PROFESSOR PATRICK WINSTON: Ladies and gentlemen, the engineers drinking song.

Back in the day, I've drunk quite a lot to that song.

And as drinking songs go, it's not bad.

I caution you, however, before playing this song in the presence of small children, audition it first.

Some of the verses are sufficiently gross as to make a sailor go beyond blushing.

It's an interesting song because there are an infinite number of verses.

Here's the mathematical proof.

Suppose there were a finite number of verses.

Then there would be a last verse.

And if there were a last verse, then some drunk alumni would compose a new one.

Therefore, there is no last verse, the size is not finite, and there are an infinite number of verses.

I play it for you today because I'm an engineer.

I like to build stuff.

I build stuff out of wood.

I build stuff out of metal.

I build stuff out of rocks.

And I especially like to write programs.

I don't know, sometimes people come to me and say, I'm majoring in computer science, but I don't like to write programs.

I've always been mystified by that.

I mean, if you want to show how tough you are, you can go bungee jumping or drive a nail through your hand or something like that instead.

But I've written quite a few programs for demonstrating stuff in this subject.

They're all written in Java, principally because I can therefore make them available to you and to the rest of the world by way of Web Start.

A few weeks ago, I was mucking around with the system and broke the version on the server, and within 15 minutes, I got an email from somebody in the depths of Anatolia complaining about it and asking me to bring it back up.

This particular program is patterned after an early AI classic.

And it was the business end of a program written by Terry Winograd, who became, and is, a professor of computer science at Stanford University-- which is on the west coast for those of you-- on the strength of his work on the natural language front end of this program.

But the natural language part is not what makes it of interest for us today.

It's more the other kinds of stuff.

Let's pile these things up.

Now, I'm going to ask to do something, maybe put B2 on top of B7.

Not bad.

How about B6 on B3?

This program's kind of clever.

Let me do one more.

Let's put B7 on B2.

OK, now let's see.

Maybe B5 on B2?

B4 on B3 first, maybe?

Oh, I must have clicked the wrong button.

Oh, there it goes.

OK.

Let's put B4 on B1.

Agh, my mouse keeps getting out of control.

Now, let's put B1 on B2.

This is an example I'm actually going to work out on the board.

Oh, I see.

My touch pad accidentally got activated.

B1 on B2.

Now, let's ask a question.

OK.

Well.

SPEAKER 2: [SINGING] PROFESSOR PATRICK WINSTON: Stop.

SPEAKER 3: [LAUGHTER] PROFESSOR PATRICK WINSTON: Had enough of that.

Let's see.

Why did you put-- why did you want to get rid of B4?

OK, one-- SPEAKER 2: [SINGING] PROFESSOR PATRICK WINSTON: Maybe they think, that's what happens when you use software you write yourself.

Why did you want to clear the top of B2?

Did I do that?

Why did you clear the top of B1?

OK.

SPEAKER 2: [SINGING] SPEAKER 3: [LAUGHTER] PROFESSOR PATRICK WINSTON: Oh, it's haunting me.

Yeah.

So the drinking song is easily offended, I guess.

But I won't develop that scenario again.

What I want to show you is that this program looks like it's kind of smart, and it somehow can answer questions about its own behavior.

Have you ever written a program that's answered questions about its own behavior?

Probably not.

Would you like to learn how to do that?

OK.

So by the end of the hour, you'll be able to write this program and many more like it that know how to answer questions about their own behavior.

There have been tens of thousands of such programs written, but only by people who know the stuff I'm going to tell you about right now, OK?

So what I want to do is I want to start by taking this program apart on the board and talking to you about the modules, the subroutines that it contains.

So here it is.

The first thing we need to think about, here are some blocks.

What has to happen if I'm going to put the bottom block on the larger block?

Well, first of all, I have to find space for it.

Then I have to grasp the lower block.

And I have to move it and I have to ungrasp it.

So those are four things I need to do in order to achieve what I want to do.

So therefore, I know that the put-on method has four pieces.

It has to find space on the target block.

It has to grasp the block that it's been commanded to move.

Then it has to move, and then it has to ungrasp.

But taking hints from some of the questions that it did answer before I got haunted by the music, taking our cue from that, we know that in order to grasp something, in this particular world, you can't have anything on top of it.

So grasp, therefore, may call clear top in order to get stuff off from the target object.

And that may happen in an iterative loop because there may be several things on top.

And how do you get rid of stuff?

Well, by calling get rid of.

And that may go around a loop several times.

And then, the way you get rid of stuff is by calling put-on.

So that gives us recursion, and it's from the recursion that we get a lot of the apparent complexity of the program's behavior when it solves a problem.

Now, in order to find space, you also have to call get rid of.

So that's where I meant to put this other iterative loop, not down here.

Cleat top has got the iterative loop inside of it.

So that's the structure of the program.

It's extremely simple.

And you might say to me, well, how can you get such apparently complex-looking behavior out of such a simple program?

A legitimate question.

But before we tackle that one head on, I'd like to do a simulation of this program with a very simple blocks problem.

And it's the one I almost showed you, but it goes like this.

Here's B1.

We'll call this BX because I forgot its name.

Here's BY.

And here's B2.

And the task is to put B1 on B2.

And according to our system diagram, that results in four calls to subroutines.

We have to find space.

We have to grasp B1.

We have to move, and then we ungrasp.

Now, the way we grasp something is the first thing we have to do is clear off its top.

So grasp calls clear top.

And clear top in turn calls get rid of.

And let me see.

Let me keep track of these.

This is clearing the top of B1, and this is getting rid of BX.

And the way we get rid of BX is by putting BX on the table.

And then that in turn causes calls to another find space, another grasp, another move, and another ungrasp.

So that's a little trace of the program as it works on this simple problem.

So how does it go about answering the questions that I demonstrated to you a moment ago?

Let's do that by using this trace.

So how, for example, does it answer the question, why did you get rid of BX?

[INAUDIBLE], what do you think?

How can it answer that question?

SPEAKER 4: [INAUDIBLE] PROFESSOR PATRICK WINSTON: So it goes up one level and reports what it sees.

So it says, and said in the demonstration, I got rid of BX because I was trying to clear the top of B1.

So if I were to say why did you clear the top of B1, it would say because I was trying to grasp it.

If I were to say, why did you grasp B1, it would say because I was putting B1 on B2.

If I say, why did you put B1 on B2, it would say, slavishly, because you told me to.

OK, so that's how it deals with why questions.

How about how questions?

Timothy, what do you think about that one?

How would it go about answering a question about how you did something?

Do you have a thought?

TIMOTHY: Um, yeah, it would think about what I was trying to accomplish.

PROFESSOR PATRICK WINSTON: Yeah, but how would it do that?

How would the program do that?

We know that answering a why question makes it go up one level.

How does it answer a how question?

Sebastian?

SEBASTIAN: It goes down one level.

PROFESSOR PATRICK WINSTON: You go down one level.

So you start off all the way up here with a put-on.

You will say, oh, well I did these four things.

You say, why did you grasp B1?

It will say because I was trying to clear its top.

Why did you clear its top?

Because I was trying to get rid of it.

Why were you trying to get rid of it?

Because I was trying to put it on the table.

So that's how it answers how questions, by going down in this tree and this trace of the program of action so as to see how things are put together.

What are these things that are being put together?

What's the word I've been avoiding so as to bring this to a crescendo?

What are these objectives, these things it wants to do?

They're goals.

So this thing is leaving a trace, which is a goal tree.

Does that sound familiar?

Three days ago, we talked about goal trees in connection with integration.

So this thing is building a goal tree, also known as an and-or tree.

So this must be an and tree.

And if this is an and tree, are there any and nodes?

Sure, there's one right there.

So do you think then that you can answer questions about your own behavior as long as you build an and-or tree?

Sure.

Does this mean that the integration program could answer questions about its own behavior?

Sure.

Because they both build goal trees, and wherever you got a goal tree, you can answer certain kinds of questions about your own behavior.

So let me see if in fact it really does build itself a goal tree as it solves problems.

So this time, we'll put B6 on B3 this time.

But watch it develop its goal tree.

So in contrast to the simple example I was working on the board, this gets to be a pretty complicated goal tree.

But I could still answers questions about behavior.

For example, I could say, why did you put B6 on B3?

Because you told me to.

All right, so the complexity of the behavior is largely a consequence not of the complexity of the program in this particular case, but the building of this giant goal tree as a consequence of the complexity of the problem.

This brings us to one of our previous matters-- early on to one of the gold star ideas of today.

And this gold star idea goes back to a lecture given in the late '60s by Herb Simon, who was the first Nobel Laureate in the pseudo Nobel Prize for economics.

Is that right, Bob?

Was he the first?

All right, he was the first winner of the Nobel Prize, pseudo Nobel Prize in economics.

And in this lecture, which was titled "The Sciences of the Artificial," he said imagine that you're looking on a beach at the path of a ant.

And he said, well, you know, the path of the ant looks extremely complicated.

And you're tempted to think the ant is some kind of genius or monster brain ant.

But in fact, when you take a closer look, what you discover is that there are a bunch of pebbles on the beach, and

all the ant is doing is avoiding those pebbles on his way home.

So the complexity of the behavior, said Simon, is a consequence of the complexity of the environment, not the complexity of the program.

So that's the metaphoric Simon's ant.

And what it says is that the complexity of the behavior is the max of the complexity of the program and the complexity of the environment.

So that's something we'll see many times during the rest of the semester.

Complex behavior, simple program.

You think it's going to be complicated.

It turns out to be simple because the problem has the complexity, not the program.

So that brings us to check box three in today's talk, and there's a little bit of a scene here because now I want to stop talking about goal-centered programming and start talking about rule-based expert systems.

The rule-based expert systems were developed in a burst of enthusiasm about the prospects for commercial applications of artificial intelligence in the mid-1980s.

At that time, it was supposed lengthy articles are written, but you could account for useful aspects of human intelligence by writing all the knowledge in the form of simple rules.

So if this is true, then that's true.

If you want to achieve this, then do that.

But all the knowledge had to be encapsulated in the form of simple rules.

So what might you want to do with this?

All sorts of things.

Thousands of these systems were written, as I indicated before.

But here's an example.

I'm going to work out an example having to do with identification.

And this example is patterned off of a classic program, strangely also written at Stanford, called MYCIN.

It was developed to diagnose bacterial infections in the blood.

So you come in.

You got some horrible disease, and the doctor gets curious about what antibiotic would be perfect for your disease.

He starts asking a lot of questions.

So I'm not going to deal with that because that world has all kinds of unpronounceable terms like bacterioides and anaerobic and stuff like that.

So it's completely analogous to talk about identifying animals in a little zoo, sort of a small town type of zoo.

So I'm going to suggest that we write down on a piece of paper all the things we can observe about an animal, and then we'll try to figure out what the animal is.

So I don't know, what can we start with?

Has hair.

Then there are some characteristics of the following form.

Has claws.

Sharp teeth.

And forward-pointing eyes.

And these are all characteristics of carnivores.

We happen to have forward-pointing eyes too, but that's more because we used to swing around the trees a lot, and we needed the stereo vision.

And we don't have the claws and the sharp teeth that go with it.

But anyhow, those are typically characteristics of carnivores, as is eating meat.

And this particular little animal we're looking at has also got spots, and it's very fast.

What is it?

Well, everybody says it's a cheetah.

Let's see how our program would figure that out.

Well, program might say, let's see if it has hair, then rule one says that that means it must be a mammal.

We can imagine another rule that says if you have sharp claws, sharp teeth, and forward-pointing eyes, then you're a carnivore.

And I'm using sort of hardware notation here.

That's an and gate, right?

So that means we have to have all of these characteristics before we will conclude that the animal is a carnivore.

Now, this animal has been also observed to eat meat.

So that means we've got extra evidence that the animal is carnivorous.

And now, because the animal is a mammal and a carnivore and has spots, and it's very fast, then the animal is a cheetah.

And I hope all of our African students agree that it must be a cheetah.

It's a small zoo-- I mean, a big zoo.

Who knows what it is?

It's probably got some unpronouncable name-- there's possibilities.

But for our small zoo, that will do.

So we have group now written down in the form of these and gates.

Several rules, R1, R2-- and there needs to be an and gate here-- that's R3 and an R4.

All of which indicate that this animal is a cheetah.

So we built ourself a little rule-based expert system that can recognize exactly one animal, but you could imagine filling out this system with other rules so that you could recognize giraffes and penguins and all the other sorts of

things you find in a small zoo.

So when you have a system like this that works as I've indicated, then what we're going to call that, we're going to give that a special name, and we're going to call that a forward-chaining rule-based-- because it uses rules-- expert system.

And we're going to put expert in parentheses because when these things were developed, for marketing reasons, they called them expert systems instead of novice systems.

But are they really experts in a human sense?

Not really, because they have these knee-jerk rules.

They're not equipped with anything you might want to call common sense.

They don't have an ability to deal with previous cases, like we do when we go to medical school.

So they really ought to be called rule-based novice systems because they reason like novices on the basis of rules.

But the tradition is to call them rule-based expert systems.

And this one works forward from the facts we give it to the conclusion off on the right.

That's why it's a forward-chaining system.

Can this system answer questions about its own behavior?

[INAUDIBLE], what do you think?

SPEAKER 5: [INAUDIBLE].

PROFESSOR PATRICK WINSTON: Why?

SPEAKER 5: [INAUDIBLE].

PROFESSOR PATRICK WINSTON: Because it looks like a goal tree.

Right.

This is, in fact, building a goal tree because each of these rules that require several things to be true is creating an and node.

And each of these situations here where you have multiple reasons for believing that the thing is a carnivore, that's creating an or node.

And we already know that you can answer questions about your own behavior if you leave behind a trace of a goal tree.

So look at this.

If I say to it, why were you interested in the animal's claws?

Because I was trying to see if it was a carnivore.

How did you know that the animal is a mammal?

Because it has hair.

Why did you think it was a cheetah?

Because it's a mammal, a carnivore, has spots, and very fast.

So by working forward and backward in this goal tree, this too can answer questions about its own behavior.

So now you know how, going forward, you can write programs that answer questions about their behavior.

Either you write the subroutines so that each one is wrapped around a goal, so you've got goal-centered programming, or you build a so-called expert system using rules, in which case it's easy to make it leave behind a trace of a goal tree, which makes it possible to answer questions about its own behavior, just as this [INAUDIBLE] program did.

But now, a little more vocabulary.

I'm going to save time by erasing all of these things that I previously drew by way of connections.

And I'm going to approach this zoo in a little different way.

I'm going to not ask any questions about the animal.

Instead, I'm going to say, mommy, is this thing I'm looking at a cheetah?

And how would mommy go about figuring it out.

In her head, she would say, well, I don't know.

If it's going to be a cheetah, then it must be the case that it's a carnivore, and it must be the case that it has spots.

And it must be the case that it's very fast.

So so far, what we've established is that if it's going to be a cheetah, it has to have the four characteristics that mommy finds behind this rule are four.

So instead of working forward from facts, what I'm going to do is work backward from a hypothesis.

So here the hypothesis is this thing is a cheetah.

How do I go about showing whether that's true or not?

Well, I haven't done anything so far because all I know is a cheetah if all these things are true, but are they true?

Well, to find out if it's a mammal, I can use rule one.

And if I know or can determine that the animal has hair, then that part of it is taken care of.

And I can similarly work my way back through carnivore.

I say, well, it's a carnivore if it has claws, sharp teeth, and forward-pointing eyes.

And then as much as the animal in question does, then I'm through.

I know it's a carnivore.

I don't have to go through and show that it's a carnivore another way.

So I never actually ask questions about whether it eats meat.

Finally, the final two conditions are met by just an inspection of the animal.

That's to say, it's in the database.

I don't have to use any rules to determine that the animal has spots and is very fast.

So now, I've got everything in place to say that it's a cheetah, because it's a carnivore, because it has claws, sharp teeth, and forward-pointing eyes, and all the rest of this stuff is similarly determined by going backwards, backwards from the hypothesis toward the facts, instead of from the facts forward to the conclusions.

So building a system that works like that, I have a backward-chaining rule-based expert system.

But there's a very important characteristic of this system in both backward and forward mode, and that is that this thing is a deduction system.

That's because it's working with facts to produce new facts.

When you have a deduction system, you can never take anything away.

But these rule-based systems are also used in another mode, where it's possible to take something away.

See, in fact world, in deduction world, you're talking about proving things.

And once you prove something is true, it can't be false.

If it is, you've got a contradiction in your system.

But if you think of this as a programming language, if you think of using rules as a programming language, then you can think of arranging it so these rules add or subtract from the database.

Let me show you an example of a couple of systems.

First of all, since I've talked about the MYCIN system, let me show you an example of a MYCIN dialogue.

That's a MYCIN dialogue.

And you can see the appearance of words you have to go to medical school to learn.

And here's a typical MYCIN rule, just like the rules for doing zoo analysis, only a more complicated domain.

But here's another example of a system that was written, not in the '80s, but just a couple of years ago by a student in the architecture department, Ph.D. thesis.

He was interested in the architecture of a Portuguese architect named Siza.

And Siza's done a lot of mass housing stuff.

And Siza has the idea that you ought to be able to design your own house.

And so Jose Duarte, a Portuguese student, a Ph.D.

student in architecture, wrote a rule-based system that was capable of designing Siza-like houses in response to

the requirements and recommendations and desires of the people who are going to occupy the houses.

So the most compelling part of this thing, of this exercise, was that Duarte took some of the designs of the program, mixed them up with some of the designs of Siza, and put them in front of Siza, and said, which ones did you do?

And Siza couldn't tell.

So somehow, the rule-based system that was built using this kind of technology was sufficient to confuse even the expert that they were patterned after.

But this program is a little complicated.

It, too, has its own specialized lingo.

So I'm not going to talk about it in detail, but rather talk instead about an analogous problem.

And that is a problem that everyone has faced at one point or another, and that is the problem of putting groceries in a bag at a grocery store.

It's the same thing, right?

Instead of putting rooms in a house, you're putting groceries in a bag.

And there must be some rules about how to do that.

In fact, maybe some of you have been professional grocery store baggers?

[INAUDIBLE] a grocery store professional bagger.

You're a-- which one?

LISA: [INAUDIBLE] PROFESSOR PATRICK WINSTON: Yeah, what is your name?

LISA: Lisa.

PROFESSOR PATRICK WINSTON: Lisa.

OK, well we got two professional grocery store baggers.

And I'm going to be now simulating a highly paid knowledge engineer desirous of building a program that knows how to bag groceries.

So I'm going to visit your site, Market Basket, and I'm going to ask Lisa, now fearful of losing her job, if she would tell me about how she bags groceries.

So could you suggest a rule?

LISA: Sure.

Large items in the bottom.

PROFESSOR PATRICK WINSTON: Large items in the bottom.

You see, that's why I'm a highly paid knowledge engineer, because I translate what she said into an if-then rule.

So if large, then bottom.

So now I-- SPEAKER 3: [LAUGHTER] PROFESSOR PATRICK WINSTON: So how about you, [INAUDIBLE]?

Have you got a suggestion?

About how to bag groceries?

SPEAKER 6: The small things on top.

PROFESSOR PATRICK WINSTON: If small, then on top.

Lisa, have you got anything else you could tell me?

LISA: Don't put too many heavy things in the same bag.

PROFESSOR PATRICK WINSTON: Don't put too many heavy things in the same bag.

So if heavy greater than three, then new bag, or something like that.

Is that all we're going to be able to-- does anybody else want to volunteer?

[INAUDIBLE], have you bagged groceries in Turkey?

LISA: So they don't have grocery baggers, so we have to-- PROFESSOR PATRICK WINSTON: So everybody's a professional bagger in Turkey.

Yeah.

It's outsourced to the customers.

SPEAKER 7: So no squishies on the bottom.

So if you have-- PROFESSOR PATRICK WINSTON: No squishies on the bottom.

SPEAKER 7: If you have tomatoes-- PROFESSOR PATRICK WINSTON: That's good.

Tomatoes.

SPEAKER 7: You don't want them to get squished.

PROFESSOR PATRICK WINSTON: Now, there's a very different thing about squishies and tomatoes because tomato is specific, and squishy isn't.

Now, one tendency of MIT students, of course, is that we all tend to generalize.

I once knew a professor in the Sloan School who seemed real smart.

And-- SPEAKER 3: [LAUGHTER] PROFESSOR PATRICK WINSTON: Then I figured out what he did.

If I were to say, I'm thinking about a red apple.

They'd sit back and say, oh, I see you're contemplating colored fruit today.

They're just taking it up one level of abstraction.

Not a genius.

He also was able to talk for an hour after he drew a triangle on the board.

Amazing people.

Anyhow, where were we?

Oh, yes, bagging groceries.

So we're making some progress, but not as much as I would like.

And so in order to really make progress on tasks like this, you have to exercise-- you know about some principles of knowledge engineering.

So principle number one, which I've listed over here as part of a gold star idea, is deal with specific cases.

So while you're at the site, if all you do is talk to the experts like Lisa and [INAUDIBLE], all you're going to get is vague generalities because they won't think of everything.

So what you do is you say, well, let me watch you on the line.

And then you'll see that they have to have some way of dealing with the milk.

And then you'll see that they have to have some way of dealing with the potato chips.

Nobody mentioned potato chips, except insofar as they might be squishy.

We don't have a definition for squishy.

Nobody talked about the macaroni.

And no one talked about the motor oil.

This is a convenience store.

I don't want that in the same bag with the meat.

And then no one talked about canned stuff.

Here's a can of olives.

So by looking at specific cases, you elicit from people knowledge they otherwise would not have thought to give you.

That's knowledge engineering rule number one.

And within a very few minutes, you'll have all three knowledge engineering rules and be prepared to be a highly paid knowledge engineer.

Heuristic, let's call these heuristics.

Heuristic number one, specific cases.

Heuristic number two is ask questions about things that appear to be the same, but are actually handled differently.

So there's some Birds Eye frozen peas, and here-- ugh, some fresh cut sweet peas.

And to me, the person who's never touched a grocery bag in my life-- maybe I'm from Mars-- I can't tell the difference.

They're both peas.

But I observe that the experts are handling these objects differently.

So I say, why did you handle those peas differently from those peas, and what do they say?

One's canned, and one's frozen.

So what happens?

Bingo, I've got some new words in my vocabulary.

And those new vocabulary words are going to give me power over the domain because I can now use those words in my rules.

And I can write rules like if frozen, then put them all together in a little plastic bag.

Actually, that's too complicated, but that's what we end up doing, right?

Why do we put them all together in a little plastic bag?

SPEAKER 8: [INAUDIBLE] PROFESSOR PATRICK WINSTON: What's that?

SPEAKER 8: [INAUDIBLE] PROFESSOR PATRICK WINSTON: Well, there are two explanations.

There's the MIT explanation.

We know that temperature flow is equal to the fourth power of the temperature difference and the surface area and all that kind of stuff.

We want to get them all together in a ball, sphere.

The normal explanation is that they're going to melt anyway, so they might as well not get everything else wet.

All right.

SPEAKER 3: [LAUGHTER] PROFESSOR PATRICK WINSTON: So that's heuristic number two.

And actually, there's heuristic number three, that I just want to relate to you for the first time because I have been

dealing with it a lot over this past summer.

Heuristic number three is you build a system, and you see when it cracks.

And when it cracks is when you don't have one of the rules you need in order to execute-- in order to get the program to execute as you want it to execute.

So if I were to write a grocery store bagging program and have it bag some groceries, again, eventually it would either make a mistake or come to a grinding halt, and bingo, I know that there's a missing rule.

Isn't that what happens when you do a problem set, and you hit an impasse?

You're performing an experiment on yourself, and you're discovering that you don't have the whole program.

In fact, I've listed this as a gold star idea having to do with engineering yourself because all of these things that you can do for knowledge engineering are things you can do when you learn a new subject yourself.

Because essentially, you're making yourself into an expert system when you're learning circuit theory or electromagnetism or something of that sort.

You're saying to yourself, well, let's look at some specific cases.

Well, what are the vocabulary items here that tell me why this problem is different from that problem?

Oh, this is a cylinder instead of a sphere.

Or you're working with a problem set, and you discover you can't work with the problem, and you need to get another chunk of knowledge that makes it possible for you to do it.

So this sort of thing, which you think of primarily as a mechanism, heuristics for doing knowledge engineering, are also mechanisms for making yourself smarter.

So that concludes what I want to talk with you about today.

But the bottom line is, that if you build a rule-based expert system, it can answer questions about its own behavior.

If you build a program that's centered on goals, it can answer questions about its own behavior.

If you build an integration program, it can answer questions about its own behavior.

And if you want to build one of these systems, and you need to extract knowledge from an expert, you need to

approach it with these kinds of heuristics because the expert won't think what to tell you unless you elicit that information by specific cases, by asking questions about differences, and by ultimately doing some experiments to see where your program is correct.

So that really concludes what I had to say, except I want to ask the question, is this all we need to know about human intelligence?

Can these things be-- are these things really smart?

And the traditional answer is no, they're not really smart because their intelligence is this sort of thin veneer.

And when you try to get underneath it, as written, they tend to crack.

For example, we talk about a rule, we could talk about a rule that knows that you should put the potato chips on the top of the bag.

But a program that knows that would have no idea why you would want to put the potato chips on top of the bag.

They wouldn't know that if you put them on the bottom of the bag, they'll get crushed.

And it wouldn't know that if they get crushed, the customer will get angry, because people don't like to eat crushed potato chips.

So that's what I mean when I say the knowledge of these things tends to be a veneer.

So the MYCIN program, during debugging, once prescribed a barrel of penicillin to be administered to a patient for its disease.

They don't know, they don't have any common sense.

So the question then becomes, well, I don't know.

Does rule-based-- do rules have anything to do with common sense?

And I'm becoming a little bit agnostic on that subject.

Because there are certain indications, there are certain situations, in which rules could be said to play a role in our ordinary understanding of things.

Would you like to see a demonstration?

What I'm going to show you, when the clip speaks up-- well, before I make any promises, let me see if I'm actually connected to the web.

MIT, good.

MIT.

Guest.

Yeah, that's me.

Sounds good.

OK, I just tested the system, and I've seen it is actually connected to the web.

And I'm going to adjust some systems options here.

I'll get rid of the text box.

And we'll get rid of those changes scale a little bit.

What I'm going to do is I'm going to read a little synopsis of the Macbeth plot.

You're MIT students.

I'm sure you're all classically educated and very familiar with Shakespearean plots.

So I'm going to read one.

I'm going to read a version of a Macbeth plot.

And it's going to go along like this.

It's basically reading a rule base so far.

And pretty soon, it's going to get beyond the rule base and start reading the Macbeth story.

And there it is.

It's read the Macbeth story.

Let me show you what the Macbeth story looks like as it's actually retained by the system.

That's it.

Read that.

OK, you ran out of time because the machine's already finished.

It takes about five seconds to read this story.

Now, as you look at this little synopsis of Macbeth, there are a couple things to note.

For one thing, it says that Duncan is murdered.

Duncan, I hope this doesn't bother you.

Duncan is murdered by Macbeth.

But at no time does it say that Duncan is dead.

But you know Duncan's dead because he was murdered.

If murdered, then dead.

SPEAKER 3: [LAUGHTER].

PROFESSOR PATRICK WINSTON: So if you look a little further down, what you see is that Macduff kills Macbeth.

Fourth line up from the bottom.

Why did Macduff kill Macbeth?

Doesn't say why in this story, but you have no trouble figuring out that it's because he got angry.

And when you get angry, you don't necessarily kill somebody, but it's possible.

SPEAKER 3: [LAUGHTER].

PROFESSOR PATRICK WINSTON: So now that you see what's in the story, let me take you back to this display.

It's what we call an elaboration graph.

And when I blow it up, you can see that there's some familiar looking things in there.

For example, up here in the left-hand corner, Macbeth murders Duncan, right over there.

And over here, Macduff kills Macbeth.

And if you look at what is a consequence of that, it looks like there must be a rule that says if you murder somebody, you harm them.

And if you murder somebody, then they're dead.

And one reason why you might kill somebody is because they angered you.

And if you go the other way, one consequence of killing somebody is that you harm them, and that they die too.

And if you harm somebody, they get angry, and their state goes negative.

So that suggests that there are some things that we have on our hands that are very compiled and very, strangely enough, very rule-like in their character.

Now, to close, I'm just going to read Hamlet.

The Hamlet demonstration is much like the Macbeth one.

In fact, Hamlet and Macbeth are very alike in their plot.

But there's one thing that's well-illustrated by our particular capturing of Hamlet here.

And that is that you'll note that the ratio of gray stuff to white stuff is considerable.

The gray stuff is stuff that has been deduced by rules.

And the reason there's so much gray stuff in this Hamlet story is because everybody's related to everybody else.

So when you kill anybody, you irritate everybody else.

So look at that.

A few white things, those are the things that are explicit in the story, and lots of gray stuff.

So what this is suggesting is that when we tell a story, it's mostly a matter of controlled hallucination.

I know what rules are in your head, so I could take advantage of that in telling the story and not have to tell you anything I'm sure you're going to know.

And so that's why, we've discovered, that storytelling is largely a matter of just controlling how you're going along,

a kind of controlled hallucination.