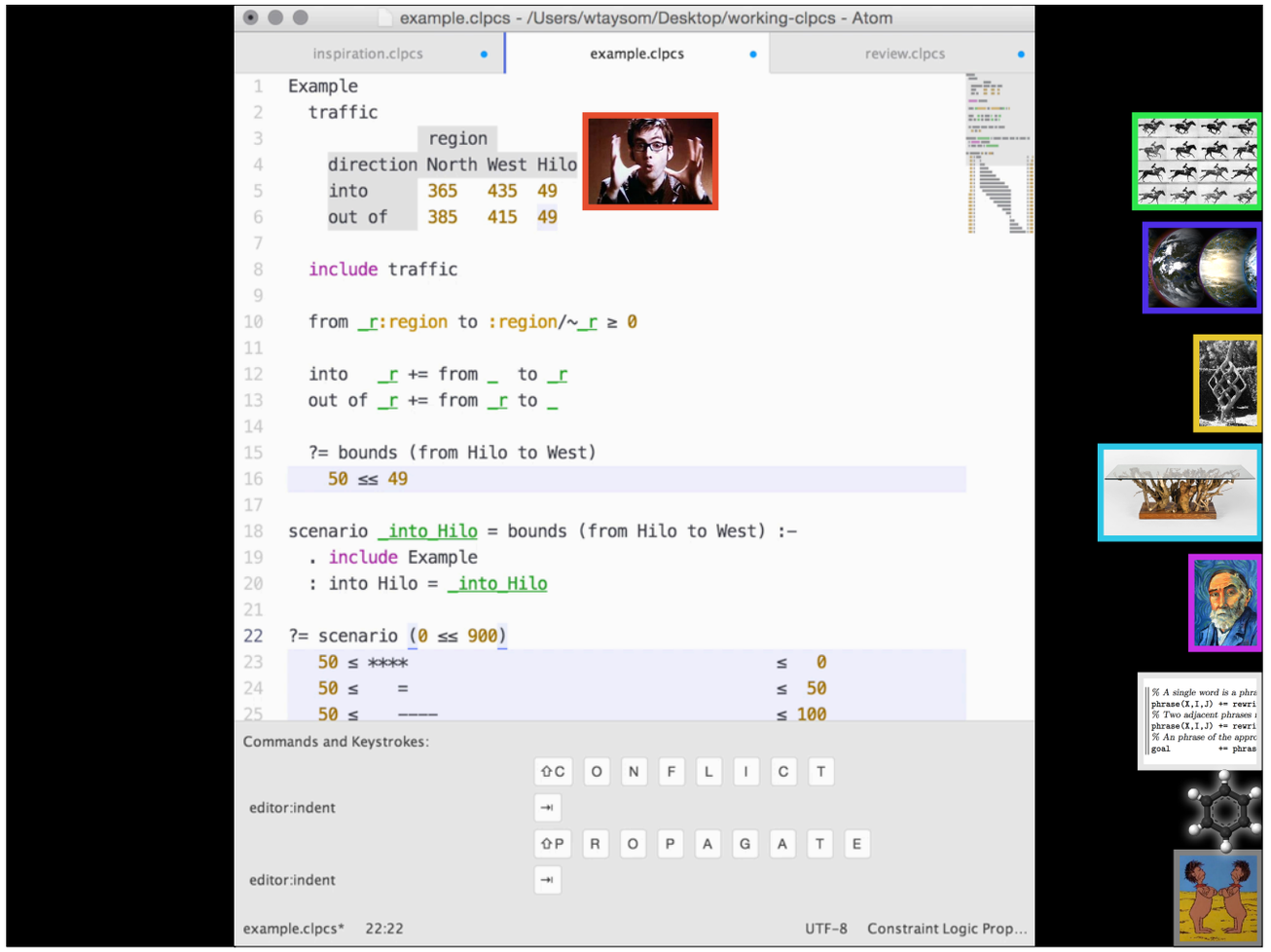
I'm not saying we should all be writing Ruby. A better choice may be a pure language in which chaos is a first class construct. In the example, some variables had multiple values. Instead of the conventional non-deterministic OR semantics, I experimented with a sort of over-deterministic AND semantics.



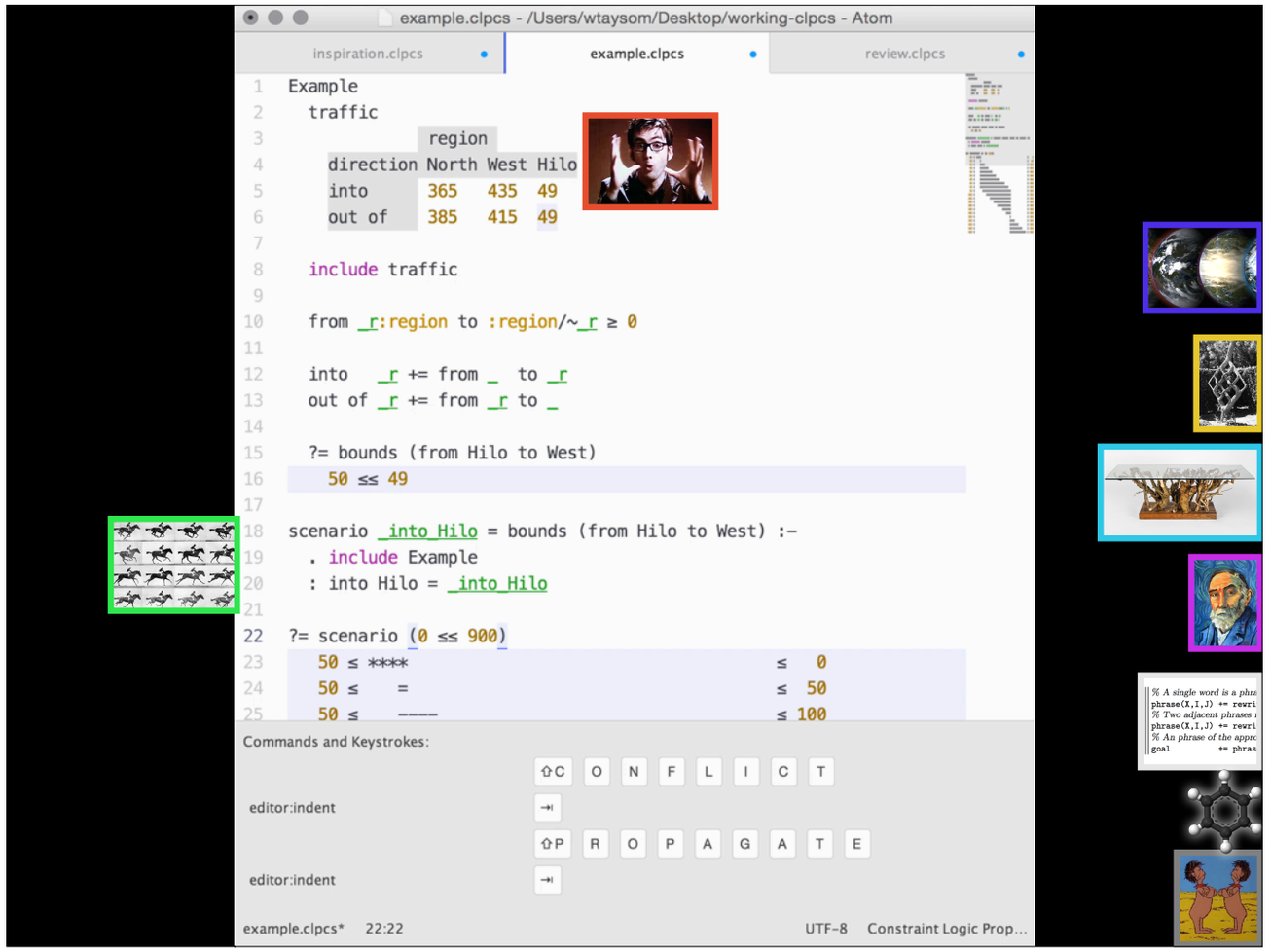
Let's take one last look at the Example.



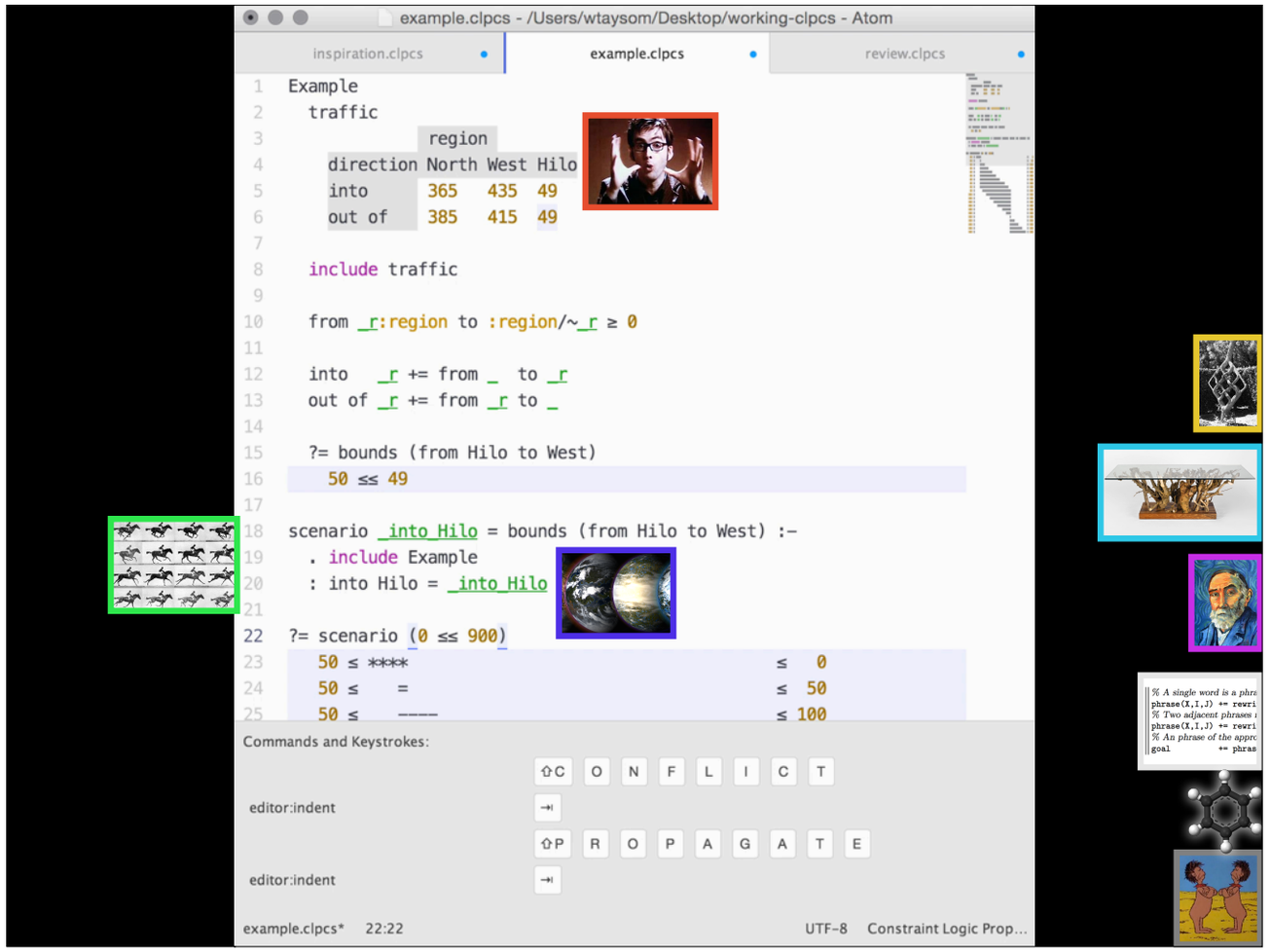
Is this the future of programming? I like the economy of expression. From a table, some equations, and a simple question, we get a nuanced answer. There may indeed be a wide and productive bridge joining logic programming to generalized propagation. This is not it. This is a curiosity, hopefully, a thought provoking one with qualities that future tools will share.



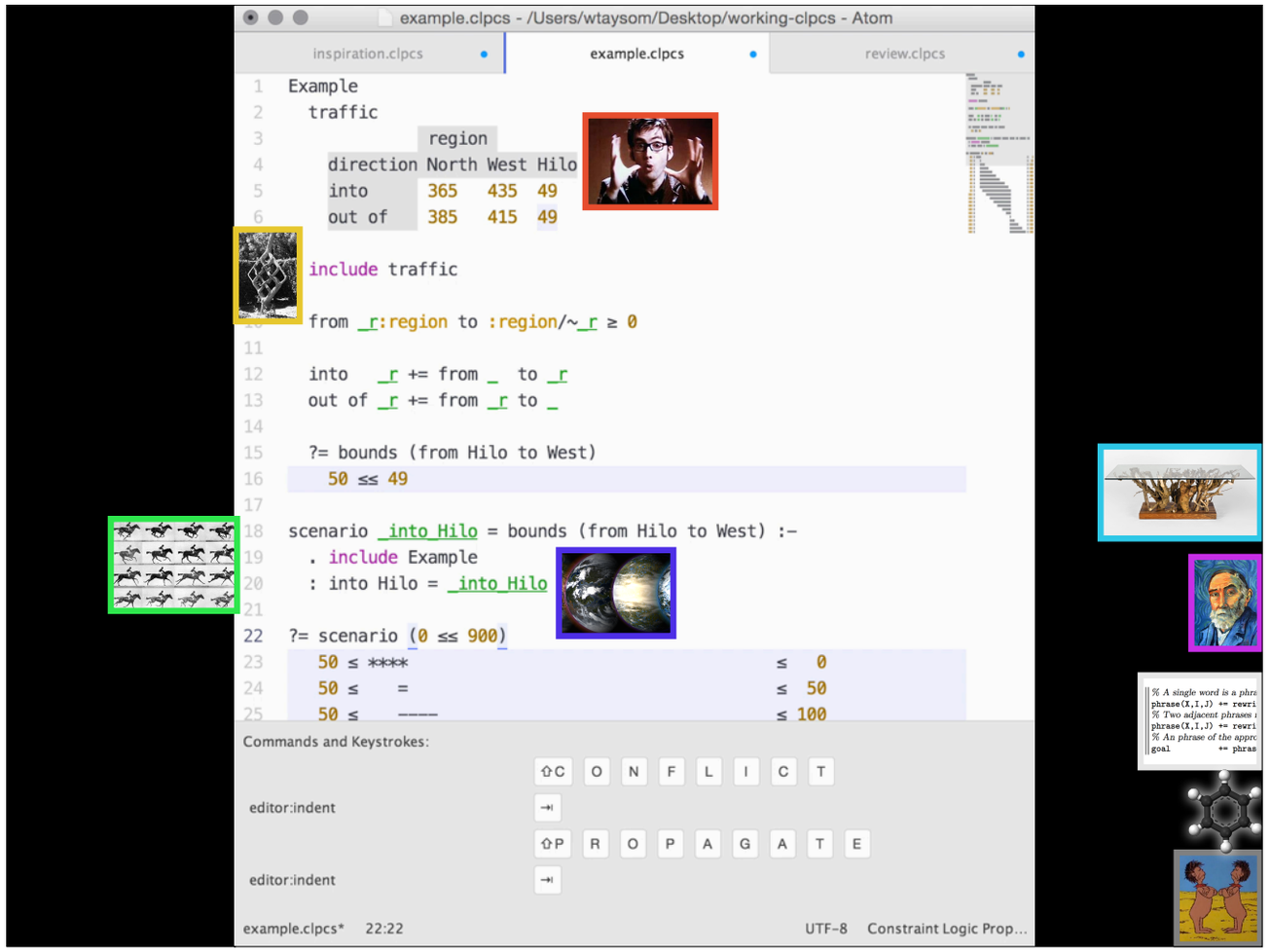
Reactive wibbles wobble.



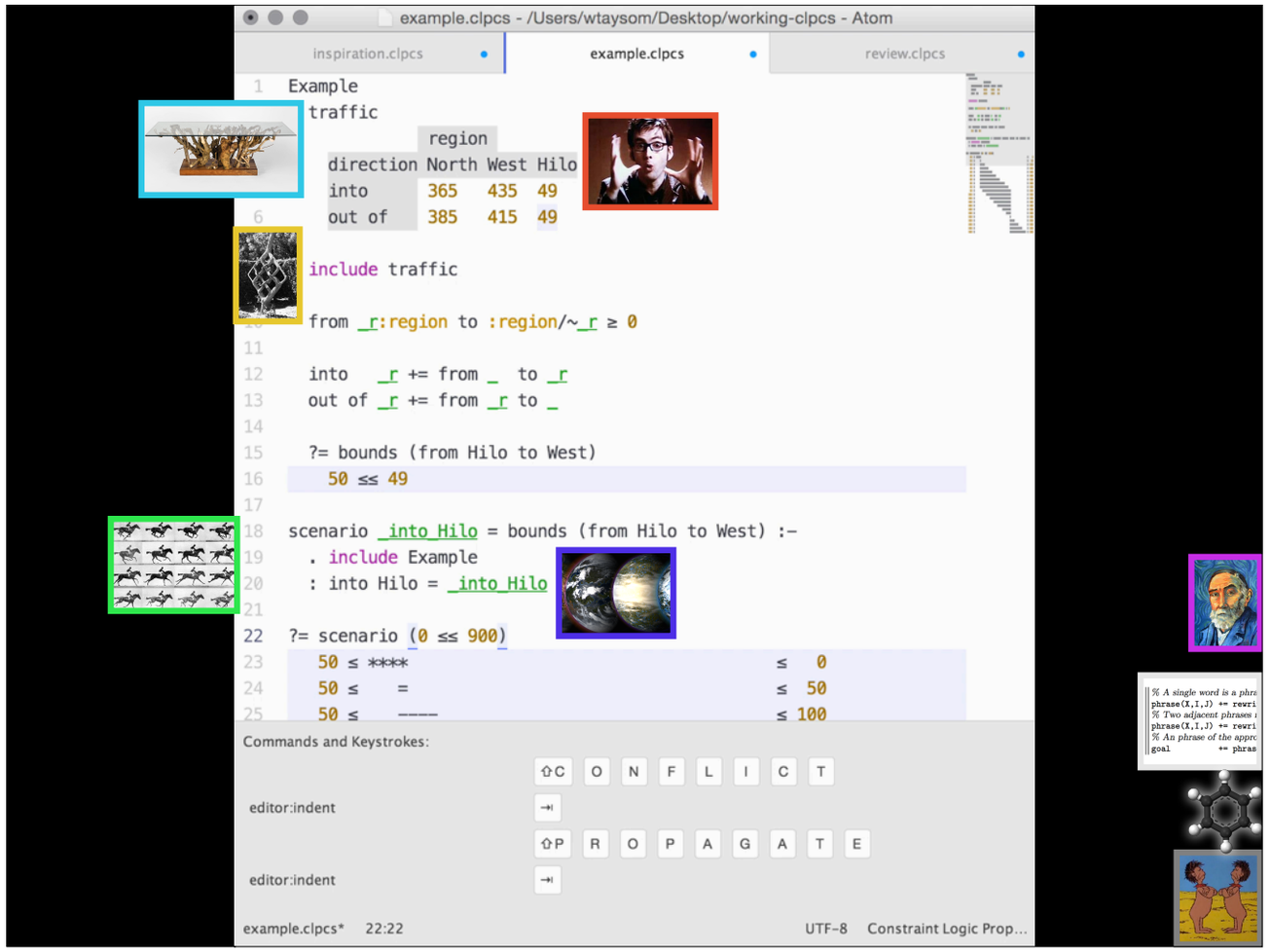
Accumulative time as space.



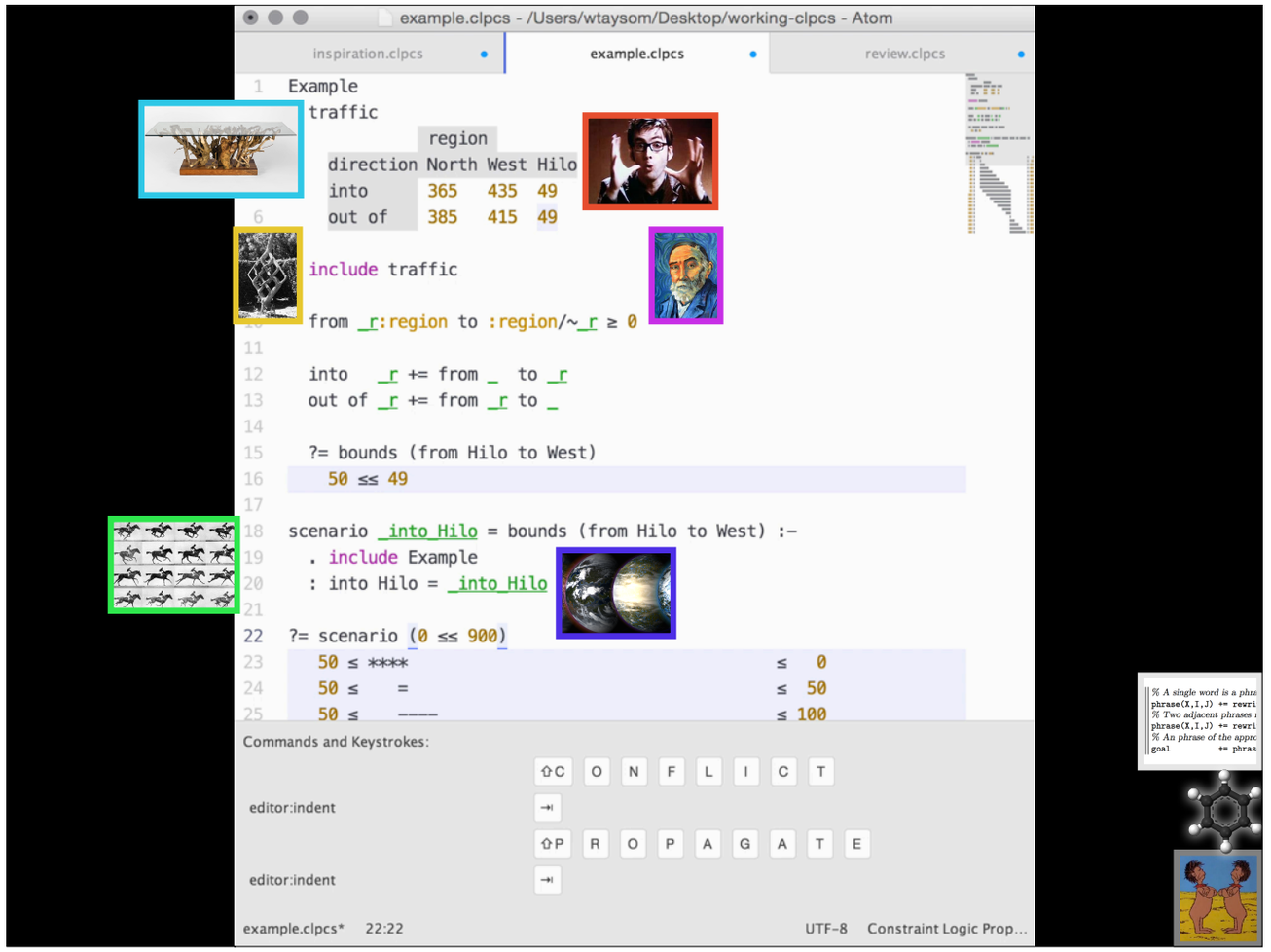
Speculative possible futures.



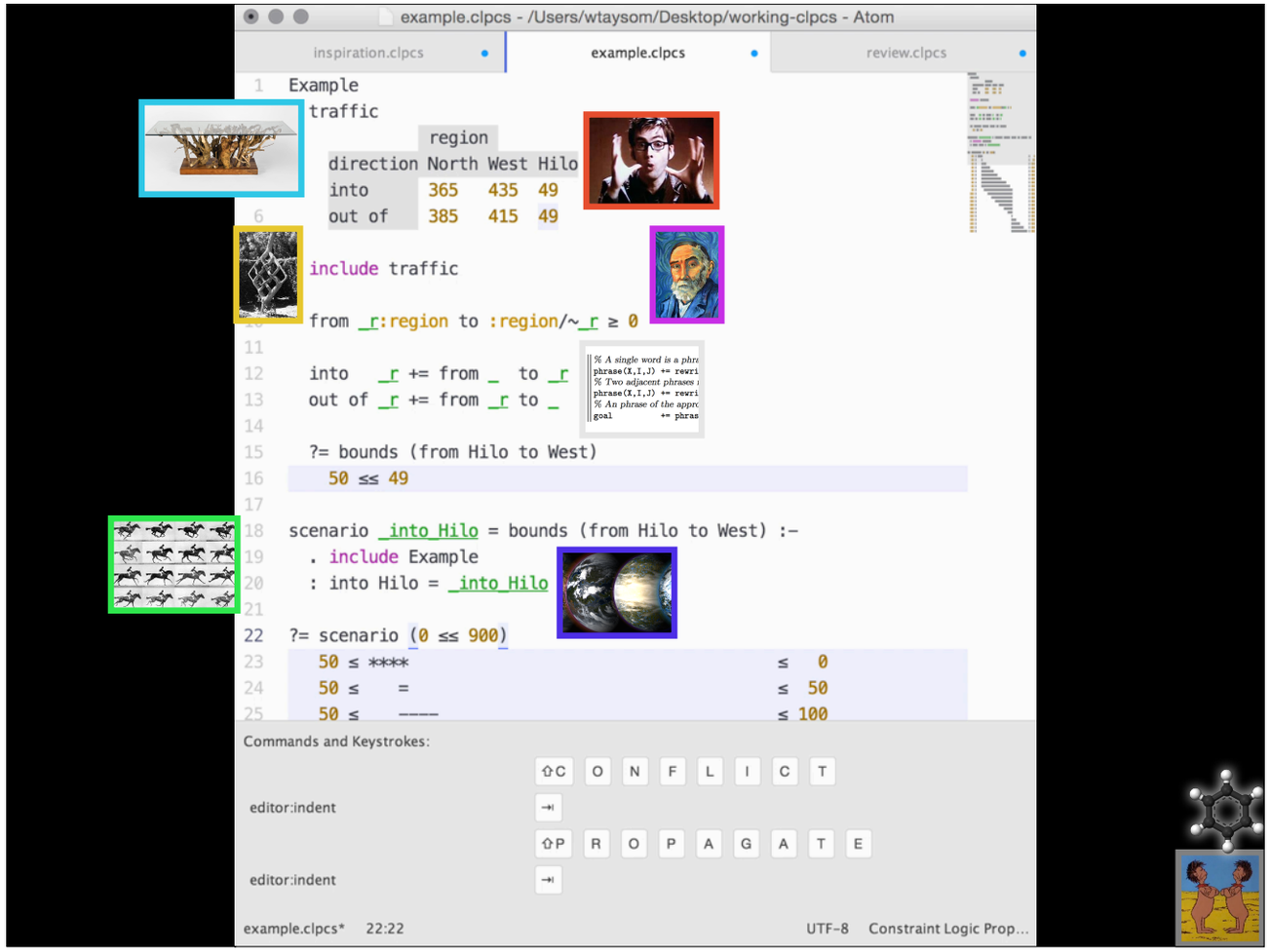
Modular indent and include.



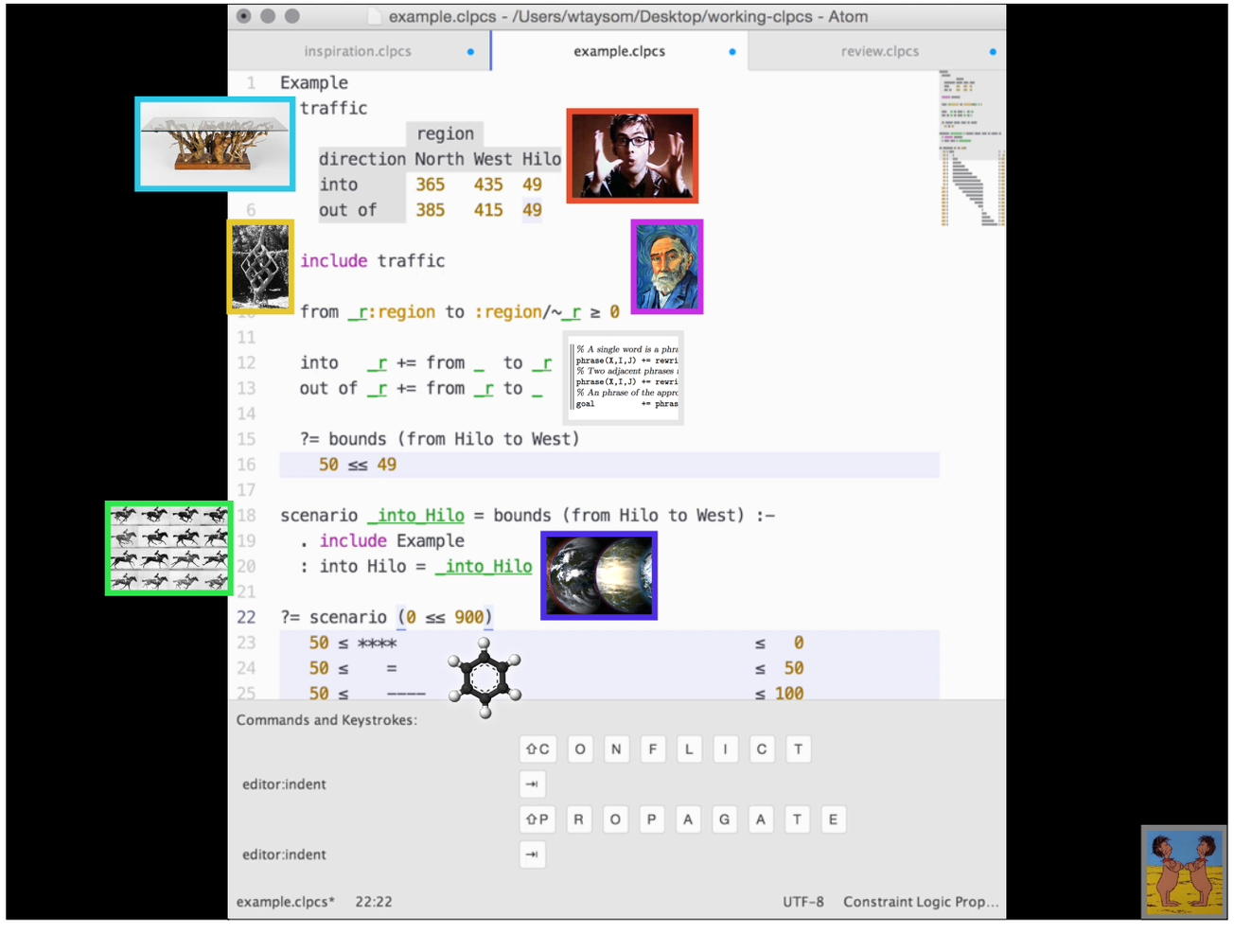
Tabular trees group tables.



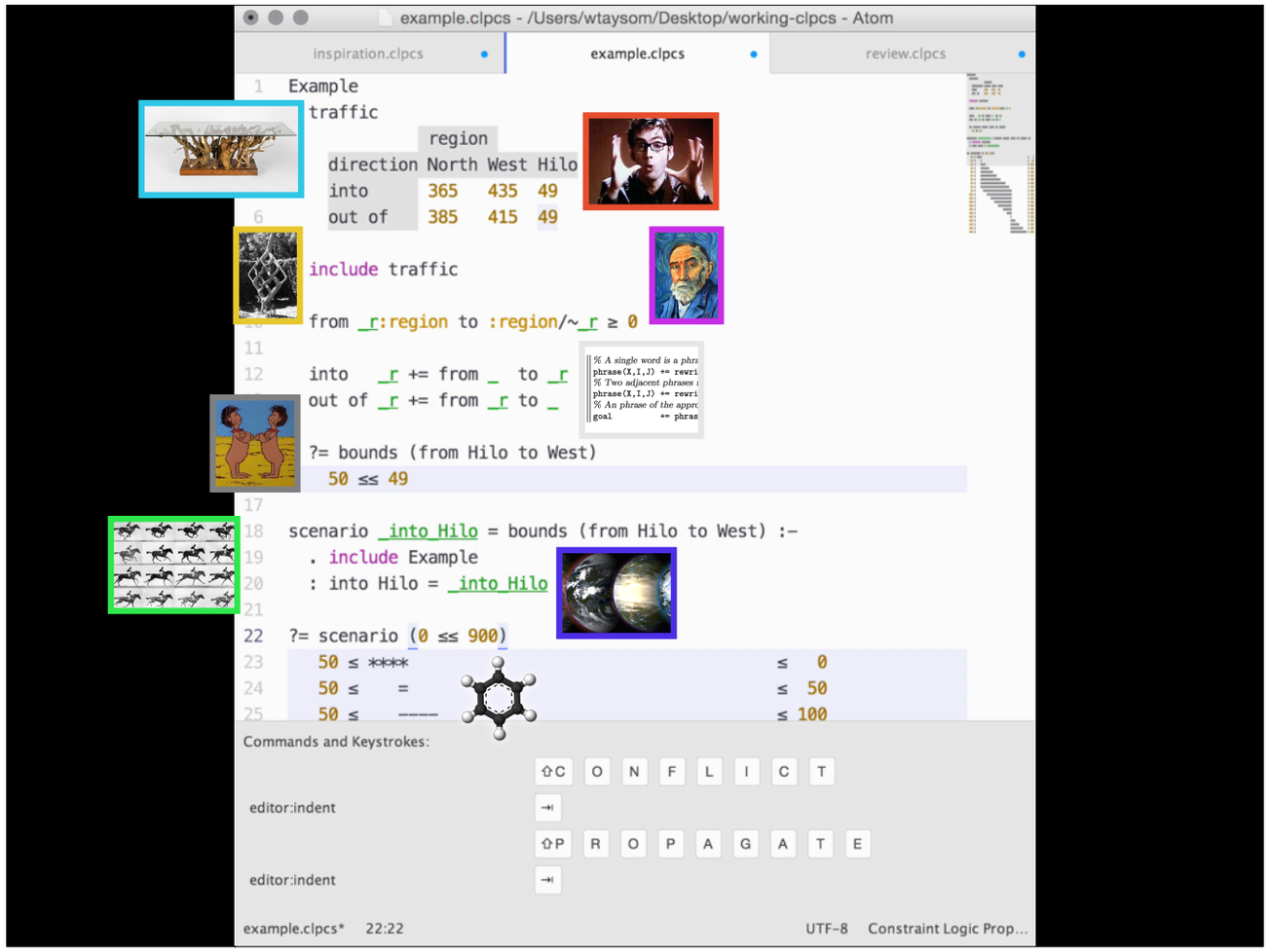
Binocular sense and reference.



Aggregates unify unfolds.



Propagate flexible shape.



Conflict resolves muddled to clear.

So I leave you with these nine criteria against which you can measure your own efforts and the promising ones we'll hear about today. I'm looking forward to it. Thank you.