Thinking as Computation

Hector J. Levesque
Dept. of Computer Science
University of Toronto

1

Thinking and Computation

Intelligent behaviour

In this course, we will consider what it takes to get a computer to engage in intelligent activities such as

- understanding English sentences;
- recognizing objects in a visual scene;
- planning courses of actions;
- solving recreational puzzles;
- playing strategic games.

What these very different activities all have in common, is that when they are performed by people, they appear to require *thought*.

Are computers electronic brains?

Historically, models of the brain have often been associated with the most advanced technology of the time.

- clockwork
- steam engine
- telephone switchboard
- and now it's . . . *computers*!

Invariably, we end up *laughing* at these simplistic models!

- found to be simplistic, misleading
- tell us very little about what we are, and how we do what we do

Why should we think computers are any different??

The brain

In this course, as in much of AI, we will be concerned with *thinking* (of various sorts), but have very little to say about the *brain*.

Here is a useful analogy: the study of *flight* (before airplanes)

- we might like to understand how animals like birds can fly
- we might want to build machines that are capable of flight

Two possible strategies:

1. study birds, their feathers, muscles, very carefully and construct machines to emulate them
2. study *aerodynamics*: principles of flight applicable to anything

We will be following the second strategy here (for thinking, not flight).

Thinking

Q: What is thinking?

A: Thinking is a process that occurs in the brain over time,
but largely unconsciously.

Let’s look at thinking in action. . .

Read this sentence:

> *The trophy would not fit into the brown suitcase because it was too small.*

Now answer this question:

> What was too small? What is the “*it*”?

How did you figure this out?

Using what you know

Observe: There is nothing in the sentence itself that gives away the answer!

To see this, replace “*small*” by “*big*”:

> *The trophy would not fit into the brown suitcase because it was too big.*

Now what is the “*it*”?

So you have to use a lot of what you knew about the sizes of things, things fitting inside of other things etc. even if you are unaware of it.

This is thinking!

Roughly: Bringing what you *know* to bear on what you are doing.

What sort of process is thinking?

It is a biological process that happens in the brain.

> Like digestion in the stomach?
>
> Like mitosis in cells?

Key Conjecture: thinking can be usefully understood as a *computational* process

> Perhaps thinking has more in common with *multiplication* or *sorting a list of names*, than with *digestion* or *mitosis*.

This is extremely controversial!

We will spend most of the course exploring this one idea.

But first we need to understand what “computational process” means . . .

Computer Science

It's not about *computers*!

Having problems with your PC? Don't ask a Computer Scientist!

> The electronic / physical properties of computers play little or no role in Computer Science
>
> Another analogy to think about: musical instruments *vs.* music
>
> Like music, Computer Science is not about anything physical!

Computer Science is about *computation*:

> a certain form of manipulation of *symbols*

Modern electronic computers (like an *Apple iMac* or a *Dell PC*) just happen to provide a fast, cheap, and reliable medium for computation.

Symbols and symbolic structures

- Simplest sort: characters from an alphabet

  digits: 3, 7, V

  letters: a, R, α

  operators: +, $\times$, $\cap$

- String together into more complex ones

  numerals: 335.42

  words: “don’t”

- More complex groupings

  expressions: $247 + 4(x - 1)^3$

  phrases: “the woman John loved”

- Into truth-valued symbolic structures

  relations: $247 + 4(x - 1)^3 > \frac{z!}{5}$

  sentences: “The woman John loved had brown hair.”

Manipulating symbols

The idea: take strings of symbols, break them apart, compare them, and reassemble them according to a recipe called a *procedure.*

It is important to keep track of where you are, and follow the instructions in the procedure exactly. (You may not be able to figure *why* you are doing the steps involved!)

The symbols you have at the start are called the *inputs*. Some of the symbols you end up with will be designated as the *outputs*.

> We say that the procedure is *called* on the inputs
> and *returns* or produces the outputs.

Then construct more complex procedures out of simple procedures.

In the next few slides, we will look at an interesting special case: **arithmetic**!

Arithmetic procedures

Imagine you are explaining to someone (a young child) how to do subtraction:

$$\begin{array}{r} 5\,2 \\ -\,1\,7 \\ \hline \end{array}$$

You might use words like this:

> *First, you have to subtract* 7 *from* 2*. But since* 7 *is bigger than* 2*, you need to borrow* 10 *from the* 5 *on the left.*
>
> *That changes the* 2 *to a* 12 *and changes the* 5 *to a* 4*.*
>
> *So you subtract* 7 *not from* 2 *but from* 12*, which gives you* 5*, and you subtract* 1 *not from* 5 *but from* 4 *which gives you* 3*.*
>
> *So the answer is* 35*.*

A very first procedure

Let’s go back to the most basic form of arithmetic we can imagine: adding two single digit numbers.

What is the procedure there?

Here’s one version: call it **PROC0**

- You will be given two digits as input and return two digits as output. (For example, given 7 and 6 as input, you will return 13 as output.)
- To do so, you will use a table which is on the next page.
- To add the two digits, find the *row* for the first digit on the table, and the *column* for the second digit, and return as output the two digit number at the intersection on the table.

A table for addition

7 + 6 = 13

	0	1	2	3	4	5	↓ 6	7	8	9
0	00	01	02	03	04	05	06	07	08	09
1	01	02	03	04	05	06	07	08	09	10
2	02	03	04	05	06	07	08	09	10	11
3	03	04	05	06	07	08	09	10	11	12
4	04	05	06	07	08	09	10	11	12	13
5	05	06	07	08	09	10	11	12	13	14
6	06	07	08	09	10	11	12	13	14	15
→ 7	07	08	09	10	11	12	**13**	14	15	16
8	08	09	10	11	12	13	14	15	16	17
9	09	10	11	12	13	14	15	16	17	18

This is how we all learned multiplication!

Procedure PROC1

We can now start building on procedure PROC0 to do more.

The first thing we will need is to be able to add *three* digits (which will later allow us to handle carry digits).

Here is a procedure PROC1 that takes three digits a, t, and b, as input, and returns two digits c and s.

1. Call PROC0 on t and b to get u and v.
2. Call PROC0 on a and v to get u' and v'.
3. If $u = u' = 0$ then return with c as 0 and s as v'.
   If $u = u' = 1$ then return with c as 2 and s as v'.
   Otherwise, return with c as 1 and s as v'.

Note that we do not say *why* we do any of this! (But do try it out!)

Addition of two numbers

Now that we can follow the PROC1 procedure exactly, we can build on it to do something that we may start to recognize as real addition!

We will construct another procedure PROC2 that can add two strings of digits:

> **input of PROC2**: two strings of digits of the same length
>
> $$x_1\, x_2\, \ldots\, x_k$$
> $$y_1\, y_2\, \ldots\, y_k$$
>
> **output of PROC2**: one string of digits of that length + 1
>
> $$z_0\, z_1\, z_2\, \ldots\, z_k$$

For example, given 747 and 281 (with three digits each), PROC2 will return 1028 (with four digits).

The procedure PROC2

1. Start at the right hand side of the inputs.
   Call PROC1 with a as the digit 0, t as x_k, and b as y_k.
   Let z_k be the s returned. Hang on to the c for the next step.

2. Now move over one step to the left.
   Call PROC1 with a as the c returned in the previous step,
   t as x_{k-1}, and b as y_{k-1}.
   Let z_{k-1} be the s returned. Hang on to the c for the next step.

3. Continue this way through all the digits, from right to left,
   filling out $z_{k-2}, z_{k-3}, \ldots, z_3, z_2, z_1$.

4. Let z_0 be the final c returned (from PROC1 with x_1 and y_1).

Note again that the procedure does not explain what it all means.

Trying PROC2

We can trace what happens with PROC2 in detail.

Let's look at what PROC2 does when the x_i are 747 and the y_i are 281.

1. First, we call PROC1 on 0, 7, 1. It will return 0 and 8.
   So z_3 will be 8.
2. Next, we call PROC1 on 0, 4, 8. It will return 1 and 2.
   So z_2 will be 2.
3. Then, we call PROC1 on 1, 7, 2. It will return 1 and 0.
   So z_1 will be 0 and z_0 will be 1.

So PROC2 will return 1028, which is indeed the sum of 747 and 281.

So even if you don't *why* you are doing all the steps, if you follow the directions *exactly*, you will be adding the numbers.

Adding a list of numbers

We can now consider a more general procedure PROC3 that uses PROC2 to add any list of numbers $n_1, n_2, \ldots,$ each with any number of digits.

1. Let *sum* be the single digit 0.
2. Start with the first number n_1 and *sum*. Make sure both of these have the same number of digits by using a PROC4 (it will insert 0's as necessary to the left). Then call PROC2 on the two resulting numbers, and let *sum* now be the number returned.
3. Next, do the same thing with the new *sum* and n_2, to produce the next value of *sum*.
4. Continue this way with the rest of numbers, $n_3, n_4, \ldots$.
5. Return as output the final value of *sum*.

And on it goes ...

Let's assume we already have procedures to do $+, \times, -, \div$ and $\leq$, as well as string operations (e.g. $u\hat{\ }v$ means concatenate strings u and v).

We can use these to define still more complex procedures.

On the next slide, we will consider a mystery procedure called PROCX that takes one positive integer x as input and returns a positive integer y as output.

Try to figure out what this procedure does by trying it out on some sample inputs, for example, 137641.

You should be able to follow the procedure without knowing what exactly you are supposed to be calculating.

The PROCX procedure

Group the digits in x into pairs starting from the right.
Start with u, v, *bot*, *top*, and *side* all having value 0.

Then, working your way from left to right on the groups in x, repeat the following:

1. Set *bot* to $($*bot* $- u)$ ˆ(the next group from x)
2. Set *side* to $2 \times$ *top*
3. Set v to the largest single digit such that $v \times ($*side*ˆ$v) \leq$ *bot*
4. Set u to $v \times ($*side*ˆ$v)$
5. Set *top* to *top*ˆv

The answer y to return is the final value of *top*.

Lesson: arithmetic as symbol manipulation

Arithmetic can be built from more primitive symbol manipulations

- starting with very simple operations (such as table lookup), the ability to string and unstring symbols, compare them *etc.*, we can build procedures to do addition
- using these we can build ever more complex procedures, to do multiplication, division, testing for prime numbers, etc.
- using pairs of numbers we can deal with fractions (rational numbers); and using numbers arranged into matrices, we can solve systems of equations, perform numerical simulations, etc.

As we will see in this course, there are also many interesting cases of symbol manipulation that are not numeric!

A key observation

You don't need to know what you're doing!

> To get meaningful answers, you do not have to understand what the symbols stand for, or why the manipulations are *correct*.

The symbols can be manipulated completely mechanically and still end up producing significant and interesting results.

This is the trick of computation:

> We can get computers to perform a wide variety of very impressive activities precisely because we are able to describe those activities as a type of symbol manipulation that can be carried out purely mechanically.

This "trick" has turned out to be one of the major inventions of the 20th century. And note: It has nothing to do with electronics!

But what does this have to do with thinking?

Let us return to what we called the Key Conjecture from earlier:

> **Key Conjecture:** thinking can be usefully understood as a *computational* process

What does this controversial conjecture amount to?

- **not** that the brain is something like an electronic computer (which it is in some ways, but in very many other ways is not)
- but that the process of *thinking* can be usefully understood as a form of *symbol processing* that can be carried out purely *mechanically* without having to know what you are doing.

This is what we will explore in this course!

Thinking as computation

Some thinking clearly appears to be computation:

- doing math homework
- filling out a tax form
- estimating a grocery bill

But much of our thinking seems to have very little to do with calculations or anything numerical.

Unlike arithmetic, thinking seems to involve *ideas* about an unbounded variety of subjects.

> You can think about anything; but it is always *about* something.

So in what sense can thinking be computational?

An example

Consider the following:

> I know that my keys are either in my coat pocket or on the fridge.
> That's where I always leave them.
>
> I feel my coat pocket and I see that there is nothing there.
>
> So my keys must be on the fridge.
> And that's where I should go looking.

This is thinking that obviously has nothing to do with numbers.

> But it is clearly *about* something: my keys, pocket, refrigerator.

Can we understand it as a form of computation?

Gottfried Leibniz (1646-1716)

Co-inventor of the calculus (with Newton)

The first person to seriously consider the idea that ordinary thinking was computation

- the rules of *arithmetic* allow us to deal with *abstract numbers* in terms of concrete symbols

  manipulations on numeric symbols mirror relationships among the numbers being represented

- the rules of *logic* allow us to deal with *abstract ideas* in terms of concrete symbols

  manipulations on propositional symbols mirror relationships among ideas being represented

Propositions vs. sentences

Definition: A proposition is the *idea* expressed by a declarative sentence.

for example, the idea that

- my keys are in my coat pocket
- dinosaurs were warm-blooded
- somebody will fail this course

They are abstract entities (like numbers) but have special properties:

- They are considered to *hold* or *not hold*. A sentence is considered to be *true* if the proposition it expresses holds.
- They are *related to people* in various ways: people can believe them, disbelieve them, fear them, worry about them, regret them, *etc.*
- They are *related to each other* in various ways: entail, provide evidence, contradict, *etc.*

A first clue

We do not necessarily need to know what the terms in a sentence *mean* to be able to think about it.

Example: The snark was a boojum.

And now answer the following:

- What kind of thing was the snark?
- Is there anything that was a boojum?
- Was the snark either a beejum or a boojum?
- Given that no boojum is every a beejum, was the snark a beejum?

We can still figure out appropriate answers using simple rules.

Compare: the PROCX procedure!

Some related thoughts

My keys are in my coat pocket or on the fridge.

Nothing is in my coat pocket.

So my keys are on the fridge.

Compare to this:

Henry is in the basement or in the garden.

Nobody is in the basement.

So Henry is in the garden.

And compare to this:

Jill is married to George or Jack.

Nobody is married to George.

So Jill is married to Jack.

It’s all in the form

All three examples are really the same:

Blue thing is green thing or yellow thing.

Nothing is green thing.

So blue thing is yellow thing.

It does not matter whether

- blue thing is “my keys” or “Henry” or “Jill”
- green thing is “in my coat pocket or “in the basement” or “married to George”

Note: The thinking is the same!

The only thing that matters is that it is the same term (the blue thing) that is used in the first and third sentences, etc.

Entailment

A collection of sentences, S_1, S_2, $\ldots S_n$ *entail* a sentence S if the truth of S is implicit in the truth of the S_i.

> No matter what the terms in the S_i really mean, if the S_i sentences are all *true*, then S must also be *true*.

For example,

	The snark was a boojum.
Entails:	Something was a boojum.
	My keys are in my coat pocket or on the fridge.
and	Nothing is in my coat pocket.
Entails:	My keys are on the fridge.

Can this be how we think??

Suppose we are told at a party: George is a bachelor.

Here is what is entailed:

- Somebody is a bachelor.
- George is either a bachelor or a farmer.
- Not everyone is not a bachelor.
- It is not the case that George is not a bachelor. . . .

All true, but very very *boring*!

But when *we* think about it, we think about

- George (who we may know a lot about)
- being a bachelor (which we may know a lot about)

So *our* thinking appears to depend on what the terms mean!

Using what you know

We must consider the entailments of not only George is a bachelor,
but this + a large collection of other sentences we might already know:

George was born in Boston, collects stamps.

A son of someone is a child who is male.

George is the only son of Mary and Fred.

A man is an adult male person.

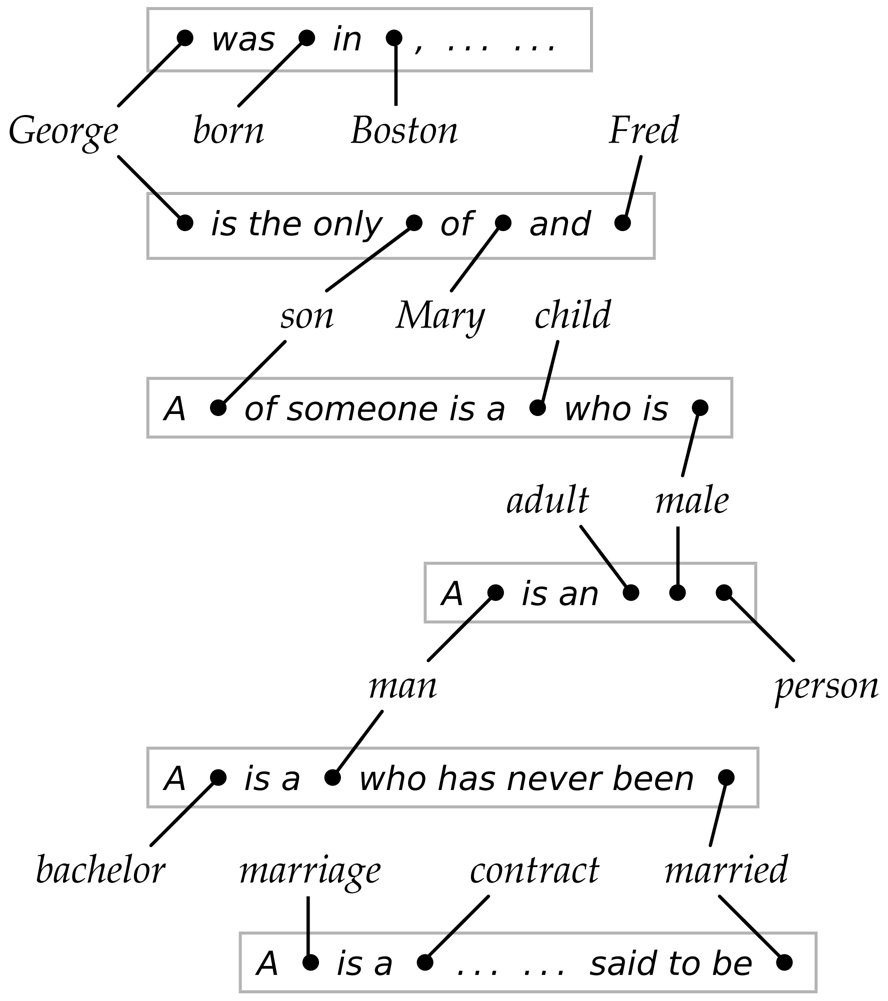
A bachelor is a man who has never been married.

A (traditional) marriage is a contract between a man and a woman, enacted by a wedding and dissolved by a divorce. While the contract is in effect, the man (called the husband) and the woman (called the wife) are said to be married.

A wedding is a ceremony where . . . bride . . . groom . . . bouquet . . .

The terms like "George" and "bachelor" and "person" and "stamps" appear in many places and link these sentences together.

The web of belief



Each sentence we believe is linked to many others by virtue of the terms that appear in them.

It is the job of logic to crawl over this web looking for connections among the terms.

Using what you know

When we include all these other facts, here are some entailments we get:

- George has never been the groom at a wedding.
- Mary has an unmarried son born in Boston.
- No woman is the wife of any of Fred's children.

The conclusions are much more like those made by ordinary people.

We find where the same term appears (like we did with the blue thing, the green thing etc.).

We then apply simple rules of logic to draw conclusions.

We still do not need to know what "George" or "bachelor" means!

Now, the big step:

Imagine drawing conclusions from *millions* of such facts.

Two hypotheses

1. Much of the richness of meaning that *we* experience during thinking may be accounted for in terms of simple symbolic manipulations over a *rich* collection of represented propositions

2. To build computers systems that are versatile, flexible, modifiable, explainable, . . . it is a good idea to

   - represent much of what the system needs to know as symbolic sentences
   - perform manipulations over these sentences using the rules of symbolic logic to derive new conclusions
   - have the system act based on the conclusions it is able to derive

   Systems built this way are called *knowledge-based systems* and the collection of sentences is called its *knowledge base* (*KB*).

Summary

Thinking, as far as we are concerned, means bringing what you know to bear on what you are doing.

But how does this work? How do concrete, physical entities like *people* engage with something formless and abstract like *knowledge*?

The answer (via Leibniz) is that we engage with *symbolic representations* of that knowledge.

> We represent knowledge symbolically as a collection of sentences in a knowledge base, and then compute entailments of those sentences, as we need them.

So *computation over a knowledge base* is the direction that we will be pursuing in this course, although we will only ever deal with tiny knowledge bases here.

2

A Procedure for Thinking

If-then reasoning

A special but very common case of a knowledge base is one consisting of just two sorts of sentences:

1. *atomic sentences*: simple basic sentences
2. *conditional sentences*: of the form

   If P_1 and . . . and P_n then Q

   where the P_i and the Q are atomic sentences.

The symbols If, and, and then are special *keywords*.

The other symbols in the sentences are either *variables* (written here capitalized) or *constants* (written here uncapitalized).

So john and dog23 are constants.

But X and Dog are variables.

A family example

Some atomic sentences:

john is a child of sue.　　john is a child of sam.

jane is a child of sue.　　jane is a child of sam.

sue is a child of george.　　sue is a child of gina.

john is male.　　jane is female.　　june is female.

sam is male.　　sue is female.　　george is male.

Some conditional sentences:

If X is a child of Y then Y is a parent of X.

If X is a child of Y and Y is male then Y is a father of X.

If X is male and Y is female then X is of opposite sex from Y.

If X is male and Y is female then Y is of opposite sex from X.

If X is a father of Y and Y is a parent of Z then X is a grandfather of Z.

Logical entailment revisited

The procedure we will consider next is used to find out if a given atomic sentence Q is *logically entailed* by the KB.

Recall: This means that if everything in the KB is true, then Q must be true no matter what the constants (like child, male, sam) actually mean.

So we cannot use our normal English understanding of the constants.

Imagine that child is written everywhere as relation257 *etc.*

Here are some entailments of the family example KB:

- jane is a child of sue

  This appears in the KB as is.

- sue is a parent of jane

  This is because the KB contains the above and

  If X is a child of Y then Y is a parent of X.

Some non-entailments

Here are some examples of atomic sentences that are *not* logically entailed by the KB:

- gina is female

  Although this may be true, there is nothing in the KB that forces it.

- sue is a mother of john

  Although the KB contains these:

  john is a child of sue
  sue is female

  we are not allowed to use our English understanding of mother.

  The symbol mother is not connected in the KB to child and female (the way father is connected to child and male).

Computing with a KB

Q: What sort of procedure do we use with a KB like this?

A: The simplest is what is called *back-chaining*.

It is used to *establish* an atomic sentence Q (called a *query*)

= determine whether or not the query Q is entailed by the KB.

For example, we will want to be able to establish the following:

jane is a child of sue.
sue is a parent of jane.

since these *are entailed* by the KB, but not

gina is female.
sue is the mother of john.

since these *are not entailed* (even though they may very well be true).

The back-chaining procedure

The back-chaining procedure returns either *success* or *failure*:

To establish a sentence Q:

1. Try to locate Q itself in the KB. If so, return success.
2. Otherwise, try to locate a conditional sentence of the form "If P_1 and ... and P_n then Q" in the KB. If not, return failure.
3. Otherwise, use back-chaining to try to establish P_1, then P_2, then ... P_n. If these are *all* successful, then return success.
4. Otherwise, go back to step (2) and look for another conditional.

Note: the back-chaining procedure for Q depends on using the back-chaining procedure for other sentences (the P_i).

This makes it a *recursive* procedure. (We will return to this later.)

Variables

Before we look at back-chaining in action, we need to deal with one complication: variables.

Suppose the query Q is “george is a parent of sue” and
we have a sentence in the KB like “If . . . then Y is a parent of X”.

We will consider this to *match* at Step 2, but at Step 3 we need to remember that the Y we care about is “george” and the X is “sue”.

we want:	george is a parent of sue (the Q)
we find:	If X is a child of Y then Y is a parent of X
	matches for Y=george and X=sue
then we want:	sue is a child of george (the P)

Tracing

We now simulate the execution of the back-chaining procedure on some simple examples. This is called *tracing* the procedure.

This is like what we did for arithmetic with the PROC2 example.

Example 1: Establish “jane is a child of sue”

1. Look for “jane is a child of sue” in KB. Find it.
   Return success.

Example 2: Establish “gina is female”

1. Look for “gina is female” in KB. Nothing found.
2. Look for “If . . . then gina is female” in KB. Nothing found.
   Return failure.

Another example

Example 3: Establish "george is a father of sue"

1. Look for "george is a father of sue" in KB. Nothing found.
2. Look for "If . . . then george is a father of sue" in KB.
   Find If X is a child of Y and Y is male then Y is a father of X.
   This matches for Y=george and X=sue.
3. - Establish "sue is a child of george" (Why?)
     1. Look for "sue is a child of george" in KB. Find it.
        Return success.
   - Establish "george is male" (Why?)
     1. Look for "george is male" in KB. Find it.
        Return success.

   Return success overall.

Yet another example

Example 4: Establish "jane is of opposite sex from george"

1. Look for "jane is of opposite sex from george" in KB. Nothing.

2. Look for "If . . . then jane is of opposite sex from george" in KB.
   Find If X is male and Y is female then X is of opposite sex from Y.

3. • Establish "jane is male". << *Stuff left out* >>
   Return failure. (No need to try "george is female".)

4. Go back to Step 2 and look for another conditional.
   Find If X is male and Y is female then Y is of opposite sex from X.

3. • Establish "george is male". << *Stuff left out* >> Return success.
   • Establish "jane is female". << *Stuff left out* >> Return success.

   Return success overall.

Variables in the query

Sometimes the query Q may contain one or more variables.

In this case, we are not interested in whether the Q is entailed,
but for what *values of the variables* Q is entailed.

> e.g. with the query “W is a child of gina” we ask for a child of Gina.
>
> In this case, instead of simply saying “Return success”, we would say something like “Return success with W=sue”

Queries with variables may have more than one answer.

> e.g. the query “sue is of opposite sex from Z” will have three answers: sam, john, and george.

Queries without variables can be thought of as *yes-no-questions*.
Queries with variables can be thought of as *WH-questions*.

Two complications

This leads to two new complications in the back-chaining procedure:

1. **The question:** What if the same variable appears in both the KB and in the Q? How to keep straight which is which?

   The answer: Rename the variables from the sentence in the KB before using it so that this does not happen.

   Note: If U is a child of V then V is a parent of U says exactly the same thing as If X is a child of Y then Y is a parent of X.

2. **The question:** What if I choose a value for a variable, but then cannot establish the query that results?

   The answer: You try the first value you find, but if it does not work, you will need to go back and reconsider other possible values.

   Note: This is just like the choice of conditional sentence in Step 2.

A (long) example with a variable in a query

Example 5: Establish “george is a grandfather of john”

1. Look for “george is a grandfather of john” in KB. Nothing found.
2. Look for “If . . . then george is a grandfather of john” in KB.
   Find If X is a father of Y and Y is a parent of Z then X is a grandfather of Z.
3. - Establish “george is a father of Y”. (Note the variable in the query!)
     1. Look for “george is a father of Y” in KB. Nothing found.
     2. Look for “If . . . then george is a father of Y” in KB.
        Find If U is a child of V and V is male then V is a father of U. (Note!)
     3. – Establish “Y is a child of george.” (Note that U=Y)
          1. Look for “Y is a child of george” in the KB. Find it for Y=sue.
             Return success with Y=sue. (only choice)

        – Establish “george is male” . . . Eventually return success.

        Return success with Y=sue.
   - Establish “sue is a parent of john”. (Note Y=sue) . . . Return success.

   Finally return overall success!

A review of the thinking

Wanted to establish: George is a grandfather of John.

- Want: George is a father of some Y
  - Want: Y is a child of George
    This holds for Y=sue.
  - Want: George is male
    This holds.

  So we get: George is a father of Y (where Y=sue)

- Want: Y is a parent of John (where Y=sue)
  - Want: John is a child of Y (where Y=sue)
    This holds.

  So we get: Y is a parent of John (where Y=sue)

So we get: George is a grandfather of John

A final example

Example 6: Establish "G is a grandfather of john"

1. Look for "G is a grandfather of john" in KB. Nothing found.
2. Look for "If . . . then G is a grandfather of john" in KB.
   Find If X is a father of Y and Y is a parent of Z then X is a grandfather of Z.
3. - Establish "G is a father of Y". (Note two variables in the query!)
     1. Look for "G is a father of Y" in KB. Nothing found.
     2. Look for "If . . . then G is a father of Y" in KB.
        Find If U is a child of V and V is male then V is a father of U.
     3. – Establish "Y is a child of G." (Note that U=Y and V=G)
          1. Look for "Y is a child of G" in the KB.
             Find john is a child of sue. (First choice that matches!)

We will eventually fail with this Y and G, and must consider other choices!

Eventually we will find sue is a child of george and then succeed as before.

Some properties of back-chaining

Back-chaining has some desirable features:

- it is *goal-directed*: You work your way back from what you are trying to establish towards what you know.
- it is *sound*: Anything that can be established is entailed by the KB.
- it is (sometimes) *complete*: Back-chaining does not miss any entailments *provided that* it does not get stuck in a loop.

  Consider this KB:

  fido is a collie.
  If X is a poodle then X is a dog.
  If X is a poodle then X is a poodle. (This one is trouble!)
  If X is a collie then X is a dog.

  This KB entails “fido is a dog”, but back-chaining gets stuck! (Why?)

- it forms the basis of *Prolog programming*, which we turn to next.

3

The Prolog Language

Prolog in perspective

PROLOG = PROgramming in LOGic.

Invented in 1971 by a Frenchman, Alain Colmeraurer, while visiting at the University of Montreal, for the purpose of *natural language parsing*.

Simultaneously invented by an American, Carl Hewitt, at MIT in Boston (under the name **Planner** there).

Mainly developed and promoted in Europe, especially Britain and France. Now a commercial product, with many available systems.

In this course: a system called **SWI-Prolog**.

Radically different concept of programming from more conventional computer languages such as C++, Java, Python etc.

Now widely used for AI applications, as well as many other uses.

Prolog programs

Prolog programs are simply *knowledge bases* consisting of atomic and conditional sentences as before, but with a sightly different notation.

Here is the "family" example as a Prolog program:

family.pl

```
% This is the Prolog version of the family example

child(john,sue).   child(john,sam).
child(jane,sue).   child(jane,sam).
child(sue,george).   child(sue,gina).

male(john).   male(sam).     male(george).
female(sue).  female(jane).  female(june).

parent(Y,X) :- child(X,Y).
father(Y,X) :- child(X,Y), male(Y).
opp_sex(X,Y) :- male(X), female(Y).
opp_sex(Y,X) :- male(X), female(Y).
grand_father(X,Z) :- father(X,Y), parent(Y,Z).
```

Now let's look at all the pieces in detail . . .

Constants and variables

A Prolog *constant* must start with a *lower case* letter, and can then be followed by any number of letters, underscores, or digits.

> A constant may also be a *quoted-string*: any string of characters (except a single quote) enclosed within single quotes.

So the following are all legal constants:

```
sue opp_sex mamboNumber5 'Who are you?'
```

A Prolog *variable* must start with an *upper case* letter, and can then be followed by any number of letters, underscores, or digits.

So the following are all legal variables:

```
X P1 MyDog The_biggest_number Variable_27b
```

Prolog also has *numeric* terms, which we will return to later.

Atomic sentences

The atomic sentences or *atoms* of Prolog have the following form:

$$\textit{predicate}(\textit{term}_1, \ldots, \textit{term}_k)$$

where the predicate is a constant and the terms are either constants or variables.

Note the punctuation:

- immediately after the predicate, there must be a *left parenthesis*;
- between each term, there must be a *comma*;
- immediately after the last term, there must be a *right parenthesis*.

The number of terms k is called the *arity* of the predicate.
If $k = 0$, the parentheses can be left out.

Conditional sentences

The conditional sentences of Prolog have the following form:

$$\textit{head} \;\; \text{:–} \;\; \textit{body}_1,\, \ldots,\, \textit{body}_n$$

where the head and each element of the body is an atom.

Note the punctuation:

- immediately after the head, there must be a *colon* then *hyphen*, `:-`.
- between each element of the body, there must be a *comma*.

If $n = 0$, the `:-` should be omitted.

In other words, an atomic sentence is just a conditional sentence where the body is empty!

Prolog programs, defined

A Prolog program is a sequence of clauses, where a *clause* is an atomic or conditional sentence *terminated by a period*.

A *comment* may appear at the end of a program or just before a constant or variable, and starts with a *percent sign*. The rest of the line is ignored.

Here is a very short but legal Prolog program:

```
zzz.
```

(Why is this a program?)

Here is a somewhat longer one:

```
% This is an age-old bit of logical reasoning
mortal(X) :- man(X).   % All men are mortal
man(socrates).         % Socrates is a man
```

And a longer one still is the family example from before.

Typically, a program is stored in a file with a name like `family.pl`.

Prolog programs in use

Prolog programs only do something in response to *queries*.

Here is the normal way of running a Prolog program:

1. We begin by preparing a file containing the Prolog program.
2. We run the Prolog system and ask it to load the Prolog program.
3. Then we repeatedly do the following:
   - we pose a query to the system
   - we wait for Prolog to return an answer

   Often we save these queries and answers in a separate file.
4. Finally we exit the Prolog system.

The details of how these steps are done vary from system to system.

SWI-Prolog under Linux

Here is a typical run of an SWI-Prolog system under Linux.

The highlighted text is what the user types.

```
[skywolf] pl                                        This starts SWI-Prolog
Welcome to SWI-Prolog (Multi-threaded, Version 5.2.11)
Copyright (c) 1990-2003 University of Amsterdam.
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to redistribute it under certain conditions.
Please visit http://www.swi-prolog.org for details.

For help, use ?- help(Topic).  or ?- apropos(Word).

?- [family].                                        This loads the 'family.pl' file
% family compiled 0.00 sec, 2,724 bytes

Yes
?- father(sam,jane).                                This is a query to Prolog

Yes                                                 Prolog's answer is 'yes'.   Success!

?- halt.                                            This is how we exit Prolog
[skywolf]
```

Prolog under Windows or Mac OS is similar.

Simple queries

In its simplest form, a *query* is just an atom, with or without variables, and terminated with a period.

Posing a query is asking Prolog to *establish* the atom using back-chaining, just as we did by hand before.

If the query has no variables, there are three possible outcomes:

1. Prolog answers `Yes` (or `true`): the atom can be established;

   See the query `father(sam,jane)` on the previous slide.

2. Prolog answers `No` (or `false`): the atom cannot be established, and Prolog runs out of alternatives to try;

3. Prolog does not answer: the atom cannot be established, and Prolog continues to try alternatives.

Queries with variables

If a query has variables, then there are three possible outcomes:

1. Prolog answers `No`: the atom cannot be established for any value of the variables, and Prolog runs out of alternatives to try;
2. Prolog does not answer: the atom cannot be established for any value of the variables, and Prolog continues to try alternatives;
3. Prolog displays values for the variables for which it can establish the query. At this point the user has some choices:
   - type a *space* or *return*, in which case, Prolog answers `Yes` and the query is done;
   - type a *semi-colon*, in which case, Prolog tries to find new values for the variables, with the same three possible outcomes.

Example queries with variables

For these and the following examples, we assume that we are running Prolog and that the file `family.pl` has already been loaded.

```
?- father(sam,X).                 Who is Sam the father of?
X = john                          Just the first answer, please.
Yes

?- father(U,V).                   Who is a father of whom?
U = sam
V = john ;                        Any more?

U = sam
V = jane ;                        Any more?

U = george
V = sue ;                         Any more?
No
```

Conjunctive queries

More generally, queries can be *sequences* of atoms separated by commas and again terminated by a period.

These queries are understood *conjunctively*, just like the body of a conditional sentence. (So “,” plays the role of “and”.)

```
?- female(F), parent(sam,F).
F = jane ;
No
```

This asks for an F such that F is female and Sam is a parent of F.

In other words:

Q: Who is a female that Sam is a parent of?

A: Jane. And this is the only value that can be found.

Negation in queries

The special symbol \+ (meaning “not”) can also be used in front of an atom in a query to flip between “Yes” and “No”.

```
?- child(john,george).
No
```

Is John a child of George

```
?- \+ child(john,george).
Yes
```

Is John not a child of George

```
?- parent(X,john), female(X).
X = sue
Yes
```

Who is a parent of John and female

```
?- parent(X,john), \+ female(X).
X = sam
Yes
```

Who is a parent of John and not female

Note: `female(X)` is not the same as `\+ male(X)`.

for example, the query `female(gina)` will fail, but `\+ male(gina)` will succeed since Gina is not known to be male.

How does Prolog answer queries?

Before looking at more general queries and programs, let us go through the steps Prolog follows in back-chaining in more detail.

Consider: `parent(X,john), \+ female(X).`

1. We start by replacing variables in the query by new names to make sure they do not conflict with variables in the Prolog program.

   ```
   parent(_G312,john), \+ female(_G312).
   ```

2. We start working on the query `parent(_G312,john)`, and find the clause that relates `parent` to `child`. This leaves us with:

   ```
   child(john,_G312), \+ female(_G312).
   ```

3. We start working on the query `child(john,_G312)`, and find the clause `child(john,sue).` This part is tentatively done and leaves us with:

   ```
   \+ female(sue).
   ```

. . . continued

4. We start working on the query `\+ female(sue)`.
   Since this is a negated query, we consider the unnegated version.

      We start working on the query `female(sue)`, and find the clause in the program. We return success.

   Since the unnegated query succeeds, the negated query fails.
   We go back to our most recent choice point (at step 3) and reconsider.

5. We return to the query `child(john,_G312)`, and this time, we find the clause `child(john,sam).` This leaves us with:

   ```
   \+ female(sam).
   ```

6. We start working on the query `\+ female(sam)`.
   Since this is a negated query, we try the unnegated version.

      We start working on the query `female(sam)`, but we cannot find anything that matches. We return failure.

   Since the unnegated query fails, the negated query succeeds.
   This leaves us nothing left to do so.

7. We return *success*, noting that the value of `X` is `sam`.

Getting Prolog to trace its behaviour

It is possible to get Prolog to trace its back-chaining operation.

How you do so varies from system to system.

Here is what the output typically looks like:

```
?- parent(X,john), \+ female(X).
Call:  (8) parent(_G312, john)
Call:  (9) child(john, _G312)
Exit:  (9) child(john, sue)
Exit:  (8) parent(sue, john)
Call:  (8) female(sue)
Exit:  (8) female(sue)
Redo:  (9) child(john, _G312)
Exit:  (9) child(john, sam)
Exit:  (8) parent(sam, john)
Call:  (8) female(sam)
Fail:  (8) female(sam)

X = sam
```

There are 4 types of entries:

Call: start working on an atomic query;

Exit: success on an atomic query;

Fail: failure on an atomic query;

Redo: go back to a choice point and reconsider.

Caution: variables and negation

```
?- parent(X,john), \+ female(X).
X = sam
Yes
```

```
?- \+ female(X), parent(X,john).
No
```

What is going here?

In the second case, Prolog starts by working on `\+ female(X)`.
Since `female(X)` succeeds, the negated version fails.
We can go no further with the query!

To get `\+ female(X)` to succeed (as in the first case), we need to already have a value for the `X` (e.g. `X=sam`).

Moral: When using a negated query with variables, the variables should already have tentative values from earlier queries.

The equality predicate

Prolog allows elements in a query of the form *term*$_1$ = *term*$_2$, where the terms are either constants or variables. This query succeeds when the terms are equal, and fails otherwise.

```
?- Z=Y, X=jack, Y=X, \+ jill=Z.
Z = jack
Y = jack
X = jack
Yes

?- X=Y, jack=X, \+ Y=jack.
No

?- Z=Y.
Z = _G180
Y = _G180
Yes
```

These answers mean that the values are equal but otherwise unconstrained.

Using equality

You should never have to use an *unnegated equality* predicate:

Instead of: `child(john,X), child(jane,Y), X=Y.`

use: `child(john,X), child(jane,X).`

However, *negated equality* does come in handy:

```
?- parent(sam,X), \+ X=john.
X = jane
Yes
```

Is Sam the parent of someone other than John?

```
?- male(X), male(Y), male(Z), \+ X=Y, \+ X=Z, \+ Y=Z.
X = john
Y = sam
Z = george

Yes
```

Are there at least 3 males?

This gives us a simple form of *counting*.

Review: terms

Before looking at Prolog programs in more detail, we review what we have seen so far, with a few minor extensions.

- A *constant* is one of the following:
  - a string of characters enclosed within single quotes.
  - a lower case letter optionally followed by letters, digits and underscores.
- A *variable* is an upper case letter optionally followed by letters, digits and underscores.
- A *number* is a sequence of one or more digits, optionally preceded by a minus sign, and optionally containing a decimal point.
- A *term* is a constant, variable, or number.

Review: queries and programs

- A *predicate* is written like a constant.
- An *atom* is a predicate optionally followed by terms enclosed within parentheses and separated by commas.
- A *literal* is one of the following, optionally preceded by a \+ symbol:
  - an atom;
  - two terms separated by the = symbol.
- A *query* is a sequence of one or more literals, separated by commas, and terminated by a period.
- A *clause* is one of the following:
  - an atom terminated with a period;
  - an atom followed by the `:-` symbol followed by a query.
- A *program* is a sequence of one or more clauses.

Back-chaining, revisited

Having looked at Prolog queries in some detail, we now examine Prolog back-chaining, starting with a very simple program.

likes.pl

```
%  This is a program about who likes what kinds of food.
likes(john,pizza).                  % John likes pizza.
likes(john,sushi).                  % John likes sushi.
likes(mary,sushi).                  % Mary likes sushi.
likes(paul,X) :- likes(john,X).     % Paul likes what John likes.
likes(X,icecream).                  % Everybody likes ice cream.
```

We start by looking at how we decide which clauses to use in answering queries.

The way clauses are selected during back-chaining is through a matching process called *unification*.

Unification

Two atoms with distinct variables are said to *unify* if there is a substitution for the variables that makes the atoms identical.

- a query such as `likes(john,Y)` unifies with `likes(john,pizza)` in the first clause of the program, for `Y=pizza`.
- a query such as `likes(paul,pizza)` unifies with `likes(paul,X)` in the fourth clause, for `X=pizza`.
- a query such as `likes(jane,Y)` unifies with `likes(X,icecream)` in the last clause, for `X=jane` and `Y=icecream`.

In all cases, the queries will eventually succeed!

Note that both the query and the head of a clause from the program may contain variables.

Examples of unification

The following pairs of atoms unify:

- `p(b,X,b)` and `p(Y,a,b)` for `X`=a and `Y`=b
- `p(X,b,X)` and `p(a,b,Y)` for `X`=a and `Y`=a
- `p(b,X,b)` and `p(Y,Z,b)` for `X`=Z and `Y`=b
- `p(X,Z,X,Z)` and `p(Y,W,a,Y)` for `X`=a, `Z`=a, `Y`=a, and `W`=a.

The following pairs of atoms do not unify:

- `p(b,X,b)` and `p(b,Y)`
- `p(b,X,b)` and `p(Y,a,a)`
- `p(X,b,X)` and `p(a,b,b)`
- `p(X,b,X,a)` and `p(Y,Z,Z,Y)`

Renaming variables

Unification is not concerned with where the two atoms come from (which one is from a query, and which one is from a program).

However, during back-chaining, Prolog *renames* the variables in the clauses of the program before attempting unification.

Example: Consider the query `likes(X,pizza), \+ X=john`.

> Prolog first finds `likes(john,pizza)`, but this eventually fails.
>
> It eventually considers the clause whose head is `likes(paul,X)`, but this cannot unify with `likes(X,pizza)` as is.
>
> So Prolog renames the variable `X` in that clause to a totally new variable, for example, `X1`.
>
> Then `likes(X,pizza)` unifies with `likes(paul,X1)`, and the query goes on to eventually succeed.

Back-chaining: another view

Here is a slightly more accurate picture of how back-chaining works to establish a conjunctive query $A_1, A_2, \ldots, A_n$.

1. If $n = 0$, there's nothing to do, exit with success.
2. Otherwise, try each clause in the *program* from top to bottom in turn:
   (a) Assume that the current clause has head H and body $B_1, \ldots B_m$,
   (where for atomic sentences, the body is empty and so $m = 0$).
   (b) Rename all the variables in the clause.
   (c) Test to see if the head H *unifies* with the first atom A_1.
   If it does not, move on to the next clause.
   (d) Otherwise, try to establish $B_1^*, \ldots, B_m^*, A_2^*, \ldots A_n^*$, where the * means the result of replacing variables by their values from the unification.
   (e) If this works, exit with success; if not, go on to the next clause.
3. If you get this far, you've tried all the clauses in the program. Return failure.

4

Writing Prolog Programs

Writing meaningful programs

We can now consider writing much more complex Prolog programs.

It is important to consider carefully what we intend the predicates to *mean*, and to make sure that the clauses we write are all *true*.

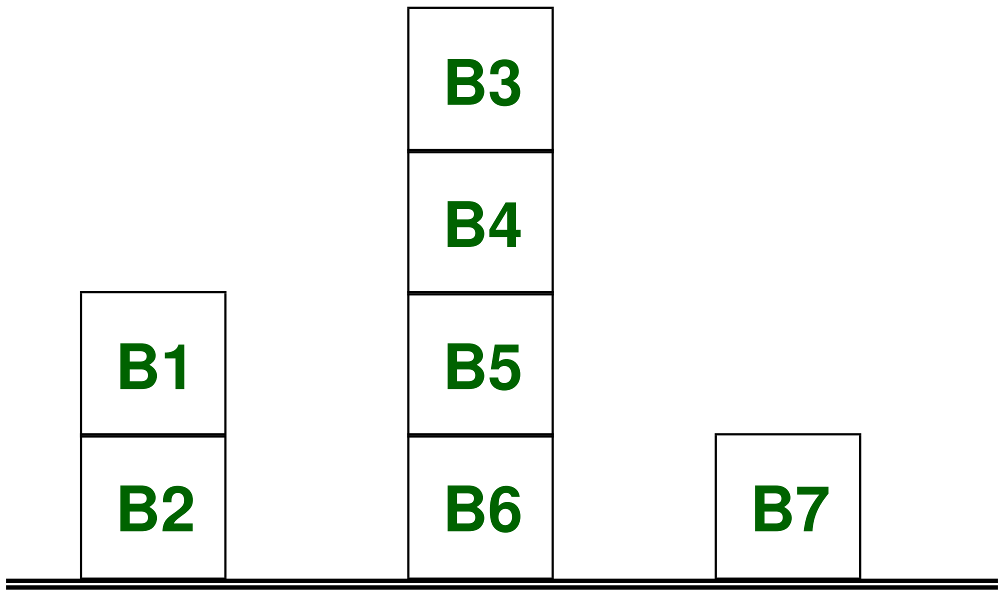
We also need to consider if we have included *enough* clauses to allow Prolog to draw appropriate conclusions involving the predicates.

Finally, we need to consider how *back-chaining* will use the clauses when actually establishing conclusions.

To summarize, we need

1. the truth, and nothing but;
2. the whole truth;
3. presented in the right form for back-chaining.

A bigger example: a world of blocks



We would like to describe the scene and get Prolog to determine that

- Block 3 is above Block 5
- Block 1 is to the left of Block 7
- Block 4 is to the right of Block 2

A blocks world program

blocks.pl

```
% on(X,Y) means that block X is directly on top of block Y.                  1
on(b1,b2).   on(b3,b4).   on(b4,b5).   on(b5,b6).                            2
                                                                             3
% just left(X,Y) means that blocks X and Y are on the table                  4
% and that X is immediately to the left of Y.                                5
just_left(b2,b6).   just_left(b6,b7).                                        6
                                                                             7
% above(X,Y) means that block X is somewhere above block Y                   8
% in the pile where Y occurs.                                                9
above(X,Y) :- on(X,Y).                                                      10
above(X,Y) :- on(X,Z), above(Z,Y).                                          11
                                                                            12
% left(X,Y) means that block X is somewhere to the left                     13
% of block Y but perhaps higher or lower than Y.                            14
left(X,Y) :- just_left(X,Y).                                                15
left(X,Y) :- just_left(X,Z), left(Z,Y).                                     16
left(X,Y) :- on(X,Z), left(Z,Y).     % leftmost is on something.            17
left(X,Y) :- on(Y,Z), left(X,Z).     % rightmost is on something.           18
                                                                            19
% right(X,Y) is the opposite of left(X,Y).                                  20
right(Y,X) :- left(X,Y).                                                    21
```

Note: The small line numbers displayed here are not part of Prolog.

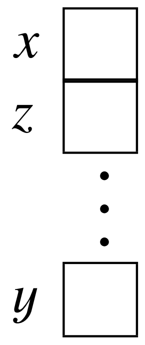
Blocks world queries

```
?- above(b1,b2).
 T Call: (7) above(b1, b2)     %  The main query
 T Call: (8) on(b1, b2)        %  First try on(b1,b2) using line 10
 T Exit: (8) on(b1, b2)        %  Succeeds because of line 2
 T Exit: (7) above(b1, b2)     %  So the main query succeeds

Yes

?- above(b3,b5).
 T Call: (8) above(b3, b5)     %  The main query
 T Call: (9) on(b3, b5)        %   - Try on(b3,b5)
 T Fail: (9) on(b3, b5)        %     This fails
 T Redo: (8) above(b3, b5)     %  Reconsider
 T Call: (9) on(b3, _L205)     %   - Try on(b3,Z) from line 11
 T Exit: (9) on(b3, b4)        %     This succeeds for Z=b4
 T Call: (9) above(b4, b5)     %   - Now try above(Z,b5) for Z=b4
 T Call: (10) on(b4, b5)       %       - Try on(b4,b5)
 T Exit: (10) on(b4, b5)       %         This succeeds
 T Exit: (9) above(b4, b5)     %     This succeeds
 T Exit: (8) above(b3, b5)     %  The main query succeeds

Yes
```

Recursion

The example on the previous slide makes use of *recursion*.

A recursive clause is one in which the predicate of the head is the same as a predicate mentioned in the body.



Line 11 in the program:

```
above(X,Y) :- on(X,Z), above(Z,Y).
```

If x is on z and z is above y, then x is above y.

Most modern programming languages (Java, Python, etc.) provide recursion, which is usually considered to be an advanced technique.

In fact, it is really quite simple and lies at the heart of Prolog programming.

Writing recursive programs

You can recognize that recursion is needed when you have a predicate that involves using another predicate *some number* of times.

For example, for a block x to be above a block y, there must be some number k of intermediate blocks such that the initial x is on x_1 which is on x_2 *etc.*, and where the final x_k is on y. (The k can be 0.)

When writing the clauses for a recursive predicate, we need to somehow take care of *all* the possible cases for k.

in our scene, k might be 0 or 3 or 7 or 20 or 155,262

Q: How can we do this, since are *infinitely many* possibilities?

A: We use a form of *mathematical induction*!

Recursion as mathematical induction

1. write clauses to handle the base case where $k = 0$;

   For the predicate above, this means the x is directly on the y.

   ```
   above(X,Y) :- on(X,Y).
   ```

2. assume the case for $k = n$ has already been taken care of, and write clauses to handle the case where $k = (n + 1)$;

   For the predicate above, suppose the number $k = (n + 1)$.

   Then the x must be directly on some other block z, where there are only n intermediate blocks between this z and y.

   So we assume that the predicate already works for z and y, and use that to write a clause for x and y.

   ```
   above(X,Y) :- on(X,Z), above(Z,Y).
   ```

A harder blocks world query

```
?- left(b1,b7).
 T Call: (7) left(b1, b7)           % The main query
 T Call: (8) just_left(b1, b7)      %  - Try just_left(b1,b7) from line 15
 T Fail: (8) just_left(b1, b7)      %    This fails
 T Redo: (7) left(b1, b7)           % Reconsider
 T Call: (8) just_left(b1, _L205)   %  - Try just_left(b1,Z) from line 16
 T Fail: (8) just_left(b1, _L205)   %    This also fails
 T Redo: (7) left(b1, b7)           % Reconsider
 T Call: (8) on(b1, _L205)          %  - Try on(b1,Z) from line 17
 T Exit: (8) on(b1, b2)             %    This succeeds for Z=b2
 T Call: (8) left(b2, b7)           %  - Try left(Z,b7) for Z=b2
 T Call: (9) just_left(b2, b7)      %      - Try just_left(b2,b7)
 T Fail: (9) just_left(b2, b7)      %        This fails
 T Redo: (8) left(b2, b7)           %    Reconsider
 T Call: (9) just_left(b2, _L223)   %      - Try just_left(b2,Z1)
 T Exit: (9) just_left(b2, b6)      %        This succeeds for Z1=b6
 T Call: (9) left(b6, b7)           %      - Try left(Z1,b7) for Z1=b6
 T Call: (10) just_left(b6, b7)     %          - Try just_left(b6,b7)
 T Exit: (10) just_left(b6, b7)     %            This succeeds
 T Exit: (9) left(b6, b7)           %        This succeeds
 T Exit: (8) left(b2, b7)           %    This succeeds
 T Exit: (7) left(b1, b7)           % The main query succeeds!

Yes
```

Non-terminating programs

Using recursion, it is possible to write programs that go on *forever*.

For example, instead of line 11, suppose we had written this:

```
above(X,Y) :- above(Z,Y), on(X,Z).
```

What this says is *true*: if z is above y and x is on z, then x is above y.

However, if we have the query `above(b2,b1)`, we get the following:

```
above(b2,b1)   ⇒
above(Z0,b1), on(b2,Z0)   ⇒
above(Z1,b1), on(Z0,Z1), on(b2,Z0)   ⇒
above(Z2,b1), on(Z1,Z2), on(Z0,Z1), on(b2,Z0)   ⇒   ...
```

Eventually Prolog reports that it has run out of memory!

Avoiding non-termination

When writing recursive programs, there is no simple way to *guarantee* that they will terminate.

However, a good rule of thumb is that when a clause is recursive, the body should contain at least one atom *before* the recursive one to provide a new value for one of the variables.

```
Instead of        above(X,Y) :- above(Z,Y), on(X,Z).
we should write   above(X,Y) :- on(X,Z), above(Z,Y).
                                         ⇑
```

These two mean the same thing in English, and together with line 10, they correctly characterize the predicate `above`.

But because of the left-to-right order of Prolog, the second one will try the recursive predicate only after it has found a value for the variable `Z`.

Excessive search

A trickier problem is seen with the query `left(b3,b3)`.

A trace of it starts as follows

```
?- left(b3,b3).
 T Call: (7) left(b3, b3)           % The main query
 T Call: (8) just_left(b3, b3)
 T Fail: (8) just_left(b3, b3)
 T Redo: (7) left(b3, b3)           % Reconsider
 T Call: (8) just_left(b3, _L205)
 T Fail: (8) just_left(b3, _L205)
 T Redo: (7) left(b3, b3)           % Reconsider again
 T Call: (8) on(b3, _L205)          %   Try on(b3,Z)
 T Exit: (8) on(b3, b4)
 T Call: (8) left(b4, b3)           %   Then start left(b4,b3)
 T Call: (9) just_left(b4, b3)
 T Fail: (9) just_left(b4, b3)
 T Redo: (8) left(b4, b3)           %   Reconsider ...
```

and continues this way for a total of 1500 lines before finally failing!

What is the problem?

The way `left(X,Y)` is characterized, we proceed in three stages:

1. we first see if the two blocks are on the table (using `just_left`);
2. then we see if the `X` is on another block and try `left` with that;
3. then we see if the `Y` is on another block and try `left` with that.

The result is that we consider the *same pairs of blocks* repeatedly.

If we consider just the `left` predicate on pairs of blocks then

- for (3,3), we also need to try (4,3) and (3,4);
- for (4,3), we also need to try (5,3) and (4,4);
- for (3,4), we also need to try (4,4) and (3,5).

So (4,4) gets considered twice, (5,5) gets considered *four* times, (6,6) gets considered *eight* times, and so on.

Algorithms

There is no easy way to write programs and be sure that they are not doing excessive work.

A good part of Computer Science involves analyzing computational problems and coming up with *effective* ways of solving them using computers.

> = comparing different *algorithms* for solving a problem.

Each problem has to be approached on its own terms.

> In Prolog, we would need to think about what sorts of approaches will allow back-chaining to do its job effectively.

In this course, we will *not* spend a lot of time worrying about algorithms!

A glimpse of a better algorithm

A more effective way of characterizing `left(X,Y)` is as follows:

- find the block below `X` that is on the table; call it `X1`;
- find the block below `Y` that is on the table; call it `Y1`;
- test whether `X1` is to the left of `Y1` using only `just_left`.

Here is a fragment of a Prolog program to do this:

```
left(X,Y) :- bottom(X,X1), bottom(Y,Y1), table_left(X1,Y1).

bottom(X,X) :- \+ on(X,Y).  % Note the use of \+ !
bottom(X,X1) :- on(X,Y), bottom(Y,X1).

table_left(X,Z) :- just_left(X,Z).
table_left(X,Z) :- just_left(X,Y), table_left(Y,Z).
```

5

Case Study: Satisfying Constraints

Thinking revisited

In this course we will look at a number of activities that require thought.

Unfortunately, a lot of our own thinking happens so *quickly* that we have a hard time describing what is actually taking place!

For example, figuring out what the “it” means in

The trophy would not fit into the brown suitcase
because it was too small.

One example where this is *not* the case is when we are solving puzzles like those that appear in the recreational section of a newspaper.

What we will see is that the thinking needed to solve a wide variety of these reasoning puzzles can be formulated in the same way, in terms of what we will call *constraint satisfaction problems*.

Constraint satisfaction

Many challenging tasks that appear to require thinking have this form:

- some choices to be made
- some constraints to be satisfied

In this section, we will examine a collection of such reasoning tasks:

1. map colouring
2. Sudoku
3. cryptarithmetic
4. 8 queens
5. logic puzzles
6. scheduling

all of which can be solved in Prolog in a similar way.

Generate and test

We will solve these problems using a method called *generate and test*.

The idea is simple:

- we guess at the value of one or more variables;
- we confirm that we are satisfied with the values we get;
- we backtrack and guess at new values if we need to.

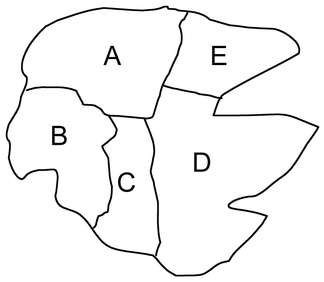
We have already seen this behaviour in answering conjunctive queries:

```
?- male(X), child(john,X).
```

This finds an `X` that is male and a child of John:

- generate an `X` that is male: the first one we find is `john`.
- test that this `X` satisfies `child(john,X)`. It does not. So backtrack.
- generate another `X` that is male: the next one we find is `sam`.
- test that this `X` satisfies `child(john,X)`. It does. We are done.

1. Colouring a map



Five countries: `A, B, C, D, E`

Three colours: `red, white, blue`

Constraint: Neighbouring countries must have different colours on the map.

map.pl

```
% solution(A,B,C,D,E) holds if A,B,C,D,E are colors
% that solve a map-coloring problem from the text.
solution(A,B,C,D,E) :-
   color(A), color(B), color(C), color(D), color(E),
   \+ A=B, \+ A=C, \+ A=D, \+ A=E, \+ B=C, \+ C=D, \+ D=E.

% The three colors are these.
color(red).
color(white).
color(blue).
```

Then we get:

```
?- solution(A,B,C,D,E).
A=red  B=white  C=blue  D=white  E=blue    % + other solutions too
```

Constraint satisfaction problems

The general form of these problems is this:

- some number of *variables*

  e.g. colours on the map

- values to be chosen from finite *domains*

  e.g. each colour must be one of red, white or blue

- there are *constraints* among subsets of the variables

  e.g. adjacent countries have different colours

A *solution* to a constraint satisfaction problem is an assignment of a value to each variable (taken from the domain of the variable) such that all of the constraints are satisfied.

Using generate and test

The simplest way to solve a constraint satisfaction problem using Prolog is to use generate and test.

So the general form is like this (see the map colouring):

```
solution(Variable_1, ..., Variable_n) :-
    domain_1(Variable_1),
        ...
    domain_n(Variable_n),

    constraint_1(Variable, ..., Variable),
        ...
    constraint_m(Variable, ..., Variable).
```

Sometimes the generation is interleaved with the testing.

Output in Prolog

It is often convenient to be able to produce output in Prolog other than just the values of variables.

Prolog provides two special atoms for queries or bodies of clauses:

- `write(`*term*`)`

  always succeeds and has the effect of printing the term

- `nl`

  always succeeds and has the effect of starting a new line

```
?- X=blue, nl, write('The value of X is '), write(X),
   write(', I believe,'), nl, write('  and that is it!').

The value of X is blue, I believe,
  and that is it!

X = blue

Yes
```

Output for map colouring

For example, for the map colouring, imagine we have the previous program together with the following additional clause:

```
print_colours :-
  solution(A,B,C,D,E), nl,
  write('Country A is coloured '), write(A), nl,
  write('Country B is coloured '), write(B), nl,
  write('Country C is coloured '), write(C), nl,
  write('Country D is coloured '), write(D), nl,
  write('Country E is coloured '), write(E), nl.
```

Then we get:

```
?- print_colours.      % Note that the query now has no variables

Country A is coloured red
Country B is coloured white
Country C is coloured blue
Country D is coloured white
Country E is coloured blue

Yes
```

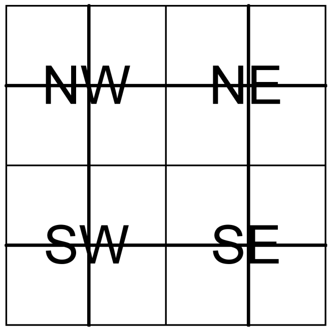
2. Mini Sudoku

We now consider a small version of the Sudoku puzzle.

The problem:

We are given a 4×4 grid, where some of the entries are blank and some contain numbers between 1 and 4. Our job is to fill in all of the entries with numbers between 1 and 4 so that

- the numbers in each row (1,2,3,4) are unique,
- the numbers in each column (1,2,3,4) are unique,
- the numbers in each square (NW,NE,SW,SE) are unique.



A small digression: The anonymous variable

In many cases, we need to use a variable, but the value of the variable is never used again.

The *underscore* symbol is called the *anonymous variable*.

It means: “there needs to be a value here, but I don’t care what it is.”

```
?- likes(john,X).      % What does John like?
X = pizza
Yes

?- likes(john,_).      % Does John like anything?
Yes

?- likes(_,_).         % Does anyone like anything?
Yes
```

Note each occurrence of the anonymous variable can be for a different value. So the last query above behaves like `likes(X,Y)`.

Sudoku queries

What we would like is to provide a *partial solution* as a query and have Prolog print out a *complete solution*.

```
?- sudoku(            % Note the use of the anonymous _ variable.
   1, 4, _, _,
   _, _, 4, _,
   2, _, _, _,
   _, _, _, 3
).
A solution to this puzzle is
   1 4 3 2
   3 2 4 1
   2 3 1 4
   4 1 2 3
Yes
```

So the `sudoku` predicate will take 16 arguments, each representing a cell of the puzzle.

Sudoku as constraint satisfaction

There are a total of 16 variables that need to get values from the domain $\{1, 2, 3, 4\}$, subject to the given constraints about rows, columns, and squares.

The easiest way to program this is to define a predicate `uniq` that takes 4 arguments and ensures that they are all different.

Here is one way to define it:

```
% Ensure that none of P, Q, R or S are equal.
uniq(P,Q,R,S) :-
  \+ P=Q, \+ P=R, \+ P=S,
          \+ Q=R, \+ Q=S,
                  \+ R=S.
```

We will use variants of this idea over and over for different problems.

In the case of Sudoku, we also want P, Q, R, S to be numbers from 1 to 4.

Here is the full program (it's the longest one we will see!) . . .

```
% The main predicate. Solve the puzzle and print the answer.
% The variable Rij stands for the number in row i and column j.
sudoku(R11,R12,R13,R14,R21,R22,R23,R24,R31,R32,R33,R34,
         R41,R42,R43,R44) :-
   solution(R11,R12,R13,R14,R21,R22,R23,R24,R31,R32,R33,R34,
              R41,R42,R43,R44),
   nl, write('A solution to this puzzle is'), nl,
   printrow(R11,R12,R13,R14), printrow(R21,R22,R23,R24),
   printrow(R31,R32,R33,R34), printrow(R41,R42,R43,R44).

% Print a row of four numbers with spaces between them.
printrow(P,Q,R,S) :- write('  '), write(P), write(' '), write(Q),
   write(' '), write(R), write(' '), write(S), nl.

%------------------------------------------------------------------
solution(R11,R12,R13,R14,R21,R22,R23,R24,R31,R32,R33,R34,
           R41,R42,R43,R44) :-
   uniq(R11,R12,R13,R14), uniq(R21,R22,R23,R24),       % rows 1,2
   uniq(R31,R32,R33,R34), uniq(R41,R42,R43,R44),       % rows 3,4
   uniq(R11,R21,R31,R41), uniq(R12,R22,R32,R42),       % cols 1,2
   uniq(R13,R23,R33,R43), uniq(R14,R24,R34,R44),       % cols 3,4
   uniq(R11,R12,R21,R22), uniq(R13,R14,R23,R24),       % NW and NE
   uniq(R31,R32,R41,R42), uniq(R33,R34,R43,R44).       % SW and SE

% uniq holds if P,Q,R,S are all distinct nums (from 1 to 4).
uniq(P,Q,R,S) :- num(P),  num(Q),  num(R),  num(S),
                 \+ P=Q, \+ P=R, \+ P=S, \+ Q=R, \+ Q=S, \+ R=S.

% The four numbers to go into each cell
num(1).  num(2).  num(3).  num(4).
```

Handling a full Sudoku

It is quite easy to generalize the 4×4 solution to the 9×9 case.

> Instead of 12 constraints (rows, columns, squares) over 16 variables,
> we get 27 constraints (rows, columns, squares) over 81 variables.

But will this work?

> **in principle**: definitely yes!
>
> **in practice**: definitely **no**!

The obvious 9×9 version did not terminate running Prolog on a Mac G5 over a full weekend (66 hours)!

Q: What is the problem?

A: The 9×9 Sudoku is *much larger* than the 4×4 one!

Search space

One useful measure of the *size* of a constraint satisfaction problem is the size of the search space.

Search Space: the different ways variables can be assigned values from their domains before considering the constraints.

For the mini Sudoku: each of 16 variables can take on 4 values.

So there are this many possibilities: $4 \times \cdots \times 4 = 4^{16}$
which is about 4×10^9 (4 billion).
This is considered *small*!

For the full Sudoku: each of 81 variables can take on 9 values.

So there are this many possibilities: $9 \times \cdots \times 9 = 9^{81}$
which is about 2×10^{77}
This is *enormous*!

Big numbers

Q: Are super fast computers able to explore these search spaces?

A: **No!**

Suppose we have a computer that can explore 1 billion possibilities in the search space per second. (No existing computer can do this.)

Suppose we magically make it one billion times faster. (Things now get done in under *1 second* that would have taken over *10 years*)

Suppose we have 1 billion of these super fast computers all working together, sharing the job of exploring the search space.

Q: How many possibilities would we now be able to handle per second?

A: $10^9 \times 10^9 \times 10^9 = 10^{27}$

Q: How long would it take to go through all of Sudoku?

A: $10^{77}/10^{27} = 10^{50}$ seconds = 3×10^{40} centuries!

A better strategy: minimize guessing

The problem with the way we solve map colouring and Sudoku is that we rely too much on *guessing*.

> Guessing = making a random selection from the search space
> and then backtracking if it is not right

Sudoku experts do not think this way. They do very little guessing, and only for the most difficult of the puzzles.

Instead they repeatedly try to see if the constraints *force* values for certain entries based on other known values.

Programming this type of algorithm for Sudoku would take some work.

But we can examine the basic idea by looking at another constraint satisfaction problem.

Numbers as terms

Before we go on to look at another constraint satisfaction problem in Prolog, we need to examine *numbers* in Prolog.

Since numbers are terms, they can appear in programs and queries anywhere a constant or variable can appear (as we saw in Sudoku).

So we can have a Prolog program that contains clauses like this:

```
age(donna,23).
age(andy,22).
current_temperature(-5).
```

These can be used in queries like this:

```
?- age(donna,N).          % How old is Donna?
N = 23
Yes

?- age(X,N), \+ N=23.     % Is anyone not 23?
X = andy
N = 22
Yes
```

Numeric expressions

In addition, Prolog provides some special arithmetic operations that can appear in queries or in the bodies of clauses:

$E_1 < E_2$	less than
$E_1 > E_2$	greater than
$E_1 =< E_2$	less than or equal
$E_1 >= E_2$	greater than or equal
E_1 `=:=` E_2	equal
Var `is` E	equal

The E_i here are *numeric expressions*, that is, formulas built out of numbers and variables using +, -, *, / and other operations.

Here is an example expression: `sqrt(3*(X+Y)/2)`.

Variables in expressions

The arithmetic operations above require the variables that appear in the expressions to already have values that are numbers.

Giving a variable a value is called *instantiating* the variable.

```
?- X = 4, X > 2.
X = 4
Yes

?- X = 4, X+5 =:= (X-1)**2.
X = 4
Yes

?- X > 2, X = 4.
ERROR: Arguments are not sufficiently instantiated

?- X = Y, X > 2, Y = 4.
ERROR: Arguments are not sufficiently instantiated
```

In this way, arithmetic operations are like the \+ operation.

Using arithmetic expressions

The left hand variable in the `is` operation does *not* need to be instantiated (and so can get a value from the right hand expression).

```
?- Y=2, X=5, X is Y+4.
No

?- Y=2, X is Y+4.
X = 6
Y = 2
Yes
```

Note: expressions are not terms, and so cannot be used as arguments to predicates.

this is ok: `?- q(Y), X is Y+4, p(X,Z).`

this is not ok: `?- q(Y), p(Y+4,Z).`

Using `is` is the typical way of getting the value of an arithmetic expression to use with another predicate.

Arithmetic programs

Using the arithmetic operations, we can write Prolog programs that perform numeric calculations.

Suppose we have the following sorts of clauses in a program:

```
birth_year(donna,1986).
birth_year(andy,1987).
current_year(2009).
```

We might have the following to compute the age of a person:

```
% Age is the difference between the current and birth years.
age(Person,X) :-
     birth_year(Person,Y1),
     current_year(Y2),
     X is Y2-Y1.              % age = current - birthyear
```

Then we get the following:

```
?- age(andy,N).        % How old is Andy?
N = 22
Yes
```

More arithmetic

Although this will *not* be the focus in this course, we can also write ordinary numerical procedures in Prolog.

Here is how we would compute $n! = n \times (n-1) \times \ldots \times 2 \times 1$.

We want a predicate `factorial` so that the query `factorial(5,X)` succeeds with `X=120` (since $5 \times 4 \times 3 \times 2 \times 1 = 120$).

Here's the entire program:

```
% factorial(N,X) holds when X = N!

% This recursive program uses the property
% that if n > 1, then n! = n * (n-1)!

factorial(1,1).     % The base case
factorial(N,X) :-
   N > 1,           % Note: the N must be instantiated
   N1 is N-1,
   factorial(N1,X1),        % Get (N-1)! recursively
   X is N*X1.               % N! = N*(N-1)!
```

3. Cryptarithmetic

Cryptarithmetic puzzles are puzzles of the form

```
  SEND
+ MORE    % Each letter stands for a distinct digit
------    % Leading digits must not be 0
 MONEY
```

Variables: S, E, N, *etc.* and "carry digits"

Domains: $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ for digits and $\{0, 1\}$ for carry digits

Constraints: unique digits, $S > 0$, $M > 0$, and

$$(D + E) \bmod 10 = Y \qquad (D + E)/10 = C_1$$

$$(N + R + C_1) \bmod 10 = E \qquad (N + R + C_1)/10 = C_{10}$$

$$(E + O + C_{10}) \bmod 10 = N \qquad (E + O + C_{10})/10 = C_{100}$$

$$(S + M + C_{100}) \bmod 10 = O \qquad (S + M + C_{100})/10 = M$$

A first program for SEND+MORE=MONEY

sendmore1.pl

```
% solution(...) holds for a solution to SEND+MORE=MONEY.
solution(S,E,N,D,M,O,R,Y) :-
   uniq_digits(S,E,N,D,M,O,R,Y), S > 0, M > 0,
   Y is (D+E) mod 10, C1 is (D+E) // 10,
   E is (N+R+C1) mod 10, C10 is (N+R+C1) // 10,
   N is (E+O+C10) mod 10, C100 is (E+O+C10) // 10,
   O is (S+M+C100) mod 10, M is (S+M+C100) // 10.

% uniq(...) holds if the arguments are all distinct digits.
uniq_digits(S,E,N,D,M,O,R,Y) :-
   dig(S), dig(E), dig(N), dig(D), dig(M), dig(O), dig(R), dig(Y),
   \+ S=E, \+ S=N, \+ S=D, \+ S=M, \+ S=O, \+ S=R, \+ S=Y,
           \+ E=N, \+ E=D, \+ E=M, \+ E=O, \+ E=R, \+ E=Y,
                   \+ N=D, \+ N=M, \+ N=O, \+ N=R, \+ N=Y,
                           \+ D=M, \+ D=O, \+ D=R, \+ D=Y,
                                   \+ M=O, \+ M=R, \+ M=Y,
                                           \+ O=R, \+ O=Y,
                                                   \+ R=Y.

% The digits
dig(0). dig(1). dig(2). dig(3). dig(4).
dig(5). dig(6). dig(7). dig(8). dig(9).
```

Running the first program

This programs works correctly:

```
?- solution(S,E,N,D,M,O,R,Y).   % SEND + MORE = MONEY
S = 9,                          % 9567 + 1085 = 10652
E = 5,
N = 6,
D = 7,
M = 1,
O = 0,
R = 8,
Y = 2
```

However there is a problem:

> It takes over 90 seconds to find the solution, even on a very fast computer. (Try it!)

Can we do better?

We must try to minimize the guessing.

Minimize guessing: rule 1

Rule 1: Avoid guessing any value that is fully determined by other values and later testing if the guess is correct.

for example, instead of

```
uniq3(A,B,C),          % guess at A, B, and C
B is (A+C) mod 10      % then test if B is ok
```

we should use something like

```
uniq2(A,C),            % guess at A and C only
B is (A+C) mod 10      % calculate B once
uniq3(A,B,C)           % ensure they are all unique
```

The first version has a search space of $10 \times 10 \times 10 = 1000$.

The second has a search space of $10 \times 10 = 100$.

Minimize guessing: rule 2

Rule 2: Avoid placing independent guesses between the generation and testing of other values.

for example, instead of

```
dig(A), dig(B),      % guess at A and B
dig(C), dig(D),      % guess at C and D
A > B                % then test if A > B
```

we should use

```
dig(A), dig(B),      % guess at A and B
A > B,               % test if A > B
dig(C), dig(D),      % guess at C and D
```

In the first version, if we guess badly for A and B, we only get to reconsider *after* we have gone through *all* the values for C and D.

In the second version, if we guess badly for A and B, we reconsider immediately, before we consider any values for C and D.

Reordering the constraints

Here is a second version of the `solution` predicate.

It uses exactly the same constraints, but in a *different order*.

sendmore2.pl

```
% solution(...) holds for a solution to SEND+MORE=MONEY.
solution(S,E,N,D,M,O,R,Y) :-
   dig(D), dig(E),
   Y is (D+E) mod 10, C1 is (D+E) // 10,
   dig(N), dig(R),
   E is (N+R+C1) mod 10, C10 is (N+R+C1) // 10,
   dig(E), dig(O),
   N is (E+O+C10) mod 10, C100 is (E+O+C10) // 10,
   dig(S), S > 0, dig(M), M > 0,
   O is (S+M+C100) mod 10, M is (S+M+C100) // 10,
   uniq_digits(S,E,N,D,M,O,R,Y).

% The rest of the program is as before.
```

It gives the same answer as before, but this time in .07 seconds!

Can we do even better?   Yes, but we will not pursue it here. . .

Producing nicer output

Finally, we can add a little sugar to sweeten the output somewhat:

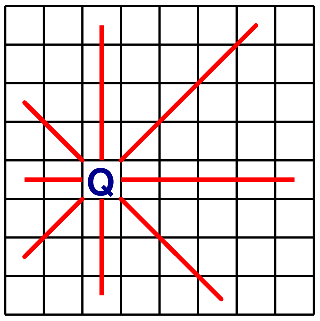
```
    % A new top-level predicate
    print_solution :-
      solution(S,E,N,D,M,O,R,Y), nl,
      write('    '), write(S), write(E), write(N), write(D), nl,
      write('+   '), write(M), write(O), write(R), write(E), nl,
      write('   '), write('-----'), nl,
      write('   '), write(M), write(O), write(N), write(E),
         write(Y), nl.
```

This gives us the following:

```
?- print_solution.

   9567
+  1085
  -----
  10652

Yes
```

4. The 8 queens problem

Problem: Place 8 queens on a 8×8 chessboard such that no queen can capture any other.



Each queen must be on its own

- – row
- | column
- \ left diagonal
- / right diagonal

We can reasonably assume that each row will have one queen.

So the job is to choose a *column* for each queen

Variables: $C_1, C_2, C_3, C_4, C_5, C_6, C_7, C_8,$

where C_i is the location for the queen in row i.

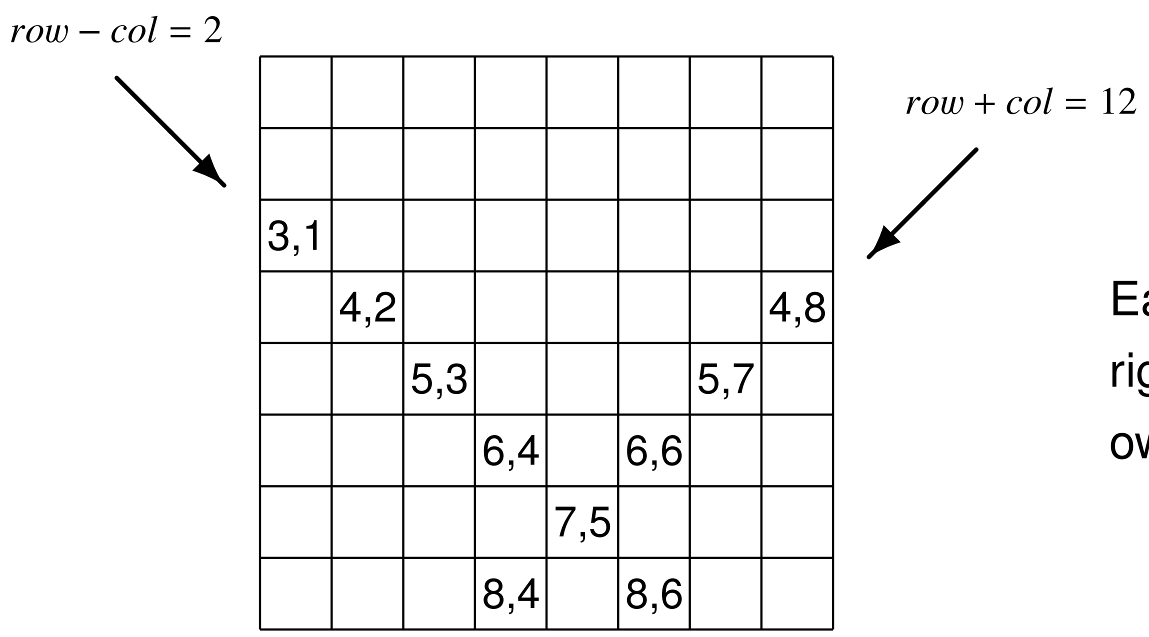
Domains: the columns 1, 2, 3, 4, 5, 6, 7, 8.

So $C_5 = 3$ would mean that the queen in row 5 goes in column 3

What is a diagonal?

Observe:

- along a left diagonal, $(row - column)$ stays constant
- along a right diagonal, $(row + column)$ stays constant



Each left diagonal and right diagonal has its own constant

Capturing a queen

A queen on (row_1, col_1) can capture a queen on (row_2, col_2) iff any of the following holds:

- they are on the same row: $row_1 = row_2$
- they are on the same column: $col_1 = col_2$
- they are on the same left diagonal: $row_1 - col_1 = row_2 - col_2$
- they are on the same right diagonal: $row_1 + col_1 = row_2 + col_2$

In Prolog:

```
%  cap(R1,C1,R2,C2): a queen on (R1,C1) can capture one on (R2,C2).

cap(R,_,R,_).       %  Note the use of the _ variable
cap(_,C,_,C).       %  Here too.
cap(R1,C1,R2,C2) :- R1-C1 =:= R2-C2.
cap(R1,C1,R2,C2) :- R1+C1 =:= R2+C2.
```

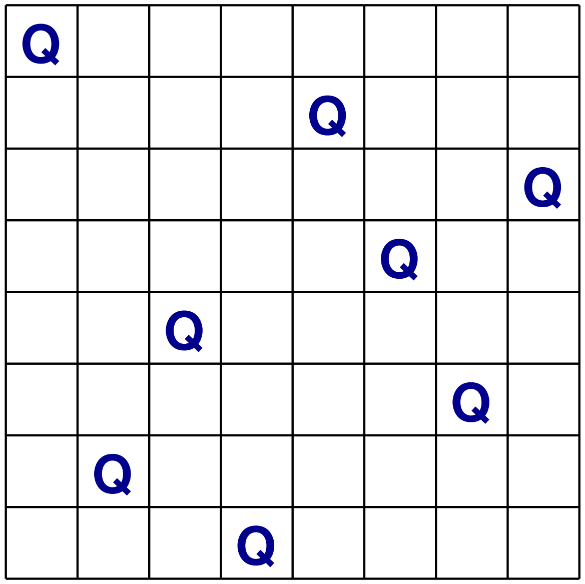
The 8 queens in Prolog

queens.pl

```
% Solve the 8-queens problem.
solution(C1,C2,C3,C4,C5,C6,C7,C8) :-
  col(C1),
  col(C2), \+ cap(2,C2,1,C1),
  col(C3), \+ cap(3,C3,1,C1), \+ cap(3,C3,2,C2),
  col(C4), \+ cap(4,C4,1,C1), \+ cap(4,C4,2,C2), \+ cap(4,C4,3,C3),
  col(C5), \+ cap(5,C5,1,C1), \+ cap(5,C5,2,C2), \+ cap(5,C5,3,C3),
      \+ cap(5,C5,4,C4),
  col(C6), \+ cap(6,C6,1,C1), \+ cap(6,C6,2,C2), \+ cap(6,C6,3,C3),
      \+ cap(6,C6,4,C4), \+ cap(6,C6,5,C5),
  col(C7), \+ cap(7,C7,1,C1), \+ cap(7,C7,2,C2), \+ cap(7,C7,3,C3),
      \+ cap(7,C7,4,C4), \+ cap(7,C7,5,C5), \+ cap(7,C7,6,C6),
  col(C8), \+ cap(8,C8,1,C1), \+ cap(8,C8,2,C2), \+ cap(8,C8,3,C3),
      \+ cap(8,C8,4,C4), \+ cap(8,C8,5,C5), \+ cap(8,C8,6,C6),
          \+ cap(8,C8,7,C7).

% The columns
col(1). col(2). col(3). col(4). col(5). col(6). col(7). col(8).

% cap(R1,C1,R2,C2): a queen on (R1,C1) can capture one on (R2,C2).
cap(R,_,R,_).                        % Note the use of the _ here
cap(_,C,_,C).                        % and here, too.
cap(R1,C1,R2,C2) :- R1-C1 =:= R2-C2.
cap(R1,C1,R2,C2) :- R1+C1 =:= R2+C2.
```

Running the 8 queens

```
?- solution(C1,C2,C3,C4,C5,C6,C7,C8).

C1 = 1,
C2 = 5,
C3 = 8,
C4 = 6,
C5 = 3,
C6 = 7,
C7 = 2,
C8 = 4

Yes
```



This is the first solution found.

This means that while (2,3) is safe from (1,1), the remaining six queens cannot be safely placed on the board if there is a queen on (1,1) and (2,3).

Similarly (2,4) is a dead end, and only (2,5) leads to a solution (requiring (3,8), even though (3,2) and (3,7) are both safe at that point).

5. Logic puzzles

Example:

In conversation, Chris, Sandy and Pat discovered that they had distinct occupations and played distinct musical instruments. Also

1. *Chris is married to the doctor.*
2. *The lawyer plays the piano.*
3. *Chris is not the engineer.*
4. *Sandy is a patient of the violinist.*

Who plays the flute?

To solve puzzles like these, we need to determine what the variables are, what values they can have, and how the given information leads to constraints on possible solutions.

Logic puzzle as constraint satisfaction

To determine who plays the flute, it is clear that we will need to figure out who plays what, and who has what job.

This suggests the following:

Variables: `Doctor`, `Lawyer`, `Engineer`, `Piano`, `Violin`, `Flute`

Domain: {`sandy`, `chris`, `pat`}

For the constraints, we need to use the fact that

- if x is married to y, then $x \neq y$.
- if x is a patient of y, then $x \neq y$ and y is the doctor.

The logic puzzle as a Prolog program

logic1.pl

```
% A logic puzzle involving jobs and musical instruments,
solution(Flute) :-

   % Distinct occupations and instruments
   uniq_people(Doctor,Lawyer,Engineer),
   uniq_people(Piano,Violin,Flute),

   % The four clues
   \+ chris = Doctor,      % Chris is married to the doctor.
   Lawyer = Piano,         % The lawyer plays the piano.
   \+ Engineer = chris,    % The engineer is not Chris.
   Violin = Doctor,        % Sandy is a patient of
   \+ sandy = Violin.      %    the violinist.

% uniq(...) is used to generate three distinct people.
uniq_people(A,B,C) :- person(A),  person(B),  person(C),
                      \+ A=B, \+ A=C, \+ B=C.

% The three given people
person(chris).   person(sandy).   person(pat).
```

Hidden variables

Sometimes the only way to express the constraints is to imagine that there are variables other than the ones mentioned in the puzzle.

cf. the carry digits in cryptarithmetic

Example: as before with Chris, Sandy, and Pat, except that (3) is

3. Pat is not married to the engineer.

It is useful to think of terms of new variables, for Chris' spouse, Sandy's spouse, and Pat's spouse.

- Domain: Chris, Sandy, Pat or *none*
- Constraints: only 4 legal arrangements

— nobody is married	— Chris and Sandy
— Chris and Pat	— Sandy and Pat

The logic puzzle revisited

logic2.pl

```
% A second logic puzzle involving jobs and musical instruments
solution(Flute) :-

   uniq_people(Doctor,Lawyer,Engineer),
   uniq_people(Piano,Violin,Flute),

   % Generate values for the three spouse variables.
   spouses(Chris_spouse,Sandy_spouse,Pat_spouse),

   Chris_spouse = Doctor,      % Chris is married to the doctor.
   Lawyer = Piano,             % The lawyer plays the piano.
   \+ Pat_spouse = Engineer,   % Pat is not married to the engineer.
   Violin = Doctor,            % Sandy is a patient of
   \+ sandy = Violin.          %    the violinist.

uniq_people(A,B,C) :-
   person(A),  person(B),  person(C), \+ A=B, \+ A=C, \+ B=C.

person(chris).   person(sandy).   person(pat).

% spouses(X,Y,Z): X,Y,Z can be spouses of Chris,Sandy,Pat.
spouses(none,none,none).       % Nobody is married.
spouses(sandy,chris,none).     % Chris and Sandy are married.
spouses(pat,none,chris).       % Chris and Pat are married.
spouses(none,pat,sandy).       % Sandy and Pat are married.
```

An old classic: the zebra puzzle

On a street, there are 5 houses of different colours occupied by 5 individuals of different nationalities, who own 5 different pets, drink 5 different beverages, and smoke 5 different brands of (American) cigarettes. Also,

1. The occupant of the red house is English.
2. The Spaniard owns a dog.
3. The coffee drinker lives in a green house.
4. The Ukrainian drinks tea.
5. The ivory house is to the left of the green one.
6. The snail owner smokes Winstons.
7. The person in the yellow house smokes Kools.
8. The occupant in the middle house drinks milk.
9. The Norwegian lives in the leftmost house.
10. The Chesterfield smoker lives next to the fox owner.
11. The Kool smoker lives next to the horse owner.
12. The orange juice drinker smokes Lucky Strikes.
13. The Parliament smoker is Japanese.
14. The Norwegian lives next to a blue house.

Who owns the zebra?

What is the domain?

Observe that most of the clues will end up as statements of equality between variables

4. The Ukrainian drinks tea.

   `Ukrainian = TeaDrinker`

6. The snail owner smokes Winstons.

   `SnailOwner = WinstonSmoker`

But what about clues involving *position*?

5. The ivory house is to the left of the green one.

This needs to relate to other positional constraints like

8. The occupant in the middle house drinks milk.
9. The Norwegian lives in the leftmost house.
10. The Chesterfield smoker lives next to the fox owner.

A positional domain

As long as we enforce the appropriate equality and inequality constraints, we are free to take the values of the variables from any set.

The positional constraints suggest we take values from a *positional ordering* of the five houses: 1 (leftmost), 2 (left middle), 3 (middle), 4 (right middle), and 5 (rightmost).

Then we use the following:

```
pos(1).  pos(2).  pos(3).  pos(4).  pos(5).

left(1,2).  left(2,3).  left(3,4).  left(4,5).

next_to(X,Y) :- left(X,Y).
next_to(X,Y) :- left(Y,X).

leftmost_pos(1).
middle_pos(3).

uniq_pos(P1,P2,P3,P4,P5) :- pos(P1), pos(P2), pos(P3), pos(P4), pos(P5),
   \+ P1=P2, \+ P1=P3, \+ P1=P4, \+ P1=P5, \+ P2=P3, \+ P2=P4, \+ P2=P5,
   \+ P3=P4, \+ P3=P5, \+ P4=P5.
```

zebra.pl

```
% This is a solution to the classic zebra puzzle.
solution(Zebra,England,Spain,Ukraine,Japan,Norway) :-

   % The fourteen clues
   England=Red, Spain=Dog, Coffee=Green, Ukraine=Tea,
   left(Ivory,Green), Winston=Snail, Kool=Yellow,
   middle_pos(Milk), leftmost_pos(Norway),
   next_to(Chesterfield,Fox),  next_to(Kool,Horse),
   LuckyStrike=OJ, Japan=Parliament, next_to(Norway,Blue),

   % The five lists: houses, nations, pets, drinks, cigarettes
   uniq_pos(Green,Red,Yellow,Ivory,Blue),
   uniq_pos(England,Spain,Ukraine,Japan,Norway),
   uniq_pos(Dog,Snail,Zebra,Fox,Horse),
   uniq_pos(Tea,Milk,OJ,Coffee,OtherDrink),
   uniq_pos(Winston,Kool,Parliament,Chesterfield,LuckyStrike).

%--------------   The positional predicates   -------------------

uniq_pos(P1,P2,P3,P4,P5) :-
   pos(P1), pos(P2), pos(P3), pos(P4), pos(P5),
   \+ P1=P2, \+ P1=P3, \+ P1=P4, \+ P1=P5, \+ P2=P3,
   \+ P2=P4, \+ P2=P5, \+ P3=P4, \+ P3=P5, \+ P4=P5.

pos(1).  pos(2).  pos(3).  pos(4).  pos(5).

leftmost_pos(1).   middle_pos(3).

left(1,2).  left(2,3).  left(3,4).  left(4,5).

next_to(X,Y) :- left(X,Y).
next_to(X,Y) :- left(Y,X).
```

Running the zebra

Since the values of the variables are positional numbers, it takes some extra work to get an answer we like. So far we get:

```
?- solution(Zebra, ...).
Zebra = 5  ...
```

We can see that the zebra is in house 5.   But who is the owner?

A better solution is this:

```
solution(Zebra,England,Spain,Ukraine,Japan,Norway) :-
   ... % puzzle solution as before.

print_solution :-
   solution(Z,E,S,U,J,N), pmatch(Z,E,englishman), pmatch(Z,S,spaniard),
   pmatch(Z,U,ukranian), pmatch(Z,J,japanese), pmatch(Z,N,norwegian).

pmatch(X,X,Name) :- nl, write('The zebra owner is '), write(Name), nl.
pmatch(X,Y,Name) :- \+ X=Y.
```

Then we get:

```
?- print_solution.
The zebra owner is japanese
```

One to think about . . .

Six couples - the ROSENs, the QUINs, the PALMERs, the OGLETHORPEs, the NEWKIRKs, and the MORGANs, departed via different flights from New York for a trip to Europe.

The wives (not respectively) are LOIS, KATE, JESSICA, INGRID, HELEN, and GLENDA. The husbands (again not respectively) are FRED, EDWARD, DAVID, CHARLES, BERTRAM, and ALAN.

These 12 persons are (in no particular order) a PHOTOGRAPHER, a NEWSPAPER COLUMNIST, a COLLEGE PROFESSOR, a MAGAZINE EDITOR, a HIGH SCHOOL PRINCIPAL, a NOVELIST, a PROFESSIONAL GOLFER, a PHYSICIAN, a TELEVISION SCRIPTWRITER, a PUBLIC RELATION DIRECTOR, a FASHION DESIGNER, and a PSYCHOANALYST. Of course, all these careers can be followed by members of either sex...

From among six countries, DENMARK, ENGLAND, FRANCE, ITALY, NORWAY, and SPAIN, each couple elected to visit four, spending exactly one week in each. No two couples visited the same four countries, or spent the same week in any given country.

The problem is to:

1. link up each wife with her husband,
2. match up the first and last names of everybody,
3. name the occupation of each of the 12 individuals
4. name the country visited by each couple during each of the four weeks.

Some additional information is available...

1. The first week found EDWARD in DENMARK, the HIGH SCHOOL PRINCIPAL in ENGLAND, the FASHION DESIGNER in FRANCE, INGRID in ITALY, the OGLETHORPEs in NORWAY, and the PSYCHOANALYST in SPAIN.

2. ALAN visited ENGLAND, FRANCE, ITALY, and SPAIN, not necessarily in that order.

3. DENMARK was visited in succession by the PHOTOGRAPHER, JESSICA, BERTRAM, and the COLLEGE PROFESSOR.

4. CHARLES and HELEN and the COLLEGE PROFESSOR are three of the four people who did not visits ENGLAND.

5. GLENDA was in NORWAY after the MAGAZINE EDITOR had been there but before either the NEWKIRKs or the PSYCHOANALYST.

6. FRED and his wife limited their picture taking to black-and-white stills, the ROSENs shot color slides exclusively, and the magazine editor and spouse took only videos. INGRID and her husband were the only couple who didn't take at least one camera on the trip.

7. Mr. PALMER and the PSYCHOANALYST and the PHOTOGRAPHER and LOIS all visited NORWAY, not necessarily in that order. No two were there during the same week.

8. KATE and her husband took both stills shots and videos in DENMARK, FRANCE, ITALY, and SPAIN though they did not necessarily tour the countries in that order.

9. The P. R. DIRECTOR and spouse got a beautiful color shot of Queen Elisabeth leaving Buckingham Palace to address Parliament. The following week, they were so engrossed in further picture making that they barely made the flight back to New York.

10.The NOVELIST, the GOLFER and FRED are three of the four people that did not visit Denmark.

11.The SCRIPTWRITER visited DENMARK, ENGLAND, ITALY, and NORWAY, though not necessarily in that order.

12.Just before they reached the midpoint of their trip, HELEN and her husband finished up their last remaining video cartridge on the top of the Eiffel Tower, and they had to record the remainder of their travels via stills.

13.ENGLAND was the last country visited by the NOVELIST and spouse. It had previously been visited, though in no particular order, by Mr. ROSEN, GLENDA, and BERTRAM, all at different times.

14.During the week that the NEWSPAPER COLUMNIST and spouse were in NORWAY, LOIS was in DENMARK, and FRED was in ITALY.

15.SPAIN was visited, in no particular order, by CHARLES, JESSICA, Mrs MORGAN, and the NEWSPAPER COLUMNIST, no two of whom were there at the same time.

16.HELEN went to NORWAY the same week that the PHYSICIAN was in FRANCE.

17.Mrs. NEWKIRK and the PHOTOGRAPHER did not tour ITALY.

18.The NEWSPAPER COLUMNIST, the GOLFER, and the MAGAZINE EDITOR are of the same sex.

19.The NOVELIST and the PHYSICIAN are of the opposite sex.

20.FRANCE was the final country on the GOLFER itinerary.

21.The P. R. DIRECTOR and the HIGH SCHOOL PRINCIPAL are of the same sex.

6. Scheduling

Constraint satisfaction is not just for puzzles.

A more practical application is *scheduling*, that is, finding times and/or places for events.

Example:

> We want to schedule a new class to meet 5 hours each week.
>
> Meeting times are on the hour from 9am to 4pm, on weekdays.
>
> Some of the periods will be taken by previously scheduled courses.
>
> Classes must not meet on successive hours on the same day, or for more than 2 hours total on any one day.

The job here is to find the periods (= days and times) for the 5 classes.

Scheduling as constraint satisfaction

Variables: P_1, P_2, P_3, P_4, and P_5,
where each P_i is one of the 5 periods for the course

Domains: each period P_i is a certain time on a certain day.
We can represent a period as a number: *24hr clock* $+ \ (100 \times$ *day*$)$
For example: 9am Monday is 109,
2pm Tuesday is 214,
4pm Friday is 516.

Constraints: the 5 chosen periods must be

- not already taken
- non-consecutive
- no more than 2 per day
- all different (this is not explicitly stated!)

Handling the periods in Prolog

We can assume that in any given problem, there will be some periods that are taken, for example,

```
taken(109).
taken(216).
```

Then we use the following to generate available periods:

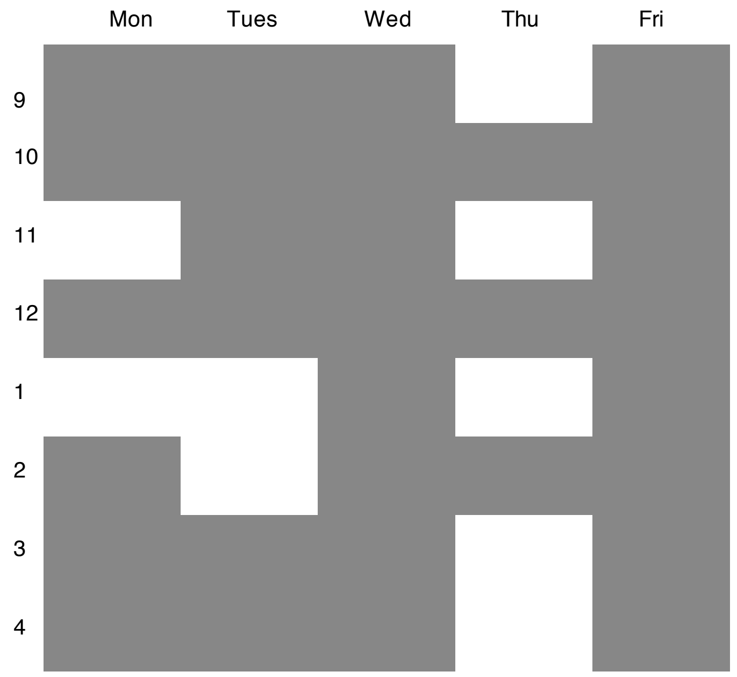
```
% P1,P2,P3,P4,P5 are five distinct available periods
uniq_periods(P1,P2,P3,P4,P5) :-
  period(P1),  period(P2),  period(P3),  period(P4),  period(P5),
  \+ P1=P2,   \+ P1=P3,   \+ P1=P4,   \+ P1=P5,   \+ P2=P3,   \+ P2=P4,
  \+ P2=P5,   \+ P3=P4,   \+ P3=P5,   \+ P4=P5.

% P is a period that is not taken
period(P) :- day(D), time(T), P is T+100*D, \+ taken(P).

day(1).  day(2).  day(3).  day(4).  day(5).        % the 5 days

 time(9).  time(10).  time(11).  time(12).         % the 8 times
time(13).  time(14).  time(15).  time(16).
```

A scheduling program

schedule-top.pl

```
% This program solves a classroom scheduling problem.
solution(P1,P2,P3,P4,P5) :-
  uniq_periods(P1,P2,P3,P4,P5),       % All different periods.
  not_2_in_a_row(P1,P2,P3,P4,P5),     % Not two consecutive hrs.
  not_3_same_day(P1,P2,P3,P4,P5).     % Not three on the same day.

not_2_in_a_row(P1,P2,P3,P4,P5) :-
  \+ seq(P1,P2), \+ seq(P1,P3), \+ seq(P1,P4), \+ seq(P1,P5),
  \+ seq(P2,P3), \+ seq(P2,P4), \+ seq(P2,P5), \+ seq(P3,P4),
  \+ seq(P3,P5), \+ seq(P4,P5).

seq(A,B) :- A =:= B-1.
seq(A,B) :- A =:= B+1.

not_3_same_day(P1,P2,P3,P4,P5) :-
  \+ eqday(P1,P2,P3), \+ eqday(P1,P2,P4), \+ eqday(P1,P2,P5),
  \+ eqday(P1,P3,P4), \+ eqday(P1,P3,P5), \+ eqday(P1,P4,P5),
  \+ eqday(P2,P3,P4), \+ eqday(P2,P3,P5), \+ eqday(P2,P4,P5),
  \+ eqday(P3,P4,P5).

eqday(A,B,C) :- Z is A // 100, Z is B // 100, Z is C // 100.

% The definition of uniq_periods is elsewhere.
```

An example grid



These are added to the program:

```
taken(X) :- X // 100 =:= 3.      % Wednesday
taken(X) :- X // 100 =:= 5.      % Friday
taken(X) :- X // 100 =:= 1,      % Monday
    \+ X=111, \+ X=113.
taken(X) :- X // 100 =:= 2,      % Tuesday
    \+ X=213, \+ X=214.
taken(410). taken(412).          % Thursday
    taken(414).
```

Then we get

```
?- solution(P1,P2,P3,P4,P5).
P1 = 111,   P2 = 113,   P3 = 213,   P4 = 409,   P5 = 411
% Mon 11,   Mon 1pm,    Tue 1pm,    Thur 9am,   Thur 11am
```

6

Case Study: Interpreting Visual Scenes

Visual interpretation

Q: Is constraint satisfaction only useful when we are dealing with a nerdy *puzzle* of some sort?

A: No, it also can apply to ordinary human behaviours like vision.

Q: What is involved with vision? with seeing something?

A: It involves coming up with an *interpretation* for a 2-dimensional grid of colours and intensities.

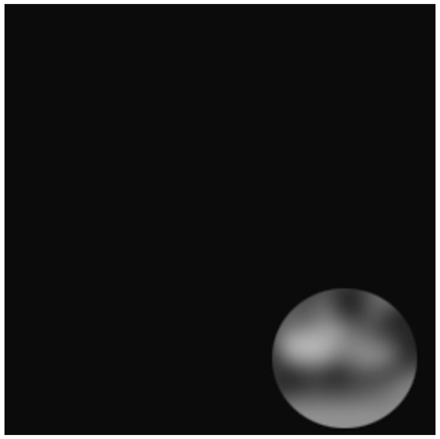
The main question:

what am I looking at?

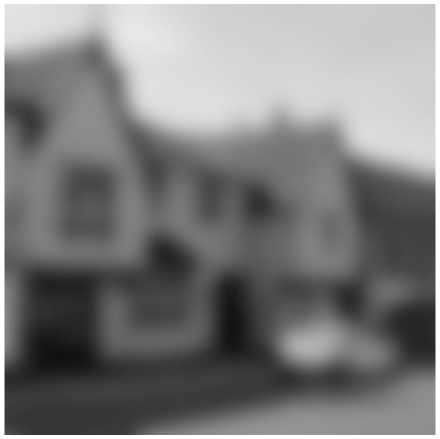
Thinking as part of seeing

Although much of the visual process is something that happens without any thought, part of seeing is using what you *know* about the world.

What is in the circle?



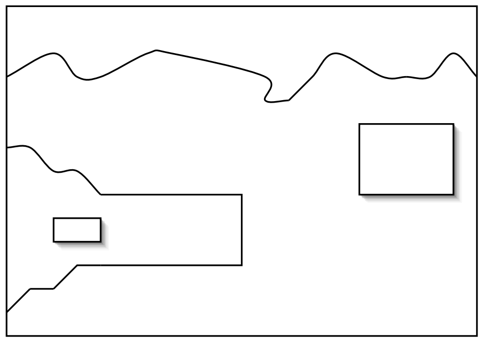
Now look at some surrounding context



A similar but simpler case: aerial sketch maps

Would like to label the regions in the sketch map on the right as either

`grass`, `water`, `pavement`,
`house`, `vehicle`

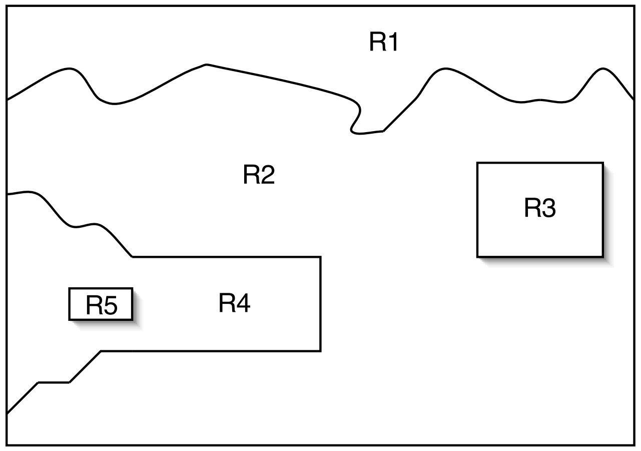


subject to constraints such as

- a region cannot border or be surrounded by another region with the same label
- houses cannot be next to or surrounded by water
- vehicles must be next to or surrounded by pavement
- pavement cannot be completely inside any other region
- houses, vehicles and pavement are regular (straight-edged); grass and water are irregular
- vehicles are small; the other regions are large

Visual properties

The task here is to determine how properties of the image can be translated into suitable constraints.



The more we extract from the image, the more we are able to rule out incorrect interpretations

In our example image:

1. region R5 is small; the others are large
2. region R3 and R5 are regular; R1 and R2 are irregular; R4 could go either way
3. region R1 borders on R2; R2 borders on R4
4. region R3 is inside R2; R5 is inside R4

Visual constraints

We can handle constraints (1) and (2) with facts like

```
large(grass).
small(vehicle).
regular(pavement).
irregular(water).
```

To handle (3), we simply ensure that the two regions do not *violate* any of the given rules about borders:

- the two regions must be different
- they must not be `house` and `water`
- if one is `vehicle`, the other must be `pavement`

Constraint (4) is handled analogously.

What is allowed in an interpretation?

regions.pl

```
% The five types of regions that can appear in an image
region(grass).   region(water).   region(pavement).
region(house).   region(vehicle).

% small(X) holds when region X can be small in an image.
small(vehicle).

% regular(X) holds when region X can be regular in an image.
regular(pavement).  regular(house).  regular(vehicle).

% border(X,Y) holds when region X can border region Y.
border(X,Y) :- \+ bad_border(X,Y), \+ bad_border(Y,X).

   % Unacceptable borders
   bad_border(X,X).
   bad_border(house,water).
   bad_border(vehicle,X) :- \+ X=pavement.

% inside(X,Y) holds when region X can be surrounded by Y.
inside(X,Y) :- \+ bad_inside(X,Y).

   % Unacceptable containment
   bad_inside(X,X).
   bad_inside(house,water).
   bad_inside(vehicle,X) :- \+ X=pavement.
   bad_inside(pavement,_).
```

The aerial sketch map interpretation in Prolog

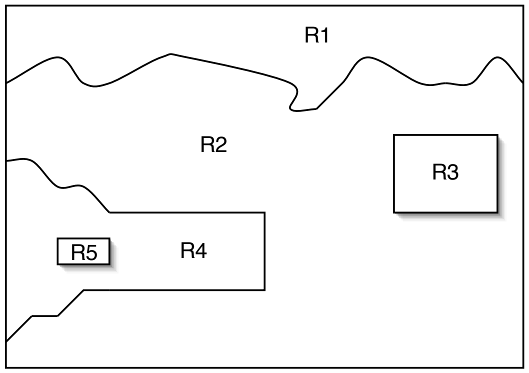
sketch.pl

```
solution(R1,R2,R3,R4,R5) :-
   region(R1), region(R2), region(R3), region(R4), region(R5),
   % Size constraints
      \+ small(R1), \+ small(R2), \+ small(R3),
      \+ small(R4), small(R5),
   % Regularity constraints (none for R4)
       regular(R3), regular(R5), \+ regular(R2), \+ regular(R1),
   % Border constraints
       border(R1,R2), border(R2,R4),
   % Containment constraints
       inside(R3,R2), inside(R5,R4).

% The definitions of region, small, border, etc. are elsewhere.
```

Loading this and the region constraints:

```
?- solution(R1,R2,R3,R4,R5).
R1 = water,  R2 = grass,  R3 = house,
R4 = pavement,  R5 = vehicle
```

and this is the only solution.



7

Lists in Prolog

A recurring problem

A problem that comes up again and again in Prolog is that we have to write collections of predicates that are very similar *except* for the number of arguments they take. For example,

`uniq_person3(X,Y,Z) :- ...` for 3 people
`uniq_person4(X,Y,Z,U) :- ...` for 4 people
`uniq_person5(X,Y,Z,U,V) :- ...` for 5 people

What would be much clearer and convenient, is if we could write just *one* such predicate

`uniq_people(L) :- ...` for any number of people

that would work for any *collection* `L` of people, no matter how big or small.

Prolog provides such a collection. It is called a *list*.

Lists

A list is a sequence of objects which are called the *elements* of the list. Here are some examples:

- `[anna, karenina]`
  A two element list, whose first element is `anna` and whose second element is `karenina`.
- `[intro, to, programming, in, prolog]`
  A five element list.
- `[1, 2, 3, 3, 2, 1]`
  Lists may have repeated elements.
- `[]`
  A list with no elements. The *empty* list.

Structured lists

Lists may contain other lists as elements:

- `[[john, 23], [mary, 14], [hello]]`
  A list whose three elements are also lists.
- `[1, john, ['199y', john]]`
  Another three element list.
- `[[]]`
  A one element list, whose element is the empty list.

Note: A one element list is different from the element itself

`[anna]` is different from `anna`;
`[[]]` is different from `[]`.

Heads and tails

The first element of a non-empty list is called the *head* of the list.

The rest of the non-empty list is called the *tail*.

For example,

- `[a,b,c,d]` has head `a` and tail `[b,c,d]`;
- `[[a],b,[c]]` has head `[a]` and tail `[b,[c]]`;
- `[a,[b,c]]` has head `a` and tail `[[b,c]]`;
- `[[a,b]]` has head `[a,b]` and tail `[]`;
- `[]` has neither head not tail.

Note that the head of a list can be *anything*, but the tail is always a *list*.

List as Prolog terms

Prolog terms are now defined as: constants, variables, numbers, *or lists*. Lists can be written in one of two ways:

1. a *left square parenthesis*, followed by a sequence of terms separated by *commas*, and terminated by a *right square parenthesis*;

   for example: `[1, -4, Z, [a, X, 'b 2'], joe]`

2. a *left square parenthesis*, followed by a non-empty sequence of terms separated by *commas*, followed by a *vertical bar*, followed by a term that denotes a list, and terminated by a *right square parenthesis*.

   for example: `[1, X, 3]` can be written

   ```
   [1 | [X, 3]]     or
   [1, X | [3]]     or
   [1, X, 3 | []].
   ```

Unification with lists

Since variables can appear within lists, we need to be concerned about which pair of lists *unify* (as needed for back-chaining).

The basic idea is this:

- two lists without variables match when they are identical, element for element;
- two lists with distinct variables match when the variables can be given values that make the two lists identical, element for element.

We will see a number of examples of matching lists on the next slides, but the idea can be induced from the example below:

the atom `p([X, 3 | Z])` unifies with `p([b, Y, c, b])`
for `X=b`, `Y=3`, `Z=[c, b]`.

Examples of matching lists

`[]` and `[]`

`[a,b,c]` and `[a,b,c]`

`[X]` and `[a]` with `X=a`

`[a,b,X]` and `[Y,b,Y]` with `X=a Y=a`

`[X,X]` and `[[Y,a,c],[b,a,c]]` with `X=[b,a,c] Y=b`

`[[X]]` and `[Y]` with `X=_G23 Y=[_G23]`

`[a,b,c]` and `[a|[b,c]]`

`[a,b,c]` and `[a|[b|[c]]]`

`[X|Y]` and `[a]` with `X=a Y=[]`

`[a|[b,c]]` and `[a,X,Y]` with `X=b Y=c`

`[a,b,c]` and `[X|Y]` with `X=a Y=[b,c]`

`[X,Y|Y]` and `[a,[b,c],b,c]` with `X=a Y=[b,c]`

Non-matching lists

Lists do not match if they have different numbers of elements or if at least one corresponding element does not match.

Some non-matching pairs:

`[a]` and `[]`

`[]` and `[[]]`

`[X,Y]` and `[U,V,W]`

`[a,b,c]` and `[X,b,X]`

`[X|Y]` and `[]`

Observe: `X` matches anything, including any list
`[X]` matches any list with exactly one element
`[X|Y]` matches any list with at least one element
`[X,Y]` matches any list with exactly two elements

Writing programs that use lists

Programs that use lists usually end up being *recursive* since we may not know in advance how big a list we will need to deal with.

We do know that each list will have *some number* k of elements.

This means that we can work recursively as follows:

- write clauses to handle the base case where $k = 0$ (that is, for the empty list);
- assume the case for $k = n$ has already been taken care of, and write clauses to handle the case where $k = (n + 1)$;

  Assume the list `T` is handled, and write clauses for `[H|T]`.

If we can do this, we will have handled all lists (by mathematical induction).

Now we consider four examples.

1. A list of people

As a first example of a predicate that deals with lists, consider the following:

> Imagine that we already have a predicate `person`(x) that holds when x is a person. We would like to write the clauses for another predicate `person_list`(z) that holds when z is a list whose elements are all people.

This will be a recursive predicate:

- we want clause(s) for the empty list: `person_list([])`.
- assuming we have `person_list(Z)` that we can use, we want clause(s) for a list with one more element: `person_list([P|Z])`.

If we can provide clauses for both of these cases, we are done.

A list of people

Here is a definition of `person_list`:

```
% We assume the person(X) predicate is defined elsewhere.

% The empty list is a (trivial) list of persons.
person_list([]).

% If P is person and Z is a list of persons then [P|Z] is a list of persons.
person_list([P|Z]) :- person(P), person_list(Z).
```

This is how the query `person_list([john,joe,sue])` would work:

- `[john,joe,sue]` unifies with `[P|Z]` for `P=john` and `Z=[joe,sue]`;
- `[joe,sue]` unifies with `[P|Z]` for `P=joe` and `Z=[sue]`;
- `[sue]` unifies with `[P|Z]` for `P=sue` and `Z=[]`;
- `[]` unifies with `[]` and succeeds immediately.

Note how the two notations for lists work together on this.

2. List membership

Suppose we want a predicate that will tell us whether or not something is an element of a list. In other words, we want this behaviour:

```
?- elem(b,[a,b,c,d]).
Yes
?- elem(f,[a,b,c,d]).
No
```

Most Prolog systems provide a predefined predicate (called `member`) with this behaviour. Nonetheless, we will define our own.

This will be a recursive predicate:

- we write clauses for the empty list

  Nothing to write, since the query `elem(X,[])` should always fail.

- assuming a list `L` is handled, write clauses to handle the list `[H|L]`

  The query `elem(X,[H|L])` should succeed if either `X = H` succeeds or `elem(X,L)` succeeds. So two clauses are needed.

The elem predicate defined

Here is how we would define the `elem` predicate in Prolog:

```
% elem(X,L) holds when X is an element of list L

% X is an element of any list whose head is X.
elem(X,[X|_]).

% If X is an element of L, then it is an element of [_|L].
elem(X,[_|L]) :- elem(X,L).
```

Note the use of the *underscore* (anonymous variable):

- in the first clause, we don't care what the tail is;
- in the second, we don't care about the head.

```
?- elem(c,[a,b,c,d,e]).
 T Call: (7) elem(c, [a, b, c, d, e])
 T Call: (8) elem(c, [b, c, d, e])
 T Call: (9) elem(c, [c, d, e])       % Here we get to use the 1st clause
 T Exit: (9) elem(c, [c, d, e])       % which succeeds immediately!
 T Exit: (8) elem(c, [b, c, d, e])
 T Exit: (7) elem(c, [a, b, c, d, e])
```

3. A list of unique people

Using `member` (or `elem`) we can now return to our original motivation:

> write the clauses for a predicate `uniq_people`(z) that holds when z is a list of people (of any size) *that are all different*.

As before, we assume that we have a predicate `person` already defined.

This will be a recursive predicate:

- The empty list `[]` is a (trivial) list of unique people.
- If `L` is a list of unique people, and `P` is a person, and `P` is not an element of `L`, then `[P|L]` is also a list of unique people.

This gives us the following two Prolog clauses:

```
uniq_people([]).
uniq_people([P|L]) :- uniq_people(L), person(P), \+ member(P,L).
```

4. Joining two lists

Suppose we want a list predicate that will join two lists together. In other words, we want this behaviour:

```
?- join([a,b,c,d],[e,f,g],L).
L=[a,b,c,d,e,f,g]
Yes

?- join([],[a,b,c,d],L).
L=[a,b,c,d]
Yes
```

Most Prolog systems provide a predefined predicate (called `append`) with this behaviour. We again define our own.

The predicate will once again be recursive.

However, the predicate `join` needs to work for any *two* lists: the first list can be empty or non-empty, and the second list can also be empty or non-empty.

A recursive definition of join

It is sufficient to do recursion on the *first argument* only:

- write clauses to handle the case where the first argument is `[]` and the second argument is any list `L`;
- write clauses to handle the case where the first argument is `[H|T]` and the second argument is any list `L` (assuming `join` already works when the first argument is `T` and the second argument is `L`).

Here is the program we get:

```
% join(X,Y,Z) means the result of joining X and Y is Z.

% Joining [] and any list L gives L itself.
join([],L,L).

% If joining T and L gives Z, then joining [H|T] and L gives [H|Z].
join([H|T],L,[H|Z]) :- join(T,L,Z).
```

Tracing join

```
?- join([a,b,c,d],[e,f,g],Z).
 T Call: (7) join([a, b, c, d], [e, f, g], _G306)
 T Call: (8) join([b, c, d], [e, f, g], _G378)
 T Call: (9) join([c, d], [e, f, g], _G381)
 T Call: (10) join([d], [e, f, g], _G384)
 T Call: (11) join([], [e, f, g], _G387)           % Here we get to use
 T Exit: (11) join([], [e, f, g], [e, f, g])       % the first clause
 T Exit: (10) join([d], [e, f, g], [d, e, f, g])
 T Exit: (9) join([c, d], [e, f, g], [c, d, e, f, g])
 T Exit: (8) join([b, c, d], [e, f, g], [b, c, d, e, f, g])
 T Exit: (7) join([a, b, c, d], [e, f, g], [a, b, c, d, e, f, g])

Z = [a, b, c, d, e, f, g]

Yes
```

Note that the first argument will be progressively reduced until it becomes `[]`.

At this point, the third argument must be equal to the second.

Then, each recursive query gets a turn to put a new head onto the third argument. (Note the use of `H` in the head of the clause.)

Using member and append

In addition to *testing* for membership in a list, the `member` predicate (or our `elem` predicate) can also be used to *generate* the elements of a list.

```
?- member(X,[a,b,c]).
X = a ;
X = b ;
X = c ;
No
```

So we can write queries (or bodies of clauses) that go through the elements of a list: `... member(X,L), p(X) ...`

In this case, `X` will be assigned to the first element of list `L`.
But if `p(X)` fails, `X` will be assigned to the next element, *etc.*

```
?- member(N,[1,2,-3,4,-5,6,7]), N < 0.
N = -3  ;
N = -5  ;
No
```

Generate finite sets only

Using a predicate to generate a set of candidates that we then test with other predicates is a powerful feature that we will use extensively in this course.

However, some care is required to make sure that predicates used this way will not generate an *infinite* set of candidates.

For example,

```
?- member(3,L).          % what are lists that contain 3 as an element?

L = [3|_G214] ;                    % any list whose head is 3
L = [_G213, 3|_G217] ;             % any list whose 2nd element is 3
L = [_G213, _G216, 3|_G220]        % and so on

Yes

?- member(3,L), a=b.
ERROR: (user://1:71):              % instead of just failing, the query
        Out of global stack        % tries to run forever!
```

More membership queries

```
?- member(a,[X,b,Y]).
X = a                   % X can be a and Y anything
Y = _G163 ;
X = _G157               % or Y can be a and X anything
Y = a ;
No

?- L=[a,X,b], member(3,L).
L = [a, 3, b]           % X must be 3
X = 3 ;
No

?- L=[X,b,Y], member(a,L), member(X,[b,c]).
L = [b, b, a]           % X can be b and Y can be a
X = b
Y = a ;

L = [c, b, a]           % X can be c and Y can be a
X = c
Y = a ;
No
```

Generating lists with append

```
?- append(X,Y,[a,b,c]).             % What pairs of lists when joined give [a,b,c]?
X = []
Y = [a, b, c] ;
X = [a]
Y = [b, c] ;
X = [a, b]
Y = [c] ;
X = [a, b, c]
Y = [] ;
No

?- append([a,b],L,[a,b,d,e,f]).     % [a,b] joined to what gives [a,b,c,d,e,f]?
L = [d, e, f]
Yes

?- append([X,b],[d|L],[a,_,Y,e,f]). % solve for X, Y, and L
X = a
L = [e, f]
Y = d
Yes
```

Defining new predicates using append

Using `append` (or our defined version, `join`), it possible to define several new predicates:

- `front(L1,L2)` holds if list `L1` is the start of list `L2`

```
front(L1,L2) :- append(L1,_,L2).
```

- `last(E,L)` holds if `E` is the last element of list `L`

```
last(E,L) :- append(_,[E],L).
```

- another version of the `elem` predicate!

```
elem(E,L) :- append(_,[E|_],L).
```

This last predicate says that `E` is an element of `L` if `L` is the result of joining some list to another list whose first element is `E`.

In other words: `L` has some elements, then `E`, then some other elements.

Generating finite sets with append

As with the `member` predicate, we are allowed to use variables freely, but we must be careful not to generate infinite sets of candidate lists.

For example, suppose we want a predicate `before` which tells us if one element appears before another in a list.

Here is a first version:

```
% A first version
before(X,Y,L) :- append(_,[X|_],Z), append(Z,[Y|_],L).
```

This version is almost correct:

```
?- before(2,4,[1,2,8,4,3,7]).
Yes

?- before(2,4,[1,2,8,9,3,7]).
ERROR: (user://1:16):
        Out of global stack
```

The problem with the before predicate

The trouble is that the first definition *generates* a list `Z` that contains `X`, and then does other things.

But there are infinitely many such lists!

So if the second part of the query fails, it can run forever (or until the lists are big enough to exhaust all the memory).

Solution: generate the `Z` as a sublist of the given `L` first.

```
% A better solution
before(X,Y,L) :- append(Z,[Y|_],L), append(_,[X|_],Z).
```

This gives us the following:

```
?- before(2,4,[1,2,8,4,3,7]).
Yes

?- before(2,4,[1,2,8,9,3,7]).
No
```

The blocks world using generate and test

From now on, we will use `member` and `append` freely.

Here, for example, is a redone version of the blocks world, using the idea that a *scene* is a list of stacks in left-to-right order, where a *stack* is a list of blocks in top-to-bottom order.

blocks2.pl

```
% This is a list-based version of the blocks-world program.

% X appears before Y in list L.
before(X,Y,L) :- append(Z,[Y|_],L), append(_,[X|_],Z).

% The given blocks-world scene: three stacks of blocks
scene([[b1,b2],[b3,b4,b5,b6],[b7]]).

% above(X,Y) means that block X is somewhere above block Y.
above(X,Y) :- scene(L), member(Stack,L), before(X,Y,Stack).

% left(X,Y) means that block X is somewhere left of block Y.
left(X,Y) :- scene(L), before(Stack1,Stack2,L),
             member(X,Stack1), member(Y,Stack2).

right(Y,X) :- left(X,Y).
```

Tracing a query in the blocks world

```
?- left(b1,b7).

Call: (7) left(b1, b7)
Call: (8) scene(_L205)                           % get the scene
Exit: (8) scene([[b1, b2], [b3, b4, b5, b6], [b7]])
Call: (8) before(_L206, _L207, ...)              % generate two stacks
Exit: (8) before([b1, b2], [b3, b4, b5, b6], ...)  % the 1st pair
Call: (8) member(b1, [b1, b2])                   %   - test X in first
Exit: (8) member(b1, [b1, b2])                   %     YES
Call: (8) member(b7, [b3, b4, b5, b6])           %   - test Y in second
Fail: (8) member(b7, [b3, b4, b5, b6])           %     NO
Redo: (8) member(b1, [b1, b2])
Fail: (8) member(b1, [b1, b2])
Redo: (8) before(_L206, _L207, ...)              % generate another pair
Exit: (8) before([b1, b2], [b7], ...)            % the 2nd pair
Call: (8) member(b1, [b1, b2])                   %   - test X in first
Exit: (8) member(b1, [b1, b2])                   %     YES
Call: (8) member(b7, [b7])                       %   - test Y in second
Exit: (8) member(b7, [b7])                       %     YES
Exit: (7) left(b1, b7)                           % the 2nd pair works!

Yes
```

Review: Programming in Prolog

Programming in Prolog consists in writing clauses that fully characterize the predicates you are interested in.

> **the easier part**: writing the *truth*, the whole truth, and nothing but
>
> **the harder part**: writing it in a way suitable for *back-chaining*

A common concern is knowing when a variable gets instantiated:

- when using `\+` and the arithmetic operations
- when using predicates recursively (see the predicate `above`)
- when generating a value to be tested (see the predicate `before`)

In some cases, predicates can be defined in a way that allows queries with variables to generate values (e.g. `append`).

In other cases, queries must provide values for one or more of the arguments (e.g. in numerical predicates like `factorial`).

8

Case Study: Understanding Natural Language

Natural language

Among all the tasks that appear to require thinking, making sense of a *natural language*, that is, a language like English or Italian or Swahili that is spoken naturally by people, holds a position of honour.

- As with recreational puzzles, it is necessary to use what we know to be able to make sense of English expressions.
  - What does the "it" mean in

    The trophy would not fit into the brown suitcase
    because it was too small.
  - Who does "the American President during the Civil War" refer to?
- However, language also feeds our knowledge!

  We acquire much of what we know not through direct experience, but by being *told* via language!

Interacting with computers

From the very beginnings of AI, researchers have had the goal of interacting with computers in a natural language.

Why?

- ease of use: not having to learn computer language, especially for non-technical users
- richness: refer to objects in a rich descriptive way

  Find me the article written by Jane just before she left on her trip to the Middle East.

- learning via books: much of what we know about the world is written down in books (or in Web pages) in natural language

Status report

To date, success in limited domains:

- limited vocabulary
- limited subject matter
- limited accuracy

Applications:

- language translation
- front-ends to database systems
- other applications

In this course: very simple form of processing of written natural language.

Linguistics

Linguistics is the study of natural language at a variety of levels

- *phonetics*: the sounds in words

  the word "car", as spoken by me, Queen Elizabeth, Rick Mercer

- *morphology*: roots of words, prefixes, suffixes

  ran = run + PAST

  children = child + PLURAL

- *syntax*: how do the words group together?

  Mary kicked the boy in the knee. *vs.*

  Mary kicked the boy in the first row.

continued. . .

More linguistics

- *semantics*: what do the words mean?

  The astronomer spotted a star. *vs.*
  The astronomer married a star.

  The trophy would not fit into the brown suitcase
  because it was too small. *vs.*
  because it was too big.

- *pragmatics*: what are the words being used for?

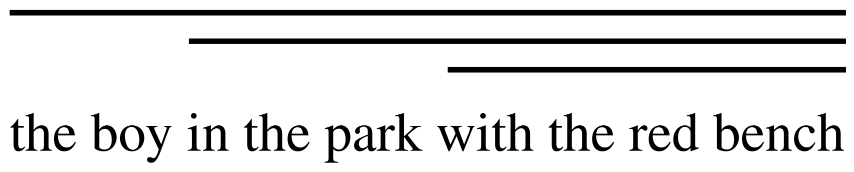
  Can you juggle? *vs.*
  Can you pass the salt?

  I wouldn't turn around if I were you.

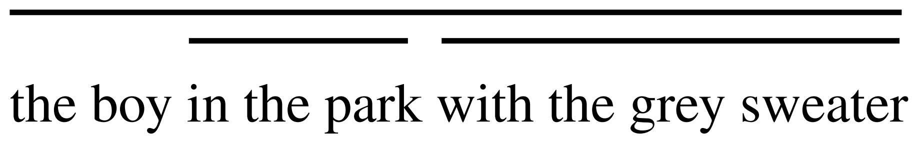
In this course: just syntax and some semantics

Syntax

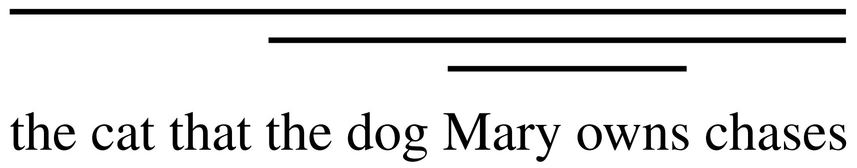
How words *group* together in a language: phrases, sentences, paragraphs, ...



the boy in the park with the red bench



the boy in the park with the grey sweater



the cat that the dog Mary owns chases

Syntax vs. semantics

Part of syntax is understanding how strings of words can be put together and taken apart in a language.

This is not the same thing as meaning.

- Syntactically well-formed sentences need not be semantically well-formed.

  Colourless green ideas sleep furiously.

- Syntactically ill-formed word sequences can sometimes be somewhat meaningful.

  Accident driver car drunk hospital.

Lexicon

The starting point for the syntactic analysis of a language is a *lexicon*. This specifies the word categories of the language:

- article: a, the
- adjective: fat, rich, happy, oldest, orange, . . .
- proper nouns: Mary, John, Toronto, Great Britain, . . .
- common nouns: boy, sweater, park, milk, justice, . . .
- transitive verbs: kick, love, eat, . . .
- intransitive verbs: swim, walk, sleep, . . .
- copula verbs: is, seems, . . .
- prepositions: in, on, from, beside, . . .
- *other word categories*: pronouns, adverbs, interjections, . . .

Words can appear in many categories, for example: set, run, fat

Grammar

A *grammar* of a language is a specification of the various types of well-formed word groups.

These word groups are called grammatical *categories*.

We need to distinguish between

- lexical or *terminal* categories, like article, or transitive verb.
  We write these in lower-case.

- group or *non-terminal* categories, like phrases or sentences.
  We will write these in upper case.

  It is customary to name them using short cryptic abbreviations:

  NP instead of noun phrase

Grammar rules

Usually grammars are specified by a collection of rules (similar to Prolog clauses) describing how each type of word group is formed from other word groups.

A grammar rule will have the following form:

$$\textit{category} \;\rightarrow\; \textit{category}_1 \;\; \textit{category}_2 \;\; \ldots \;\; \textit{category}_n$$

where the category on the left must be a non-terminal, and the categories on the right (if any) can be either terminals or non-terminals.

For example,

```
PP  →  preposition NP
```

says that a prepositional phrase (PP) can be made up of a preposition followed by a noun phrase (NP).

Sample grammar

Here is a grammar for simple English declarative sentences:

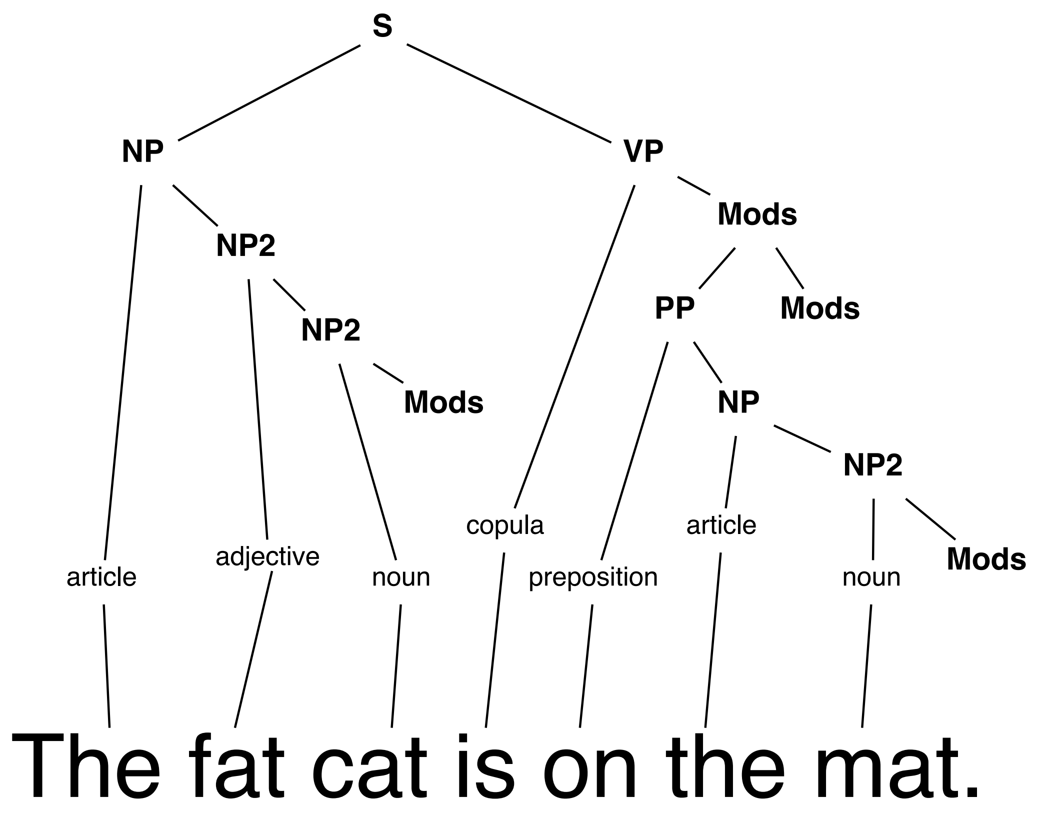
S	$\longrightarrow$	NP VP
VP	$\longrightarrow$	copula_verb Mods
VP	$\longrightarrow$	transitive_verb NP Mods
VP	$\longrightarrow$	intransitive_verb Mods
Mods	$\longrightarrow$	
Mods	$\longrightarrow$	PP Mods
PP	$\longrightarrow$	preposition NP
NP	$\longrightarrow$	proper_noun
NP	$\longrightarrow$	article NP2
NP2	$\longrightarrow$	adjective NP2
NP2	$\longrightarrow$	common_noun Mods

Note: no pronouns, no clauses, . . .   Note also: recursion!

Parse tree

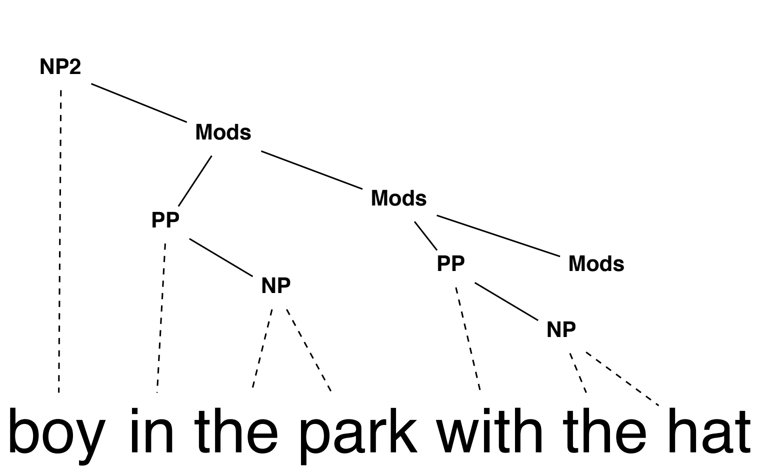
Parsing is the process of taking a sequence of words and determining how they fit into a given grammar.

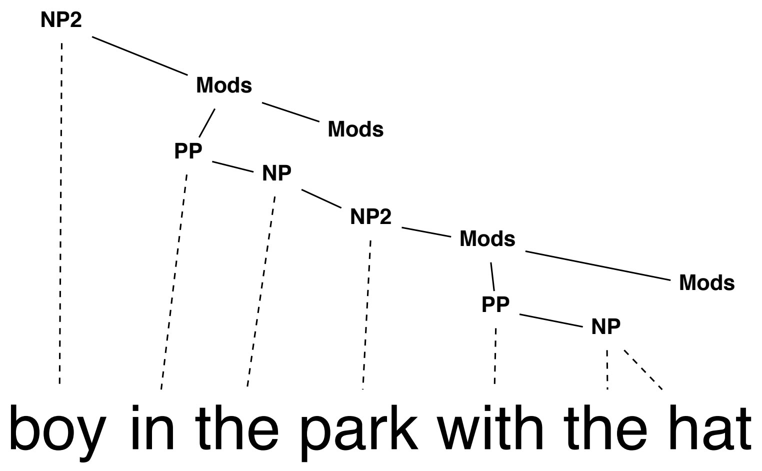
This is done by producing a *parse tree*, with the words at the leaves, and the desired group category at the root.



Ambiguity

A grammar is *ambiguous* if there is a sequence of words with two distinct parse trees.





this parse corresponds to the grouping where “with the hat” modifies “boy”

this parse corresponds to the grouping where “with the hat” modifies “park”

Thinking about the NP is what allows us to deal with the ambiguity!

Our goal

What we will do here, is write a program in Prolog that does simple syntactic and semantic processing of *yes-no questions* only:

- parse the questions according to a given grammar
- determine referents for noun phrases and answer questions (resolving ambiguities as necessary)

For example:

```
?- yes_no('is mary in a park with linda').
Yes

?- yes_no('is the man with the red hat george').
No

?- yes_no('is the small blue hat on the woman beside linda').
Yes

?- yes_no('is a red with a woman hat').        % ungrammatical
No
```

The approach

The Prolog program will have 3 distinct pieces:

1. a world model

   These are clauses that represent the *knowledge of the world* we are interested in: who the people are, where they are, what they are wearing, etc.

   Nothing in the world model is intended to be language specific.

2. a lexicon

   These are clauses that describe the *English words* we will be using in the noun phrases. It also links these words to their meaning in the predicates and constants in the world model.

   Nothing in the lexicon depends on the grammar.

3. a parser / interpreter

   These are clauses which define the *grammar*. It also uses information provided by the lexicon and world model to decide what individual is being referred to.

1. A simple world model

nldb.pl

```
person(john). person(george). person(mary). person(linda).
park(kew_beach). park(queens_park).
tree(tree01). tree(tree02).  tree(tree03).
hat(hat01).   hat(hat02).  hat(hat03).  hat(hat04).

sex(john,male).    sex(george,male).
sex(mary,female).  sex(linda,female).

color(hat01,red).   color(hat02,blue).
color(hat03,red).   color(hat04,blue).

in(john,kew_beach).     in(george,kew_beach).
in(linda,queens_park).  in(mary,queens_park).
in(tree01,queens_park). in(tree02,queens_park).
in(tree03,kew_beach).

beside(mary,linda). beside(linda,mary).

on(hat01,john). on(hat02,mary). on(hat03,linda). on(hat04,george).

size(john,small).    size(george,big).
size(mary,small).    size(linda,small).
size(hat01,small).   size(hat02,small).
size(hat03,big).     size(hat04,big).
size(tree01,big).    size(tree02,small).  size(tree03,small).
```

Building a lexicon

In the lexicon, we need to describe all the words we intend to use in noun phrases.

In addition, for each word, we need to know what constraint (if any) it imposes on the *referent* we are looking for.

For example:

- if the word is “hat”, and the object we are looking for is X, then the query `hat(X)` should succeed
- if the word is “red” and the object we are looking for is X, then the query `colour(X,red)` should succeed

So when we parse a noun phrase like “a red hat” we will end up with the conjunctive query `colour(X,red),hat(X)`.

More on the lexicon

For proper nouns, we use the noun itself (or a lowercase version of it).

For example:

- if the word is “john” and the object we are looking for is Y, then the query `Y=john` should succeed.

For prepositions, there are *two* referents involved.

For example:

- if the word is “on” and the two objects we are looking for are X and Y, then the query `on(X,Y)` should succeed

So when we parse “a red hat on John” we will end up with the conjunctive query `colour(X,red),hat(X),on(X,Y),Y=john`.

2. Example lexicon

lexicon.pl

```
article(a).  article(the).

common_noun(park,X) :- park(X).
common_noun(tree,X) :- tree(X).
common_noun(hat,X) :- hat(X).
common_noun(man,X) :- person(X), sex(X,male).
common_noun(woman,X) :- person(X), sex(X,female).

adjective(big,X) :- size(X,big).
adjective(small,X) :- size(X,small).
adjective(red,X) :- color(X,red).
adjective(blue,X) :- color(X,blue).

preposition(on,X,Y) :- on(X,Y).
preposition(in,X,Y) :- in(X,Y).
preposition(beside,X,Y) :- beside(X,Y).
% The preposition 'with' is flexible in how it is used.
preposition(with,X,Y) :- on(Y,X).       % Y can be on X
preposition(with,X,Y) :- in(Y,X).       % Y can be in X
preposition(with,X,Y) :- beside(Y,X).   % Y can be beside X

% Any word that is not in one of the four categories above.
proper_noun(X,X) :- \+ article(X), \+ adjective(X,_),
                    \+ common_noun(X,_), \+ preposition(X,_,_).
```

The world model vs. the lexicon

In our example, the world model names are similar or equal to lexicon names:

```
common_noun(hat,X) :- hat(X).
adjective(small,X) :- size(X,small).
```

But observe that they have very different purposes:

- the lexicon describes the words we are interested in using (like the *word* “hat”)
- the world model describes the facts of the world we care about (using the *concept* of a hat)

The world model is intended to be language neutral. We could have used

```
common_noun(hat,X) :- concept237(X).
```

Why distinguish world models and lexicons?

In general, we distinguish between the world model and the lexicon for two good reasons:

1. Different words may involve the same concept:
   - synonyms

     ```
     common_noun(cap,X) :- hat(X).
     common_noun(bonnet,X) :- hat(X).
     ```

   - other languages

     ```
     common_noun(chapeau,X) :- hat(X).
     ```

2. Different concepts may involve the same word:

   ```
   common_noun(cap,X) :- hat(X).
   common_noun(cap,X) :- bottle_top(X).
   common_noun(cap,X) :- regulated_maximum(X).
   ```

The parser

For each *non-terminal category* in the grammar, we will have a predicate in the parser.

Each such predicate will take two arguments:

- a *list* of words to be parsed
- an object in the world model

The predicate should hold if the list of words can be used to refer to the object according to the facts in the world model.

For example,

`np([a,woman,in,a,park],linda)` should hold.

`np([a,hat,on,linda],hat02)` should not hold.

Then each grammar rule becomes a clause in Prolog.

3. Example parser (for noun phrases)

np.pl

```
np([Name],X) :- proper_noun(Name,X).
np([Art|Rest],X) :- article(Art), np2(Rest,X).

np2([Adj|Rest],X) :- adjective(Adj,X), np2(Rest,X).
np2([Noun|Rest],X) :- common_noun(Noun,X), mods(Rest,X).

mods([],_).
mods(Words,X) :-
   append(Start,End,Words),    % Break the words into two pieces.
   pp(Start,X),                % The first part is a PP.
   mods(End,X).                % The last part is a Mods again.

pp([Prep|Rest],X) :- preposition(Prep,X,Y), np(Rest,Y).
```

Note how we use append to split a list of words into two pieces.

This will allow us to have `[in,the,park,with,a,red,hat]` break into two smaller lists: `[in,the,park]` and `[with,a,red,hat]`.

Now we are ready to load all three pieces (world model, lexicon, parser) into Prolog and try them out. . . .

Tracing a parse

```
?- np([a,big,tree],X).
 Call: (7) np([a, big, tree], _G322)
 Call: (8) article(a)
 Exit: (8) article(a)
 Call: (8) np2([big, tree], _G322)
 Call: (9) adjective(big, _G322)
 Exit: (9) adjective(big, george)      % George is something big
 Call: (9) np2([tree], george)
  ...  this eventually fails and then ...
 Exit: (9) adjective(big, tree01)      % Tree01 is something big
 Call: (9) np2([tree], tree01)
 Call: (10) adjective(tree, tree01)
 Fail: (10) adjective(tree, tree01)
 Redo: (9) np2([tree], tree01)
 Call: (10) common_noun(tree, tree01)
 Exit: (10) common_noun(tree, tree01)  % Tree01 is a tree
 Call: (10) mods([], tree01)
 Exit: (10) mods([], tree01)
 Exit: (9) np2([tree], tree01)
 Exit: (8) np2([big, tree], tree01)
 Exit: (7) np([a, big, tree], tree01)

X = tree01
```

Using a prepositional phrase

```
?- np([a,woman,beside,mary],W).
 Call: (8) np([a, woman, beside, mary], _G313)  % skip the article here
 Call: (9) np2([woman, beside, mary], _G313)
 Call: (10) adjective(woman, _G313)
 Fail: (10) adjective(woman, _G313)
 Redo: (9) np2([woman, beside, mary], _G313)
 Call: (10) common_noun(woman, _G313)
 Exit: (10) common_noun(woman, mary)      % the first woman it finds
 Call: (10) mods([beside, mary], mary)
    ... this eventually fails and then ...
 Exit: (10) common_noun(woman, linda)     % the second woman
 Call: (10) mods([beside, mary], linda)   % break the words into 2 pieces
 Call: (11) pp([], linda)                 % [] and [beside, mary]: fails!
 Call: (11) pp([beside], linda)           % [beside] and [mary]: fails!
 Call: (11) pp([beside, mary], linda)     % [beside, mary] and []: ok!
 Call: (12) preposition(beside, linda, _L720)
 Exit: (12) preposition(beside, linda, mary)  % note the binary relation
 Call: (12) np([mary], mary)              % this will eventually succeeds
 Call: (11) mods([], linda),              % this succeeds (no constraint)
     ... and eventually ...
 Exit: (8) np([a, woman, beside, mary], linda)

W = linda
```

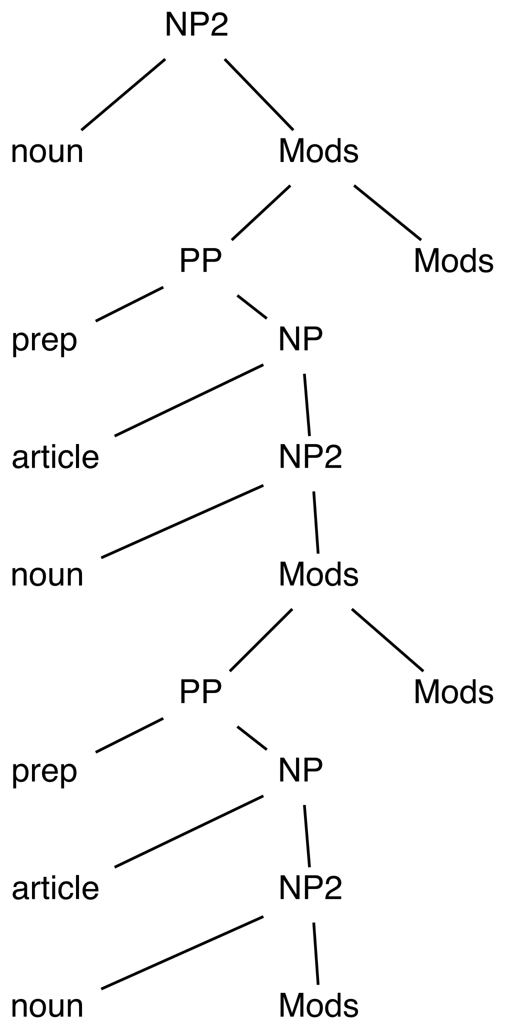
Handling ambiguity

```
Call: (7) np2([man, in, the, park, with, a, tree], _G334)
Call: (8) common_noun(man, _G334)
Exit: (8) common_noun(man, john)
Call: (8) mods([in, the, park, with, a, tree], john)   % need a split
Call: (9) pp([], john)
Call: (9) pp([in], john)
Call: (9) pp([in, the], john)
Call: (9) pp([in, the, park], john)
Exit: (9) pp([in, the, park], john)                    % 1st part: yes
Call: (9) mods([with, a, tree], john)
Fail: (9) mods([with, a, tree], john)                  % 2nd part: no!
Fail: (9) pp([in, the, park], john)
Call: (9) pp([in, the, park, with], john)
Call: (9) pp([in, the, park, with, a], john)
Call: (9) pp([in, the, park, with, a, tree], john)     % this split works
Call: (10) preposition(in, john, _G335)
Exit: (10) preposition(in, john, qbeach)
Call: (10) np([the, park, with, a, tree], qbeach)
Exit: (10) np([the, park, with, a, tree], qbeach)
Exit: (9) pp([in, the, park, with, a, tree], john)
Exit: (8) mods([in, the, park, with, a, tree], john)
Exit: (7) np2([man, in, the, park, with, a, tree], john)
```

Reconstructing a parse tree

a partial trace

- `np2([man,in,the,park,with,a,tree],_)`
- `common_noun(man,_),`
  `mods([in,the,park,with,a,tree],_)`
- `mods([in,the,park,with,a,tree],_)`
- `pp([in,the,park,with,a,tree],_), mods([],_)`
- `preposition(in,_,_),`
  `np([the,park,with,a,tree],_)`
- `np([the,park,with,a,tree],_)`
- `article(the), np2([park,with,a,tree],_)`
- `np2([park,with,a,tree],_)`
- `common_noun(park,_), mods([with,a,tree],_)`
- `mods([with,a,tree],_)`
- `pp([with,a,tree],_), mods([],_)`
- `preposition(with,_,_), np([a,tree],_)`
- `np([a,tree],_)`
- `article(a), np2([tree],_)`
- `np2([tree],_)`
- `common_noun(tree,_), mods([],_)`



More examples

```
?- np([a,man,with,a,big,hat],X).
X = george ;
No

?- np([the,hat,on,george],X).
X = hat04 ;
No

?- np([a,man,in,a,park,with,a,big,tree],X).
No

?- np([a,woman,in,a,park,with,a,big,tree],X).
X = mary ;
X = linda ;
No

?- np([a,woman,in,a,park,with,a,big,red,hat],X).
X = linda ;
No

?- np([a,woman,beside,a,woman,with,a,blue,hat],X).
X = mary ;          % Note: this is not the 'obvious' reading
X = linda ;
No

?-  np([a,woman,with,a,blue,hat,beside,a,woman],X).
X = mary ;
No
```

Fun with parsing

```
?- np([the, Word, on, john], hat01).
Word = hat ;
No

?- np([a, man, with, H], P), np([the, hat, on, P], H).
H = hat01  P = john ;
H = hat04  P = george ;
No

?- L=[_,_,_,_,_], np(L,linda), \+ member(the,L).
L = [a, small, small, small, woman] ;
L = [a, small, woman, in, queens_park] ;
L = [a, small, woman, beside, mary] ;
L = [a, small, woman, with, hat03] ;
L = [a, small, woman, with, mary] ;
L = [a, woman, in, a, park] ;
L = [a, woman, beside, a, woman] ;
L = [a, woman, with, a, hat] ;
L = [a, woman, with, a, woman] ;
No

?- np(L,linda).
L = [linda] ;
ERROR: Out of local stack     % Trying to make 'a small small small ...'
                              % Can this be fixed?
```

Interrogative sentences

To handle some simple interrogative sentences, we can use grammar rules similar to those from before:

- *wh-questions*:

  ```
  WH  →  wh_word copula_verb NP
  ```

  Who is the woman with Linda?
  What is the hat on the man in the park?

  ```
  WH  →  wh_word copula_verb PP
  ```

  What is in Queen's Park?
  Who is beside a man with a small hat?

- *yes-no questions*:

  ```
  YN  →  copula_verb NP NP
  ```

  Is the man with the blue hat John?
  Is Mary the woman beside Linda?

  ```
  YN  →  copula_verb NP PP
  ```

  Is John beside a woman with a blue hat?
  Is the big red hat on George?

From strings to lists of words

To parse yes-no questions, we use a parsing program which (as we shall see) is very similar to the one for noun phrases.

However, for convenience, we would also like to be able to use a *string* like `'is john beside mary'` and convert this into a list of words for parsing.

The predicate split_words($string$, $list$) breaks a quoted string into a list of words. (It is included in the file **wordUtils.pl**.)

```
?- split_words('a man called horse',X).
X = [a, man, called, horse]
Yes
```

The `wordUtils.pl` file also includes utilities for punctuation and upper and lower case, but these need not concern us here.

> A fancier program would convert a string like `'Is John beside Mary?'` into the list of words `[is,john,beside,mary]`.

Finally: the top-level predicate

We want a predicate `yn`(*words*) that holds when *words* is a list of words forming a yes-no question whose answer is “yes”.

yesno.pl

```
yes_no(String) :-
   split_words(String,Words),  % Get the list of words.
   yn(Words).                  % Use yn on the words.

yn([Verb|Rest]) :-
   Verb=is,                    % The first word must be "is".
   append(W1,W2,Rest),         % Break the rest into two parts.
   np(W1,Ref),                 % The first part must be an NP.
   np_or_pp(W2,Ref).           % The second part must be an NP or a PP.

np_or_pp(W,Ref) :- np(W,Ref).
np_or_pp(W,Ref) :- pp(W,Ref).
```

Note: with this version of the parser, we cannot distinguish between a *failure* (a question that is ill-formed or where no referent can be found) and a legitimate question whose answer is “no”.

What about declarative sentences?

Declarative sentences require a different form of semantics.

Consider: "John is in the park with the big tree."

- After we have determined that the two referents in question are `john` and `queens_park`, we do not want to simply *test* that `in(john,queens_park)` holds.
- Most likely, what is intended is that we should treat this as new information, and *add* it as a new fact to our world model for later use.

This raises two problems:

- How do we get a program to add facts to a Prolog world model?
- Our lexicon is geared to finding referents only. For example, for the word "in" we generate a *query* of the form `in(X,Y)`.

Dynamic predicates

Prolog allows the clauses associated with certain predicates to be changed by the program itself.

These are called *dynamic* predicates.

For example, suppose we have a predicate `my_pred`(x,y,z) that takes three arguments.

To make `my_pred` dynamic, we include the declaration

```
:- dynamic my_pred/3.
```

in the program file before the predicate is used.

We can then have clauses in the file that define `my_pred` as usual, but we also get to use two special operations: `assert` and `retract`.

Assert and retract

There are two special queries that can be performed on atoms whose predicates are dynamic:

- `assert`(*atom*)
  This query always succeeds, and has the effect of adding the atom as a fact to Prolog's knowledge base.

- `retract`(*atom*)
  This query has the effect of removing the first fact in Prolog's knowledge base that matches the atom. It fails if there is no match.

This means we can write Prolog programs like this

```
get_married(X) :-
   retract(single(X)),    % X is no longer single
   assert(married(X)).    % X is now married
```

which could be used to keep a model of a changing world up to date.

A new treatment for prepositions

Using `assert` and `retract`, we consider handling the prepositions in our lexicon in a second way that is better suited to declarative sentences.

In addition to the clauses we had before, like

```
preposition(on,X,Y) :- on(X,Y).
preposition(in,X,Y) :- in(X,Y).
preposition(beside,X,Y) :- beside(X,Y).
```

we also include in our lexicon

```
add_for_preposition(on,X,Y) :- assert(on(X,Y)).
add_for_preposition(in,X,Y) :- assert(in(X,Y)).
add_for_preposition(beside,X,Y) :- assert(beside(X,Y)), assert(beside(Y,X)).
```

The idea is that `add_for_preposition(P,X,Y)` tells us how we should change our world model if we find out that the object `X` is in the relation denoted by preposition `P` to object `Y`.

Handling simple declarative sentences

With this new treatment of prepositional phrases, we are now ready to handle simple declarative sentences as additions to the world model.

Assume we want to handle declarative sentences like these:

SD $\rightarrow$ NP copula_verb PP

John is in the park with the big tree.

The man with the red hat is beside George.

declarative.pl

```
simple_declarative(String) :-
   split_words(String,Words),          % Get words from string.
   sd(Words).                          % Use sd on the words.

sd(Words) :-
   append(NP1,[is,Prep|NP2],Words),    % Split words.
   np(NP1,X),                          % Find referent for first NP.
   np(NP2,Y),                          % Find referent for second NP.
   add_for_preposition(Prep,X,Y).      % Add new atom to database.
```

Non-referential noun phrases

This treatment of declarative sentences only deals with the simplest form of noun phrases.

One place where it goes wrong is when we use noun phrases that are not intended to refer to anything.

For example, consider

> Jane is sitting beside a man without a hat.

We should not go looking in our world model for a hat that stands in some relation to the man beside Jane!

A noun phrase can be ambiguous as to whether it is referential or not.:

> John wants to marry a rich lawyer and retire early.

In one reading, no referent is intended for the noun phrase.

Less simple declarative sentences

More complex declarative sentences require us to consider knowledge bases that go well beyond what we have been representing in Prolog.

Consider for example, the sentence

> The councillors refused a permit to the demonstrators because they feared violence.

After we have done all the syntactic and semantic analysis (including deciding who the "they" is), we would want to add a "fact" like this:

> There were two events, e_1 and e_2 that took place, involving two groups of people, z_1 and z_2, and where e_1 was a cause of e_2. In addition,
>
> – z_1 is a group of councillors (who are elected representatives)
>
> – z_2 is a group of demonstrators (participants in a demonstration)
>
> – e_1 is the members of z_1 fearing that some event e_3 will take place
>
> – e_2 is z_1 deciding as a group to prevent some event e_4 from taking place
>
> *etc.*

What we have accomplished

We have only scratched the surface of the kinds of processing necessary to make sense of a natural language.

Some of what we skipped over can be readily handled by simple changes to the program we have been considering.

example: number agreement between subjects and verbs,
tenses of verbs

Dealing with complex sentences is still beyond the state of the art.

example: *anaphora*, as in resolving the word “they” in the
“they feared violence” example sentence.

Nonetheless, the idea of interpreting a sentence as a query or an update against a background knowledge base, as we have done here, is at the root of most AI natural language systems.

9

Case Study: Planning Courses of Action

Planning

One of the abilities that people have, and an ability that we are especially proud of, is the ability to *make plans*:

> invent a sequence of actions to achieve a goal

In a sense, this is the core of AI since it deals with generating *intelligent behaviour*.

In some cases, the planning problems we tackle are natural ones

- getting the keys out of a locked car
- obtaining food for lunch

In other cases, we solve artificial problems just for fun

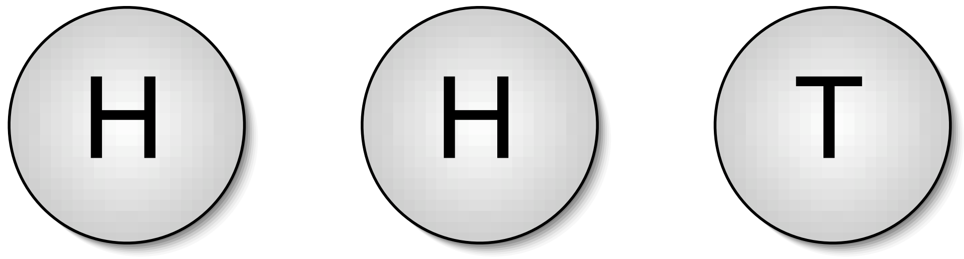
- Rubik's cube
- word problems

Here, we will examine a class of planning problems that can be solved using a single general method.

Problem 1: Three coins

Here is a first example:

Given 3 coins arranged as below, make them all the same (that is, either all heads or all tails) using exactly 3 moves.



By a *move* is meant turning one of the coins over (so that a head becomes a tail, or a tail becomes a head)

Problem 1: A solution

There are many solutions:

- (1) flip middle, (2) flip right, (3) flip middle
- (1) flip left, (2) flip left, (3) flip right *etc.*

General properties:

- all of them end up with HHH

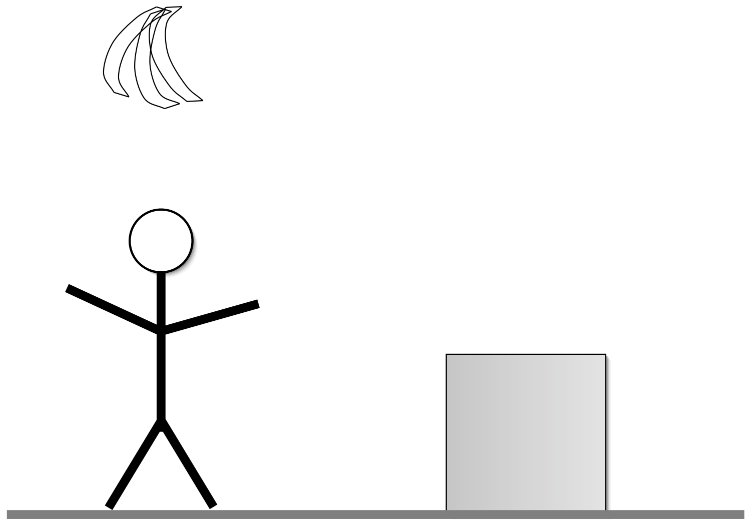
  why?

  - after 1 move: have even number of T's
  - after 2 moves: have odd number of T's
  - after 3 moves: have even number of T's
- all of them move the right coin 1 or 3 times
- those that move it 1 time, move another coin twice

Problem 2: Monkey and bananas

A monkey is in a room where a bunch of bananas is hanging from the ceiling, too high to reach. In the corner of the room is a box, which is not under the bananas. The box is sturdy enough to support the monkey if he climbs on it, and light enough so that he can move it easily. If the box is under the bananas, and the monkey is on the box, he will be able to reach the bananas.

How can the monkey get the bananas?



The monkey’s moves

What can the monkey do?

- go to anywhere in the room

  provided that the monkey is not on the box

- climb on the box

  provided the monkey is at the box

- climb off the box

  provided the monkey is on the box

- push the box anywhere

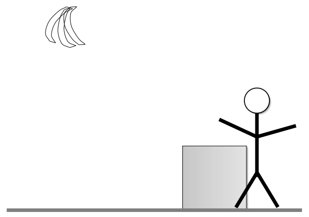
  provided the monkey is at the box, but not on it

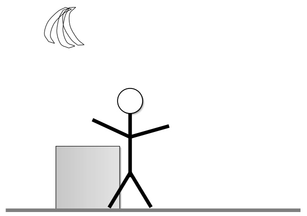
- grab the bananas

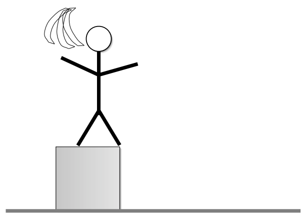
  provided the monkey can reach the bananas

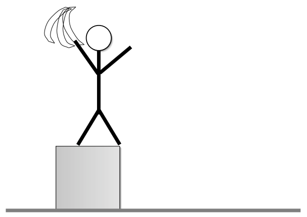
Problem 2: Solution

Here is what the monkey needs to do:


go to where the box is


push it under the bananas


climb on the box


grab the bananas

States and operators

In general, planning problems of this form can be characterized by

states: snapshots of the world that we pass through

- what sides the 3 coins are showing
- location of the monkey, bananas, and box

operators **or** ***moves*** **or** ***actions***: ways of moving from state to state

- flipping one of the coins
- climbing on a box

initial state: the starting state(s) for a problem

- HHT
- as per the first picture of monkey and bananas

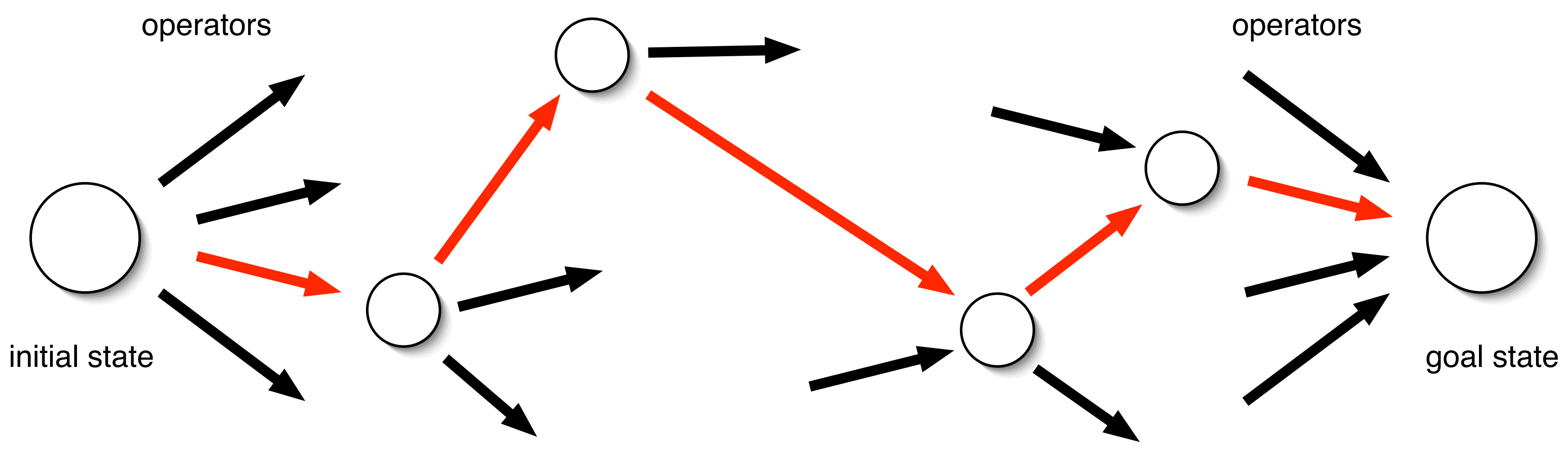
goal state: the desired final state(s)

- HHH or TTT
- any state where the monkey has the bananas

Planning problems

The problem is always the same:

> find a way of going from an initial state to a goal state by applying a sequence of operators.



The path in red is a *plan*: a sequence of operators that takes you from an initial state to a goal state.

Using Prolog

We would like to solve planning problems of this type using Prolog:

- we tell it the initial state, the goal state, the operators;
- it should tell us a way of getting from the initial state to the goal state using the operators.

How?

It will have to *think* about the problem:

- I start in an initial state, and I want to end up in a goal state.
- If I am in state S and I do move M, that would put me into a new state S'. If I'm in S' and I do M', that will take me to S''. *etc.*

Along the way, it will need to keep track of the moves it needs to make.

Getting from one state to another

We need to be clear about what it takes to go from one state S_1 to another state S_2 using a sequence of moves L.

We say that S_2 is *reachable* from S_1 using L.

There are only two general principles for reachability:

1. S is reachable from S (trivially) using no moves;
2. if there is a move M_0 that goes from state S_1 to S_2, and S_3 is reachable from S_2 using moves $\langle M_1, \ldots, M_n \rangle$, then S_3 is also reachable from S_1 using moves $\langle M_0, M_1, \ldots, M_n \rangle$.

$$\text{If } \; S_1 \xrightarrow{M_0} S_2 \overset{\langle M_1, \ldots, M_n \rangle}{\rightarrow \ldots \rightarrow} S_3 \quad \text{then } \; S_1 \overset{\langle M_0, M_1, \ldots, M_n \rangle}{\rightarrow \ldots \rightarrow} S_3$$

We can encode this directly in Prolog using *lists* of moves.

A general Prolog planner

To solve a problem, we try to reach a goal state from an initial state:

plan.pl

```
% This general planner needs the predicates below to be defined:
%     - legal_move(BeforeState,Move,AfterState)
%     - initial_state(State)
%     - goal_state(State)

% plan(L): L is a list of moves from the initial state to a goal state.
plan(L) :- initial_state(I), goal_state(G), reachable(I,L,G).

% reachable(S1,L,S2): S2 is reachable from S1 using moves L.
reachable(S,[],S).
reachable(S1,[M|L],S3) :- legal_move(S1,M,S2), reachable(S2,L,S3).
```

This is a *general* planner. For each specific problem, we need to specify the clauses for `initial_state`, `goal_state`, and `legal_move`.

The 3-coin problem in Prolog

We can represent the states of this problem as lists $[x,y,z]$, where the x, y and z are either `h` or `t` (8 possibilities).

Here is the full program:

coins.pl

```
% The three-coins problem formulated for the general planner.

initial_state([h,h,t]).

goal_state([h,h,h]).
goal_state([t,t,t]).

% The three possible moves. Each changes one of the coins.
legal_move([X,Y,Z],flip_left,[X1,Y,Z]) :- opposite(X,X1).
legal_move([X,Y,Z],flip_middle,[X,Y1,Z]) :- opposite(Y,Y1).
legal_move([X,Y,Z],flip_right,[X,Y,Z1]) :- opposite(Z,Z1).

opposite(h,t).                   % Flipping a head gives a tail.
opposite(t,h).                   % Flipping a tail gives a head.
```

To use this program, we load it together with the general planner.

Running the planner

```
?- plan([M1,M2,M3]).    % we seek a plan with 3 moves

M1 = flip_left
M2 = flip_left
M3 = flip_right ;

M1 = flip_left
M2 = flip_right
M3 = flip_left ;

M1 = flip_middle
M2 = flip_middle
M3 = flip_right ;

M1 = flip_middle
M2 = flip_right
M3 = flip_middle ;

M1 = flip_right
M2 = flip_left
M3 = flip_left    % there are 2 more solutions, but we stop here

Yes
```

Monkey and banana states in Prolog

The states in the monkey and bananas are more complex.

> We need to specify the location of the bananas, the monkey and the box, whether or not the monkey is on the box, and whether or not the monkey has the bananas.

We can use a 5-element list: $[b, m, l, o, h]$, where

- b, m and l are the locations of the bananas, the monkey and the box respectively;
- o is `y` or `n` according to whether the monkey is on the box; and
- h is `y` or `n` according to whether the monkey has the bananas.

We name the initial locations of the bananas, monkey, and box as `loc1`, `loc2`, and `loc3`. Nothing else needs a name.

A digression: Atoms as terms

Before look at operators for the monkey and bananas, we need to consider another feature of Prolog.

So far, atoms are of the form *predicate(term, . . . , term)*, where a term is a constant, a number, a variable or a list whose elements are terms.

However, a term may also be another *atom*.

```
?- X=p, Y=q(a).     % X is a constant but Y is an atom
X=p, Y=q(a)

?- Y=p(a), Z=[a,Y,b].          % atoms can be in lists
Y=p(a), Z=[a,p(a),b]
```

Up until now, it was useful to make a clear separation in our understanding of Prolog between terms and atoms.

But now we can see that a constant is just a special case of an atom where the predicate takes no arguments!

Monkey and banana operators in Prolog

We will use the following operators:

`climb_on` (climbing on the box)	a constant
`climb_off` (climbing off the box)	a constant
`grab` (grabbing the bananas)	a constant
`go(X)` (going to location `X`)	an atom!
`push(X)` (pushing the box to location `X`)	an atom!

We need to write clauses for `legal_move` for each of these operators.

These clauses should say how a *before state* is changed to an *after state* by the operator in question.

The clauses should *fail* if the operator in question cannot be used in the before state. In this case, the move would not be a legal one.

Pushing a box

Consider the `push(X)` operator.

This operator can only be used in states where the monkey is at the same location as the box, but not on the box.

After the operators is used, both the monkey and the box will be at location `X`, and the rest remains unchanged.

We can write this as follows:

```
%               before     operator      after
legal_move( [B,M,L,O,H], push(X), [B1,M1,L1,O1,H1]) :-
    M=L, O=n,                         % provisos on the old state
    B1=B, M1=X, L1=X, O1=O, H1=H.     % values of the new state
```

This can expressed more succinctly as:

```
legal_move([B,M,M,n,H],push(X),[B,X,X,n,H]).
```

The other operators will be similar.

Monkey and bananas as planning

monkey.pl

```
% This is the monkey and bananas as a planning problem.

% The bananas, monkey, and box are at different locations.
% The monkey is not on the box and has no bananas.
initial_state([loc1,loc2,loc3,n,n]).

% The goal is any state where the monkey has the bananas.
goal_state([_,_,_,_,y]).

% Climbing on the box causes the monkey to be on the box.
legal_move([B,M,M,n,H],climb_on,[B,M,M,y,H]).
% Climbing off the box causes the monkey to be off the box.
legal_move([B,M,M,y,H],climb_off,[B,M,M,n,H]).
% Grabbing the bananas causes the monkey to have the bananas.
legal_move([B,B,B,y,n],grab,[B,B,B,y,y]).
% Pushing the box changes where the monkey and the box are.
legal_move([B,M,M,n,H],push(X),[B,X,X,n,H]).
% Going to a location changes where the monkey is.
legal_move([B,_,L,n,H],go(X),[B,X,L,n,H]).
```

Again we load this program together with the general planner.

Getting the bananas

```
?- plan([M1,M2,M3,M4]).       % Is there a 4 step plan?

M1 = go(loc3)      % where the box is
M2 = push(loc1)    % where the bananas are
M3 = climb_on
M4 = grab ;        % any other plans?

No

?- plan([M1,M2,M3]).          % Is there a 3 step plan?

No

?- plan([M1,M2,M3,M4,M5]).    % Is there a 5 step plan?

M1 = go(_G325)    % first go to any location
M2 = go(loc3)     % then go to the box
M3 = push(loc1)
M4 = climb_on
M5 = grab         % and there are other plans

?- plan([M1,M2,M3,M4,M5]), \+ M1 = go(_).

No
```

Unbounded plans

We cannot ask for a plan without first restricting its length.

```
?- plan(L).
ERROR: Out of local stack
```

What is the problem?

The general planner looks for a path from an initial state S_0 to a goal state S_g. By tracing, we observe the following:

1. the first action it considers is a `go` action from S_0 to a state S_1; it then needs to find a path from S_1 to S_g;
2. the first action it considers is a `go` action from S_1 to a state S_2; it then needs to find a path from S_2 to S_g; *etc.*

So the planner never ends up considering any other actions.

Bounding plans

To avoid problems with unbounded plans, we need to ensure that we only work on `plan(L)` when we know the length of `L`.

This suggests that we should first look for 0-step plans, then 1-step plans, then 2-step plans, and so on:

bplan.pl

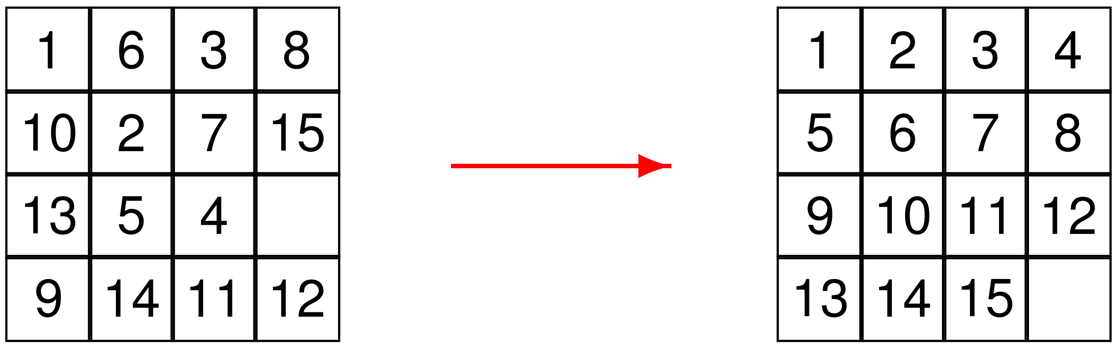
```
% This looks for plans, short ones first, using the plan predicate.
% bplan(L) holds if L is a plan.
bplan(L) :- tryplan([],L).

% tryplan(X,L): L is a plan and has X as its final elements.
tryplan(L,L) :- plan(L).
tryplan(X,L) :- tryplan([_|X],L).
```

```
?- bplan(L).
L = [go(loc3), push(loc1), climb_on, grab]
```

Problem 3: the 15-puzzle

The 15-puzzle consists of 15 numbered tiles located in a 4×4 grid.

The object of the puzzle is to move the tiles within the grid so that each tile ends up at its correct location, as shown:



initial state goal state

Each move involves shifting a tile up, down, left, or right, assuming the vacant spot is adjacent.

States for the 15-puzzle

We can represent a state of the puzzle using a 16-element list

$$[p_1,\ p_2,\ p_3,\ p_4,\ p_5,\ p_6,\ p_7,\ p_8,\ p_9,\ p_{10},\ p_{11},\ p_{12},\ p_{13},\ p_{14},\ p_{15},\ p_{16}]$$

where each p_i is a number from 0 to 15 indicating which tile is in position i (and where 0 means position i is empty).

From the diagram on the previous page, we have

- initial state: [1, 6, 3, 8, 10, 2, 7, 15, 13, 5, 4, 0, 9, 14, 11, 12]
- goal state: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 0]

Operators for the 15-puzzle

We will use the operators `up(X)`, `down(X)`, `left(X)`, `right(X)` where `X` is a tile.

The easiest way to write `legal_move` for `up` is to write a clause for each location of the empty square where a tile can go up. For example:

```
legal_move(S1,up(X),S2) :-
    S1=[A,B,0,D, E,F,X,H, I,J,K,L, M,N,O,P],
    S2=[A,B,X,D, E,F,0,H, I,J,K,L, M,N,O,P].
```

There will be 12 such locations for each of the 4 operators!

But one simplification we can use is that `up` and `down` are *inverses* (as are the `left` and `right` operators). So once we have written all the clauses for `up`, we can write a single clause for `down`:

```
legal_move(S1,down(X),S2) :- legal_move(S2,up(X),S1).
```

A simplified version

Here is the full program for a 2×3 version of the puzzle:

puzzle2x3.pl

```
% This is a 2x3 version of the 15 puzzle.                    %%%%%%%%%
initial_state([0,1,5,4,3,2]).   %---------------------->    %    1 5 %
goal_state([1,2,3,4,5,0]).                                   % 4 3 2 %
legal_move([0,B,C, X,E,F],up(X),[X,B,C, 0,E,F]).            %%%%%%%%%
legal_move([A,0,C, D,X,F],up(X),[A,X,C, D,0,F]).
legal_move([A,B,0, D,E,X],up(X),[A,B,X, D,E,0]).
legal_move(S1,down(X),S2) :- legal_move(S2,up(X),S1).

legal_move([0,X,C, D,E,F],left(X),[X,0,C, D,E,F]).
legal_move([A,0,X, D,E,F],left(X),[A,X,0, D,E,F]).
legal_move([A,B,C, 0,X,F],left(X),[A,B,C, X,0,F]).
legal_move([A,B,C, D,0,X],left(X),[A,B,C, D,X,0]).
legal_move(S1,right(X),S2) :- legal_move(S2,left(X),S1).
```

```
?- bplan(L).
L = [left(1), up(3), left(2), down(5), right(3), up(2), left(5)]
```

So what's the problem?

Is the rest of planning just more of the same?

No.   There are two significant problems that have stopped researchers from applying the techniques we have seen to larger more realistic problems:

1. a *reasoning* problem

   How do we deal with the extremely large search spaces that come up in realistic planning problems?

2. a *representation* problem

   How do we represent the states of the world needed for realistic planning problems?

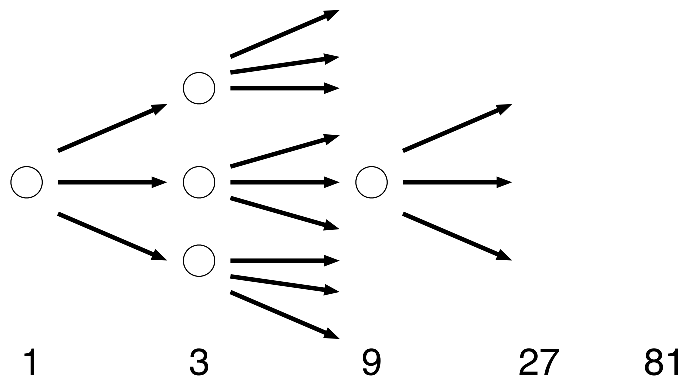
We will now turn our attention to these two problems.

1. The planning search problem

As we search for a path from an initial state to a goal state, we only consider legal moves.

Nonetheless, the number of states we end up considering in general grows *exponentially* with the length of the path.

For example, imagine that there are exactly 3 legal moves possible from any state. (In the 15-puzzle, there are states with 4 legal moves.)



To find a path with 20 moves, we might consider 10^9 states.

To find a path with 30 moves, we might consider 10^{14} states.

To find a path with 40 moves, we might consider 10^{19} states.

Planning search strategy 1

Many problems have a natural notion of subproblem that can be solved *independently* and then combined.

How does this help?

Suppose a problem can be partitioned into subproblems A and B such that

A can be solved in 15 moves
B can be solved in 10 moves

so that 25 moves are sufficient overall.

Solve each subproblem: $3^{15} + 3^{10} \approx 10^7$ states, at worst

But solve the problem directly: $3^{25} \approx 10^{12}$ states, at worst!

Decomposing a problem into smaller subproblems allows us to keep the length of the paths much smaller.

Decomposing the 15 puzzle

The 15-puzzle has a natural decomposition into subproblems:

1	2	3	4
5	–	–	–
9	–	–	–
13	–	–	–

first goal

6	7	8
10	–	–
14	–	–

second goal

11	12
15	

final goal

Similarly, the first goal above breaks down into the goal of getting the first row into place, and then (without moving the first row) getting the `5`, `9`, and `13` into place, and so on.

Depth-first search

Another search problem we have is that we look for a path (of length n) to the goal as follows:

1. for a given current state S, choose a legal move: get a new S'
2. check to see if there is a path (of length $n-1$) from S' to a goal state
3. *if that fails*, choose another legal move: get S''

Imagine that there is a path from S'' to a goal state.

We may end up searching the *entire tree below* S' in step (2) above before even looking at what is below S''.

This is called *depth-first search*.

Best-first search

A better strategy is the following:

1. let L be the list containing just S_0.
2. select from L the state S that is *closest to the goal*
3. if S is a goal state, then exit with success (and return the plan)
4. otherwise, replace S in L by all the states S' you can get to in one move (and note the action that takes you from S to S')
5. go to step (2)

This is called *best-first search*.

It requires us to be able to determine the distance to the goal.

Estimating the distance to the goal

Many problems allow us to calculate quickly an estimate of how far a state is from the goal (to be used with best-first search).

For example, in the 15-puzzle, we can use what is called the “Manhattan distance” to the goal

–	–	–	–
–	–	–	–
–	–	–	6
–	–	–	–

dist for tile 6 is 3

the sum over each tile of the minimum number of vertical and horizontal moves needed to move the tile from where it is now to its final goal position.

This is easy to calculate and is a *lower bound* on the actual number of moves that will be needed to get to the goal.

2. The planning representation problem

For artificial puzzles like the kind we have considered, the representation of a state as an explicit list works fine.

But as soon as we consider realistic domains, it becomes very awkward to have to deal with a list of *all* the relevant aspects of a state.

$$[\,aspect_1,\ aspect_2,\ aspect_3,\ \ldots,\ aspect_{10000}\,]$$

Each operator will only affect a very small number of these aspects; the rest will have to be copied unchanged in the new state.

> pushing a box under the bananas does not affect the position of the bananas, the color of the box, what objects are in the room, which doors are open, . . . , the students enrolled at the University, . . . , the current Prime Minister of Canada, . . . , the price of tea in China, . . .

A representation strategy

The purpose of a state in planning is to track various aspects as they are affected by actions

- the `climb_on` action changes whether or not the monkey is on the box
- the `push(X)` action changes the location of the monkey and of the box

But another possibility is to simply keep track of all the actions that have been performed and then *calculate* the aspects as needed.

For example, suppose we have a list of actions that have been performed so far, and we want to calculate the location of the box:

- if there is no `push` action in the list, then the box is where it was initially
- if the most recent `push` action in the list is `push(loc3)`, then the location of the box is `loc3` (and it does not matter what the earlier actions were)

Situations

A list of actions used to represent a state is called a *situation*.

The actions appear in the list *most recent first*.

`[]`	the *initial situation*, before any actions
[a]	the situation where a single action a has been performed
[b, a, a]	the situation where action a was performed twice, and then action b was performed

Note that situation [a, b] is not the same as [b, a].
The order can make a difference.

In Prolog terms, performing an action `A` in situation `S` always takes us to the new situation `[A|S]`.

Fluents

The various aspects of a state we care about are now represented as predicates that take a situation as a final argument.

We call these predicates *fluents*.

For the monkey and bananas problem, we can use three fluents:

- `on_box(S)` holds when the monkey is on the box in situation `S`;
- `has_bananas(S)` holds when the monkey has the bananas in situation `S`;
- `location(X,L,S)` holds when location of `X` is `L` in situation `S`;

We will need to write clauses for each fluent that characterize when the fluent holds in terms of the situation, *i.e.*, the actions performed so far.

Defining fluents

The easiest way to characterize a fluent is to do it in two steps:

1. state what fluents hold in the initial situation `[]`

```
% on_box([]) is false.  Nothing to say
% has_bananas([]) is false.  Nothing to say
location(box,loc1,[]).
location(monkey,loc2,[]).
location(bananas,loc3,[]).
```

   These are called the *initial-state axioms* for the fluent.

2. state what fluents hold in any non-initial situation `[A|S]`

   This involves cases depending on whether the action `A`

   - makes the fluent true
   - makes the fluent false
   - leaves the fluent unchanged from what it was in `S`

Climbing on the box and getting the bananas

For the fluent `on_box`, the action `climb_on` makes it true, `climb_off` makes it false, and any other action leaves it unchanged.

In other words: the monkey is on the box if it just did a climb on action *or* it was already on the box and did *not* just do a climb off action.

```
on_box([climb_on|_]).   % the most recent action was climb_on
on_box([A|S]) :-
       on_box(S),       % the monkey was already on the box
    \+ A = climb_off.   % the most recent action was not a climb_off
```

Similarly, the monkey has the bananas if it just did a grab action *or* it already had the bananas. (There is no action to drop the bananas.)

```
has_bananas([grab|_]).  % the most recent action was grab
has_bananas([_|S]) :-
       has_bananas(S).  % the monkey already had the bananas in S
```

Clauses like these are called the *successor-state axiom* for the fluent.

Changing locations

The successor-state axiom for `location` is best handled by considering each object separately.

The easiest case is the bananas: their location after doing an action is the same as it was before.

```
location(bananas,L,[_|S]) :- location(bananas,L,S).
```

For the monkey, the location stays the same unless the monkey does a go or a push action.

```
location(monkey,L,[go(L)|_]).
location(monkey,L,[push(L)|_]).
location(monkey,L,[A|S]) :-
   \+ A = go(_),
   \+ A = push(_),
   location(monkey,L,S).
```

The location of the box is handled similarly.

Preconditions

The clauses so far tell us the effect of actions, for example, that the monkey will have the bananas after grabbing them.

They do not tell us that the monkey has to be on the box in the right location to grab the bananas.

To state these preconditions, we use a special predicate `poss(A,S)` that should hold when it is possible to perform action `A` in situation `S`.

```
poss(climb_off,S) :- on_box(S).

poss(climb_on,S) :-
   location(monkey,L,S), location(box,L,S), \+ on_box(S).

poss(grab,S) :- location(monkey,L,S), location(bananas,L,S), on_box(S).

poss(go(X),S) :- \+ on_box(S).

poss(push(X),S) :-
   location(monkey,L,S), location(box,L,S), \+ on_box(S).
```

These are called the *precondition axioms* for the action.

Planning with situations and fluents

We would like to use the same program `plan` as before.

Recall that we need to specify the predicates `initial_state`, `legal_move`, and `goal_state`.

Using situations and fluents, we proceed as follows:

1. One clause for the initial state:

   ```
   initial_state([]).
   ```

2. One clause for the legal moves:

   ```
   legal_move(S,A,[A|S]) :- poss(A,S).
   ```

3. One or more clauses for the goal state in terms of fluents.

   For example,

   ```
   goal_state(S) :- has_bananas(S).
   ```

With these, `plan` and `bplan` will work just as before.

Why is this good?

Although much more verbose than the previous representation that used explicit states, this version has one significant advantage:

> we are able to describe the world incrementally without having to know in advance all the possible actions and fluents

Consider again, for example, the fluent `on_box`. Once we decide what actions change this fluent, we are done:

```
on_box([climb_on|_]).   % the most recent action was climb_on
on_box([A|S]) :-
    \+ A = climb_off,   % the most recent action was not climb_off
    on_box(S).          % the monkey was already on the box
```

Even if we later add a new fluent like `box_colour` and an action that changes it like `paint_box`, we can still use the axiom above.

This has the effect of making a very complex state with large numbers of properties conceptually manageable.

Other complications

In reasoning about action and change, a number of other complications need to be dealt with such as:

- the durations of actions
  - filling a bathtub *vs.* turning on a light
  - planning needs to be sensitive to temporal constraints
- exogenous actions
  - performed by an other agents (or nature)
- sensing actions
  - some actions, like reading a thermometer, are not used to change the world, but only to change what is *known* about the world

Handling these properly in a planning context remains a current very active topic of AI research.

10

Case Study: Playing Strategic Games

Game Playing

As we said at the end of the planning section, one complication is when there are *other agents* that interfere with your ability to attain the goal.

In a sense, this is what is involved in playing a game like Chess.

In this section, we will look at strategies for restricted sorts of games:

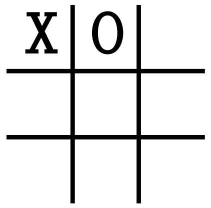
- discrete move and turn-taking (vs. many video games)
- deterministic (vs. Backgammon)
- perfect information (vs. Clue)
- two person (vs. Solitaire, Poker)
- zero sum (vs. Diplomacy)

Examples: Tic-tac-toe, Othello, Checkers, Chess, Go

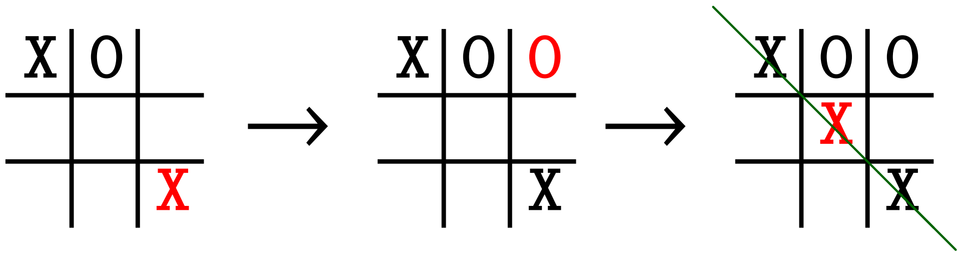
Games as problems

Consider the problem of finding an appropriate *next move*:

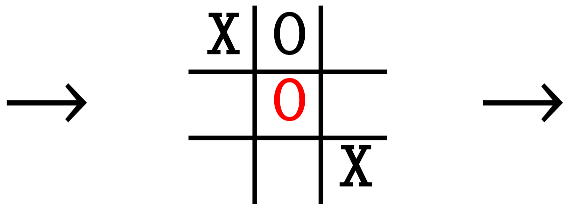
Player X to play and win
(after at most 3 moves)



Faulty reasoning:



Because:



eventually ties

Features of games

1. do not need to find a list of moves from an initial state to a final state

   I must find the best *next move*, wait for the opponent to make a move, and then find the best next move again, and so on

2. the best next move must take into account the responses available to the opponent

   and these responses will take into account the possible counter-responses, which will take into account the counter-counter-responses, and so on

3. the best move should not rely on the opponent making a mistake

   ideally, I will find a move such that *even if you play your best*, I can still find a move such that even if . . . I will still win.

How a game is defined

By analogy with planning problems, we can think of a game as follows:

- There are two *players*.
  - one of the players moves first.
- There is a space of *states*.
  - *initial state*, at the start of the game.
  - *final state*, one of the player wins, or the game is a tie.
- In any state, there is a set of *legal moves*.
  - moves go from one game state to another game state.
  - throughout the game, players take turns at *choosing* one of moves that is legal in the current state.

How we will define each game in Prolog

With planning, we had to specify the `initial_state`, `goal_state`, and `legal_move` predicates for each problem.

With game playing, we specify the following 4 predicates for each game:

- `player(`*player*`)`

  player is one of the two players

- `initial_state(`*state*`,` *player*`)`

  the game starts in *state* and *player* must move first

- `legal_move(`*old_state*`,` *player*`,` *move*`,` *new_state*`)`

  when it is the turn of *player* in *old_state*, then *move* is legal and goes to *new_state*, and the other player's turn

- `game_over(`*state*`,` *player*`,` *winner*`)`

  when it is the turn of *player* in *state*, then *winner* wins. The winner is either one of the players or the constant `neither` for a tie.

A first game: Race to 21

The rules:

> There are 21 chips on the table. Players take turns removing either 1 or 2 chips. The player removing the last chip wins.

States: a number between 0 and 21 = total chips left on the table

- initial state: 21
- final state: 0
  The player whose turn it is to move loses. (There is never a tie.)

Moves: either 1 or 2.

- the new state = the old state – the move.
  (It must not be negative.)

Race to 21 in Prolog

raceto21.pl

```
%  This is the Prolog definition of the game Race to 21.

player(max).  player(min).          % The two players.

initial_state(21,max).              % Player Max goes first.

game_over(0,max,min).               % If it is Max's turn to play,
game_over(0,min,max).               % then Min wins, and vice versa.

legal_move(OldState,_,Move,NewState) :-
   small_number(Move),              % A move is a small number.
   OldState >= Move,
   NewState is OldState - Move.

small_number(1).                    % A small number is either
small_number(2).                    % the number 1 or 2.
```

This defines the game.

But how do we play it?   And play it well?

Game trees

A player can decide on a best move by considering all possible moves, all possible counter-moves, all possible counter-counter-moves, *etc.*

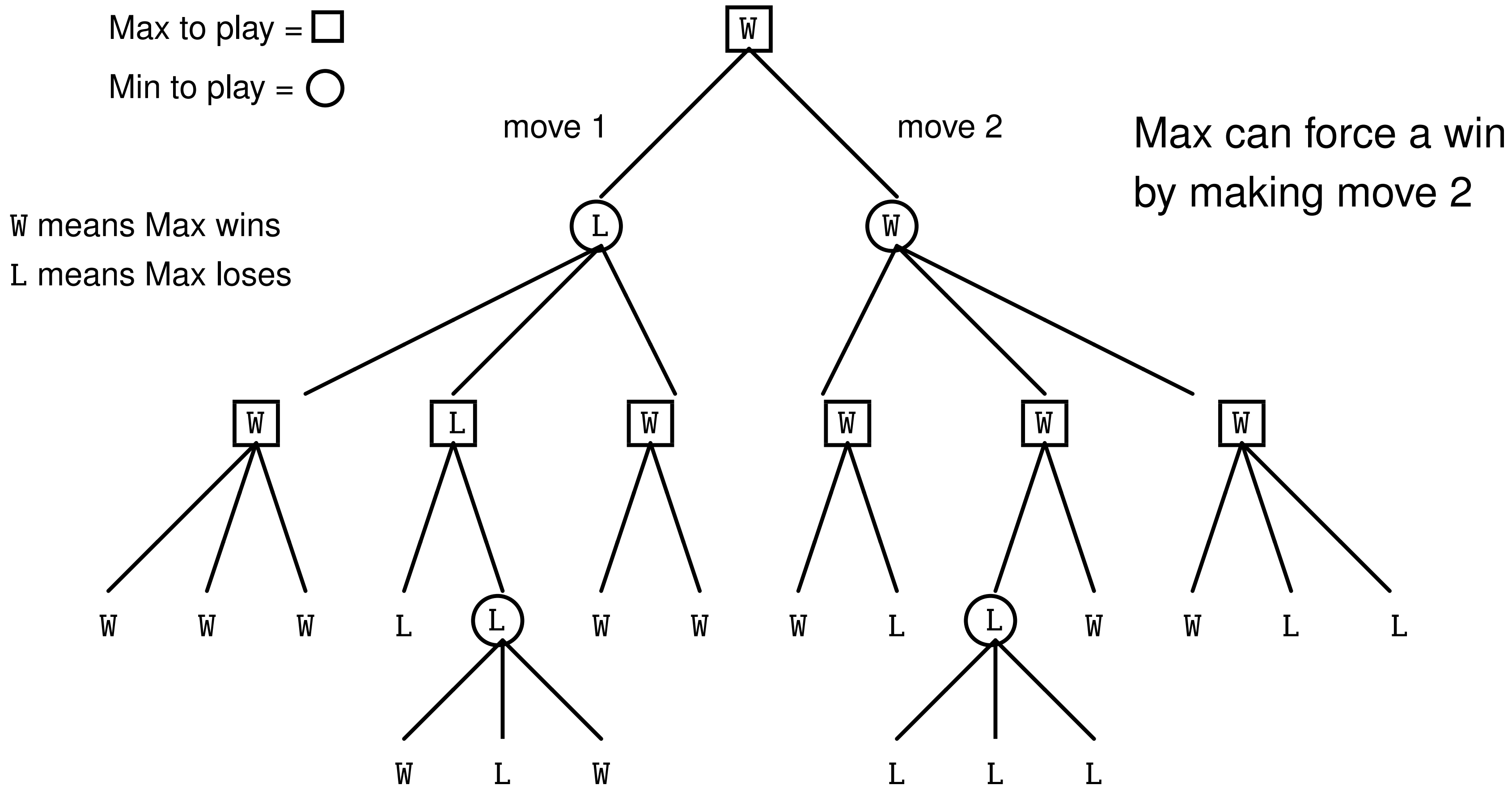
To see how this works, we can draw a tree called a *game tree* whose nodes correspond to states of the game:

- the root node of the tree is the *initial state* we start with;
- each leaf node of the tree is a *final state* of the game where one of the players has won or there is a tie;
- each non-leaf node has as children the game states we can get to by making *legal moves*;

By imagining that each player chooses a best move, we can label *every node* on the tree as a winning position for one of the players (or a tie).

An example game tree

What is the best move for Max if he starts at the root of this tree?



Max can force a win by making move 2

How did we do this?

Starting from the bottom, we did the labelling as follows:

- When it was Max's turn to play,
  - we labelled the state `W` (Max wins), if there was at least one move to a state labelled `W`;
  - we labelled the state `L` (Min wins), if all of the moves were to a state labelled `L`.
- When it was Min's turn to play,
  - we labelled the state `L` (Min wins), if there was at least one move to a state labelled `L`;
  - we labelled the state `W` (Max wins), if all of the moves were to a state labelled `W`.

Once the entire tree is labelled, it was easy to decide how to move!

Labelling the nodes in a game tree

Imagine that we have a game between two players P and Q, with P to play.

Here is how to label any game state S as P, Q, or `neither`:

1. if the game is actually over in state S, then label it with the winner;
2. otherwise, if P has a winning move, then label the node P;
3. otherwise, if P has a non-losing move, then label the node `neither`;
4. otherwise, label the node with Q as the winner.

When does P have a winning move from game state S?

When there is a legal move from S to S' where S' is labelled with P.

When does P have a non-losing move from game state S?

When there is a legal move from S to S' where S' is not labelled with Q.

A general game-playing program

gameplayer.pl

```
% This general game player needs these predicates to be defined:
%     - player(Player)
%     - game_over(State,Player,Winner)
%     - legal_move(BeforeState,Player,Move,AfterState)

% label(S,P,W): state S with player P to move is labeled winner W.
label(S,P,W) :- game_over(S,P,W).
label(S,P,P) :- win_move(S,P,_).
label(S,P,neither) :- \+ win_move(S,P,_), tie_move(S,P,_).
label(S,P,Q) :- opp(P,Q), \+ tie_move(S,P,_), \+ game_over(S,P,_).

% win_move(S,P,M):  P can win by making move M.
win_move(S,P,M) :- \+ game_over(S,P,_), opp(P,Q),
   legal_move(S,P,M,New), label(New,Q,P).

% tie_move(S,P,M):  P can avoid losing by making move M.
tie_move(S,P,M) :- \+ game_over(S,P,_), opp(P,Q),
   legal_move(S,P,M,New), \+ label(New,Q,Q).

opp(P,Q) :- player(P), player(Q), \+ P=Q.
```

This gives us a general game playing program. As with planning, it needs the predicates `player`, `legal_move`, and `game_over` defined elsewhere.

Tracing the labelling

To see how this labelling works, we load the general game player together with the program for a particular game, such as Race to 21.

When tracing a game playing program, it is useful to start near the end of the game, where only a few options are available.

```
?- label(3,max,max).                  % Can Max force a win with 3 chips left?
 T Call: (7) label(3, max, max)       % Is the game over?  No.
 T Redo: (7) label(3, max, max)       % Try again...

 T Call: (9) label(2, min, max)       % What if Max takes 1 chip
 T Call: (11) label(1, max, max)      %    Then if Min takes 1 chip
 T Exit: (11) label(1, max, max)      %       Max will win
 T Call: (11) label(0, max, max)      %    But if Min takes 2 chips
 T Fail: (11) label(0, max, max)      %       Max will lose
 T Fail: (9) label(2, min, max)       % So 2 is not a winning position for Max

 T Call: (9) label(1, min, max)       % What if Max takes 2 chips
 T Call: (11) label(0, max, max)      %    Then Min must take 1 chip
 T Fail: (11) label(0, max, max)      %       Max will lose
 T Fail: (9) label(1, min, max)       % So 1 is not a winning position for Max

 T Fail: (7) label(3, max, max)       % So Max cannot force a win with 3 chips
No
```

A longer trace

Here is a much longer trace, but with many of the details left out!

```
?- label(5,max,W).                 % Can Max win with 5 chips left?
 Call: (7) label(5, max, _G314)
                                   %---  try picking 1 chip
 Call: (9) label(4, min, max)      % is 4 a winning position for Max?
 Call: (11) label(3, max, max)     % |  if Min then moves 1 also
                                   % |    then we get to 3
 Fail: (11) label(3, max, max)     % |    where Max fails to win
 Fail: (9) label(4, min, max)      % No
                                   %---  try picking 2 chips
 Call: (9) label(3, min, max)      % is 3 a winning position for Max?
 Call: (11) label(2, max, max)     % |  if Min moves 1
                                   % |    then we get to 2
 Exit: (11) label(2, max, max)     % |    where Max wins
 Call: (11) label(1, max, max)     % |  if Min moves 2
                                   % |    then we get to 1
 Exit: (11) label(1, max, max)     % |    where Max wins
 Exit: (9) label(3, min, max)      % Yes
 Exit: (7) label(5, max, max)
W = max                            % So yes, max can win by picking 2 chips
```

Playing Race to 21 from the start

```
?- win_move(21,max,M).     % Can Max force a win from the initial state?
No

%%  There are no ties in Race to 21. So Max must choose any legal move.

?- win_move(20,min,M).     % Can Min force a win after Max moves 1?
M = 2

?- win_move(19,min,M).     % Can Min also force a win after Max moves 2?
M = 1

?- win_move(13,_,M).       % What should a player do with 13 left?
M = 1

?- win_move(14,_,M).       % What should a player do with 14 left?
M = 2
```

The winning strategy for a player in Race to 21 is to ensure that the number of chips left for the next player is divisible by 3.

So, the first player cannot force a win, but the second player can!

A second game: Tic-tac-toe

States: `[`p_1`,` p_2`,` p_3`,` p_4`,` p_5`,` p_6`,` p_7`,` p_8`,` p_9`]`

where each p_i is either `x`, o, or `-` (empty)

p_1	p_2	p_3
p_4	p_5	p_6
p_7	p_8	p_9

- initial state: `[-,-,-,-,-,-,-,-,-]`
  (We are using `-` as an atom, not to be confused with `_`.)
- game over: any state where
  - there are `x` marks making a straight line (`x` wins)
  - there are o marks making a straight line (o wins)
  - every p_i is either `x` or o (a full board)
- move: a number m where $1 \leq m \leq 9$ and $p_m =$ `-`

Tic-Tac-Toe in Prolog

tictactoe.pl

```
player(x). player(o).                % This is the tic-tac-toe game.

initial_state([-,-,-,-,-,-,-,-,-],x).                 % x moves first.

game_over(S,_,Q) :- three_in_row(S,Q).                % A winner
game_over(S,_,neither) :- \+ legal_move(S,_,_,_).     % A tie

three_in_row([P,P,P,_,_,_,_,_,_],P) :- player(P).
three_in_row([_,_,_,P,P,P,_,_,_],P) :- player(P).
three_in_row([_,_,_,_,_,_,P,P,P],P) :- player(P).
three_in_row([P,_,_,P,_,_,P,_,_],P) :- player(P).
three_in_row([_,P,_,_,P,_,_,P,_],P) :- player(P).
three_in_row([_,_,P,_,_,P,_,_,P],P) :- player(P).
three_in_row([P,_,_,_,P,_,_,_,P],P) :- player(P).
three_in_row([_,_,P,_,P,_,P,_,_],P) :- player(P).

legal_move([-,B,C,D,E,F,G,H,I],P,1,[P,B,C,D,E,F,G,H,I]).
legal_move([A,-,C,D,E,F,G,H,I],P,2,[A,P,C,D,E,F,G,H,I]).
legal_move([A,B,-,D,E,F,G,H,I],P,3,[A,B,P,D,E,F,G,H,I]).
legal_move([A,B,C,-,E,F,G,H,I],P,4,[A,B,C,P,E,F,G,H,I]).
legal_move([A,B,C,D,-,F,G,H,I],P,5,[A,B,C,D,P,F,G,H,I]).
legal_move([A,B,C,D,E,-,G,H,I],P,6,[A,B,C,D,E,P,G,H,I]).
legal_move([A,B,C,D,E,F,-,H,I],P,7,[A,B,C,D,E,F,P,H,I]).
legal_move([A,B,C,D,E,F,G,-,I],P,8,[A,B,C,D,E,F,G,P,I]).
legal_move([A,B,C,D,E,F,G,H,-],P,9,[A,B,C,D,E,F,G,H,P]).
```

Playing Tic-Tac-Toe

```
?- win_move([x,o,-,-,-,-,-,-,-],x,M).     % what can x do to win here?
M = 4 ;                                   % a surprising move?
M = 5 ;                                   % center
M = 7 ;                                   % a corner (but not 3 or 9)
No

?- win_move([x,-,-,-,-,-,-,-,-],o,M).     % can o force a win from here?
No

?- tie_move([x,-,-,-,-,-,-,-,-],o,M).     % can o force a tie from here?
M = 5 ;                                   % yes, but this is the only way
No

?- win_move([-,-,-,-,-,-,-,-,-],x,M).     % can x force a win initially?
No

?- tie_move([-,-,-,-,-,-,-,-,-],x,M).     % can x force a tie initially?
M = 1 ;                                   % yes, and any first move is ok!
M = 2 ;
% etc.
M = 9 ;
No
```

Playing a game all the way through

playuser.pl

```
% play_user(U): play entire game, getting moves for U from terminal.
play_user(U) :-
   initial_state(S,P), write('The first player is '), write(P),
   write(' and the initial state is '), write_state(S),
   play_from(S,P,U).

% play_from(S,P,U): player P plays from state S with user U.
play_from(S,P,_) :-                        % Is the game over?
   game_over(S,P,W), write('-------- The winner is '), write(W).
play_from(S,P,U) :-                        % Continue with next move.
   opp(P,Q), get_move(S,P,M,U), legal_move(S,P,M,New),
   write('Player '), write(P), write(' chooses move '), write(M),
   write(' and the new state is '), write_state(New),
   play_from(New,Q,U).

write_state(S) :- nl, write('   '), write(S), nl.

% Get the next move either from the user or from gameplayer.pl.
get_move(S,P,M,U) :- \+ P=U, win_move(S,P,M).       % Try to win.
get_move(S,P,M,U) :- \+ P=U, tie_move(S,P,M).       % Try to tie.
get_move(S,P,M,U) :- \+ P=U, legal_move(S,P,M,_).   % Do anything.
get_move(_,P,M,P) :-
   write('Enter user move (then a period): '), read(M).
```

Playing a full game of Tic-Tac-Toe

```
?- play_user(nobody).
The first player is x and the initial state is
    [-, -, -, -, -, -, -, -, -]
Player x chooses move 1 and the new state is
    [x, -, -, -, -, -, -, -, -]
Player o chooses move 5 and the new state is
    [x, -, -, -, o, -, -, -, -]
Player x chooses move 2 and the new state is
    [x, x, -, -, o, -, -, -, -]
Player o chooses move 3 and the new state is
    [x, x, o, -, o, -, -, -, -]
Player x chooses move 7 and the new state is
    [x, x, o, -, o, -, x, -, -]
Player o chooses move 4 and the new state is
    [x, x, o, o, o, -, x, -, -]
Player x chooses move 6 and the new state is
    [x, x, o, o, o, x, x, -, -]
Player o chooses move 8 and the new state is
    [x, x, o, o, o, x, x, o, -]
Player x chooses move 9 and the new state is
    [x, x, o, o, o, x, x, o, x]
-------- The winner is neither          % no big surprise here!
```

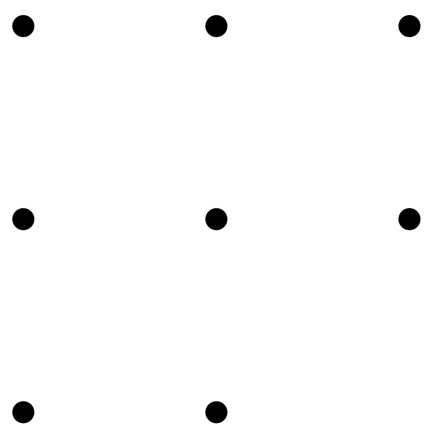
Playing against X

This time, the moves for O are entered (using the special predicate `read`):

```
?- play_user(o).
The first player is x and the initial state is
    [-, -, -, -, -, -, -, -, -]
Player x chooses move 1 and the new state is
    [x, -, -, -, -, -, -, -, -]
Enter user move (then a period): 2.          % a bad move!
Player o chooses move 2 and the new state is
    [x, o, -, -, -, -, -, -, -]
Player x chooses move 4 and the new state is
    [x, o, -, x, -, -, -, -, -]
Enter user move (then a period): 7.          % a fine move
Player o chooses move 7 and the new state is
    [x, o, -, x, -, -, o, -, -]
Player x chooses move 5 and the new state is
    [x, o, -, x, x, -, o, -, -]
Enter user move (then a period): 6.          % a good move
Player o chooses move 6 and the new state is
    [x, o, -, x, x, o, o, -, -]
Player x chooses move 9 and the new state is
    [x, o, -, x, x, o, o, -, x]
-------- The winner is x                     % but X wins anyway
```

A more challenging game

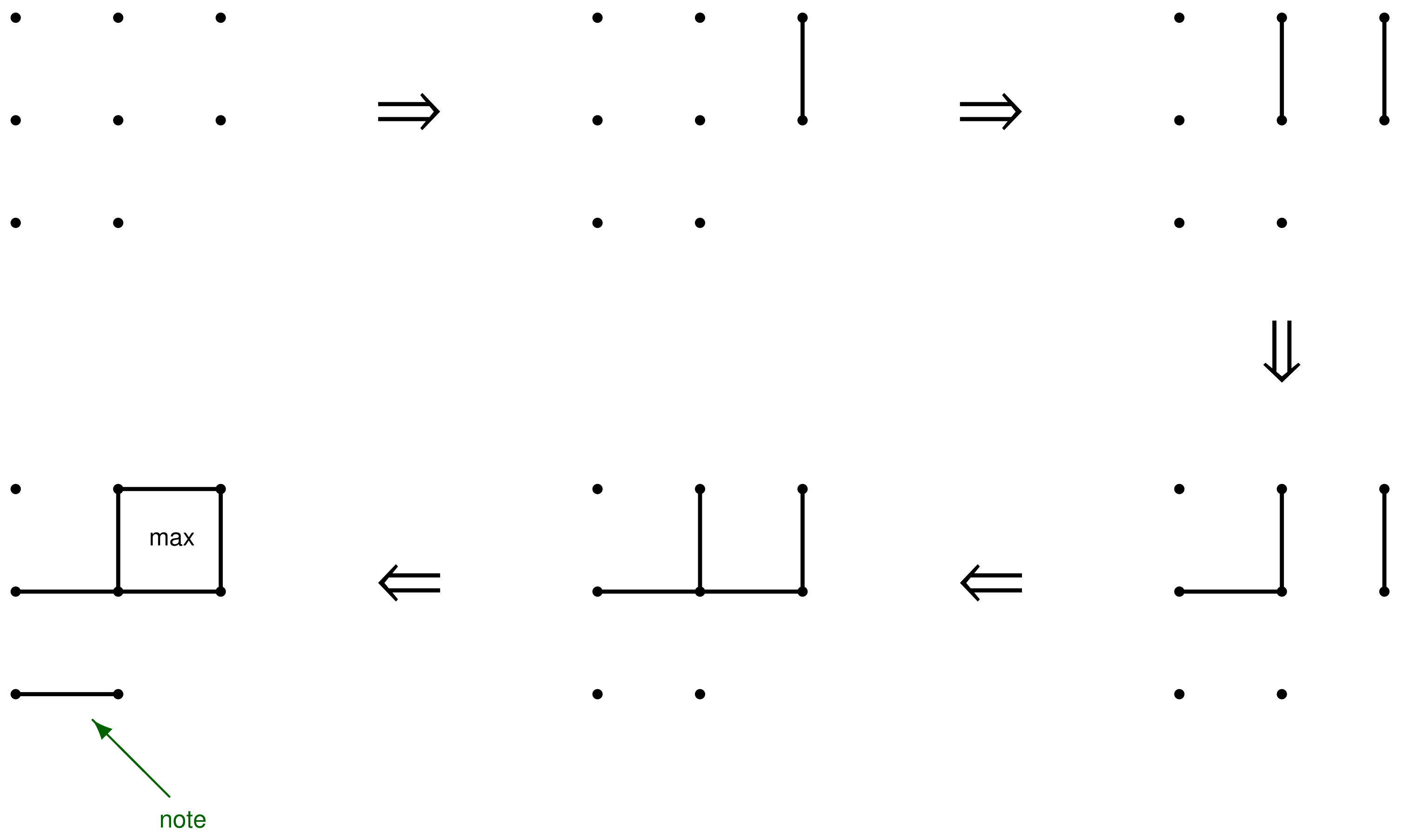
The game of *Boxes*:



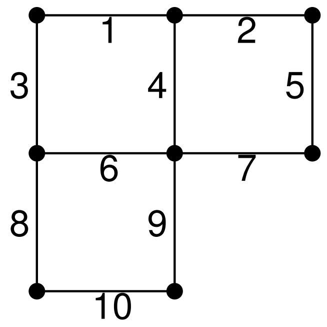
There are 8 dots aligned as in the diagram, making the outline of 3 squares. At each turn, a player draws one of the undrawn edges of a square by connecting two adjacent dots. If this happens to be the last undrawn edge of a square, the player owns the square, and may optionally move again.

The first player to own 2 squares wins.
(There are no ties.)

The start of an example game



Representing Boxes states and moves



Square 1 has lines 1, 3, 4, 6

Square 2 has lines 2, 4, 5, 7

Square 3 has lines 6, 8, 9, 10

A *move* will be a list of one or more lines drawn.

A *state* will be a list of terms of the form `draw`($player$, $line$), with the most recent moves first.

This is like the situation representation (from planning), and will allow us to calculate what we need:

- `own`($player$, $square$, $state$) the player owns the square
- `avail_line`($line$, $state$) the line has not yet been drawn

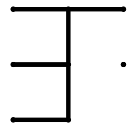
The Boxes game in Prolog

boxes.pl

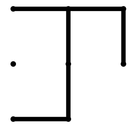
```
player(max).    player(min).              % This is the Boxes game.

square_lines(sq1,[1,3,4,6]).              % Lines for square 1
square_lines(sq2,[2,4,5,7]).              % Lines for square 2
square_lines(sq3,[6,8,9,10]).             % Lines for square 3

initial_state([],max).                    % Initially no lines are drawn.
game_over(St,_,W) :-                      % Winner W owns two squares.
   owns(W,Sq1,St), owns(W,Sq2,St), \+ Sq1 = Sq2.

% Player P owns Sq if just drew last line or owned Sq before.
owns(P,Sq,[draw(P,L)|St]) :- last_avail_line(L,Sq,St).
owns(P,Sq,[_|St]) :- owns(P,Sq,St).

% Line L is available and is the last of square not yet drawn.
last_avail_line(L,Sq,St) :-
   avail_line(L,Sq,St), \+ avail_line(_,Sq,[draw(_,L)|St]).

% Line L is from Sq and not yet drawn in state St.
avail_line(L,Sq,St) :-
   square_lines(Sq,Ls), member(L,Ls), \+ member(draw(_,L),St).

% The legal moves
legal_move(St,P,[L],[draw(P,L)|St]) :-   % Draw a line and stop.
   avail_line(L,_,St).
legal_move(St,P,[L|Rest],New) :-         % Draw a line and go on.
   last_avail_line(L,_,St), legal_move([draw(P,L)|St],P,Rest,New).
```

Running Boxes

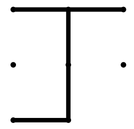
```
?- win_move([ draw(min,6),draw(max,10),draw(min,2),draw(max,4),
                draw(min,9),draw(max,1) ],  max, Move).
Move = [3, 8]
Yes
```



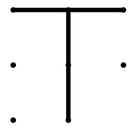
```
?- win_move([ draw(min,5),draw(max,10),draw(min,2),draw(max,4),
                draw(min,9),draw(max,1) ],  max, Move, _).
Move = [7] ;              % Note: draw one line only!
No
```



```
?- tie_move([             draw(max,10),draw(min,2),draw(max,4),
                draw(min,9),draw(max,1) ],  min, Move).
No                        % So Min loses from here
```



```
?- win_move([                          draw(min,2),draw(max,4),
                draw(min,9),draw(max,1) ],  max, Move).
Move = [8] ;
Move = [10] ;             % As above
No
```



```
?- win_move([],max,M).    % Can the first player guarantee a win?
No                        % After 2 minutes of hard computing!
```

So where is the fun?

Every game of the sort we are considering, including Chess and Go, has the property that one of the following conditions must hold:

- the first player can guarantee a win
- the first player can guarantee at least a tie
- the second player can guarantee a win

So what exactly is the fun of playing??

Can we not simply run

```
    ?- initial_state(S,white), win_move(S,white,Move).
```

to find out what the *best first move of Chess* is, once and for all, *etc.*?

Big games

The problem is that the `win_move` program we have been using is practical only for small games.

To decide how to move, the program considers *all* the states up to the end of the game.

For the first move, this means searching the *entire* space:

- for Checkers: about 10^{70} states
- for Chess: about 10^{120} states
- for Go: about 10^{750} states!

Impractical for even the fastest supercomputers!

For big games like these, we need to be able to decide on a best move without examining the entire space. This is what makes it challenging!

Handling big games

There are two ideas that will allow us to decide on a move to make without having to examine an entire game tree (to the very end of the game):

- replace the `W`, `L` and `T` (for ties) labels on the tree by *numbers*, where
  - a large +ve number (say 999) means a winning position for Max,
  - a large -ve number (say -999) means a winning position for Min,
  - 0 means a position that is equally good for Min and Max.

- allow other numbers to be assigned to nodes on the tree that are not at the end of the game, by *estimating* how good the position is for Max.

  for example, a node labelled 108 is estimated to be better for Max than a node labelled 27, which is better than a node labelled -48.

So Max will be interested in getting as *large* a value as possible, and Min is interested in getting as *small* a value as possible.

Numeric game trees

The game playing procedure that deals with a game tree with numeric labels on the nodes is called *minimax*.

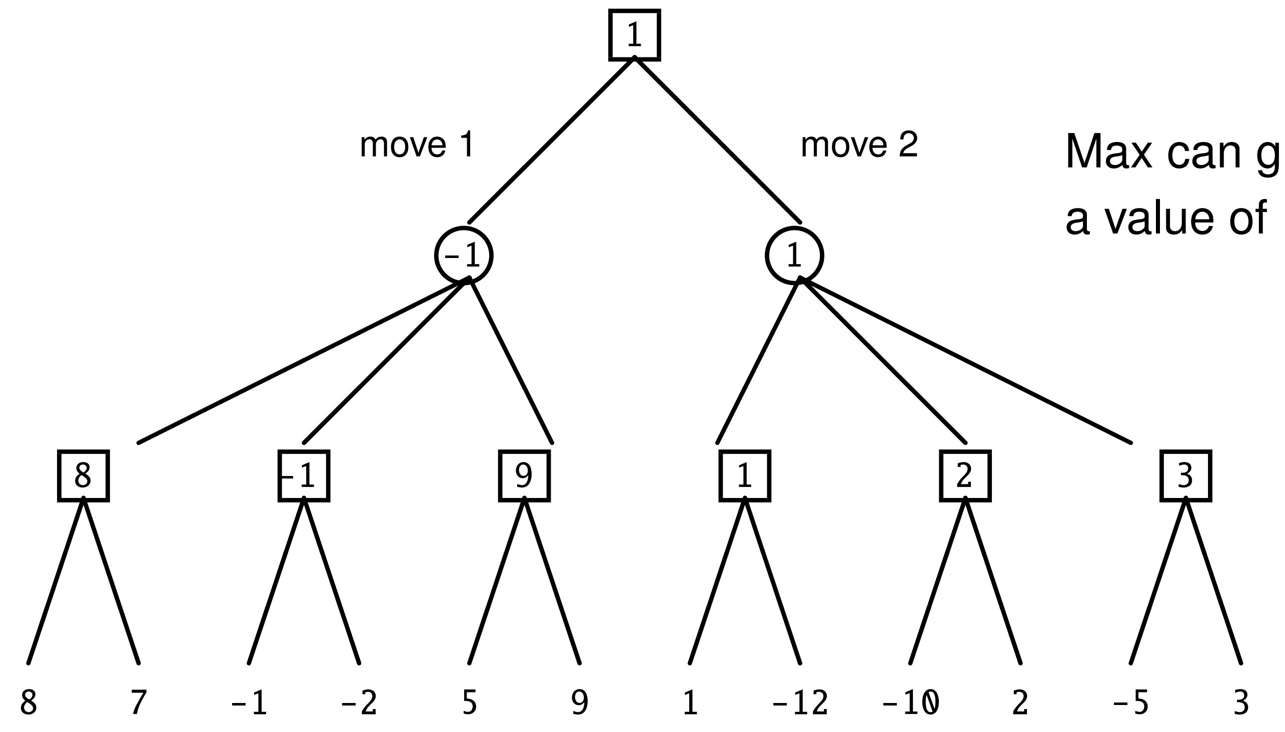
Numeric game trees are similar to the win/lose game trees from before:

- the root node of the tree is the initial state we start with;
- each leaf node is a state that has been given a numeric value (either at the end of the game, or as an estimate).
- each non-leaf node has as children the game states we can get to by making legal moves;

By looking at this tree, and the numbers on the leaf nodes, we can determine the largest number that Max can guarantee and the smallest number that Min can guarantee.

Examining a game tree

For example, suppose we run the minimax procedure on a tree of depth 3 only, where estimates are then provided for the values of the leaf nodes.



Max can guarantee a value of 1

Calculating backed up values

Using this tree, it is easy to calculate the *backed-up value* of any node.

> This is the largest value that Max can guarantee and the smallest value that Min can guarantee.

Here are the rules:

- for a leaf node, the backed up value is the given value (at the end of the game or estimated);
- for a non-leaf Max node, the backed up value is the *maximum* of the backed value of its children;
- for a non-leaf Min node, the backed up value is the *minimum* of the backed value of its children.

This finally explains the names Max and Min!

Minimax in Prolog

The easiest way to handle minimax in Prolog is to think of the two players trying to get to a game state with a certain value V or better:

- for Max, better means $> V$:
- for Min, better means $< V$:

What are the states where a player can get a value of V or better?

- a final state where the given value is V or better;
- a state where the player has a legal move to a new state where the *opponent* is limited:
  - for Max: a new state where Min cannot get $V - 1$.
  - for Min: a new state where Max cannot get $V + 1$.

These rules will lead us to another general game playing program.

Minimax game player

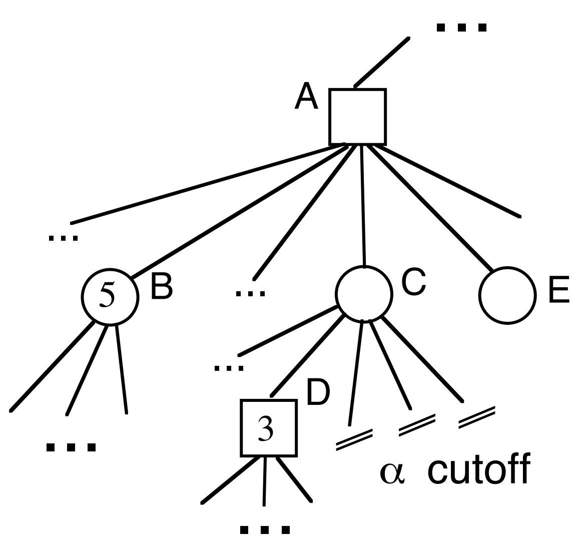
minimax.pl

```
% Computing the values of moves for two players, Max and Min.
% The game-dependent predicate estval estimates values of states.
% The search of the game tree is limited to a depth given by D.

% can_get(S,P,D,V): P can get a value of V or better in state S.
can_get(S,max,_,V) :- game_over(S,max,W), winval(W,V1), V1 >= V.
can_get(S,min,_,V) :- game_over(S,min,W), winval(W,V1), V1 =< V.
can_get(S,max,0,V) :- \+ game_over(S,_,_), estval(S,max,E), E >= V.
can_get(S,min,0,V) :- \+ game_over(S,_,_), estval(S,min,E), E =< V.
can_get(S,P,D,V) :- \+ game_over(S,_,_), val_move(S,P,D,_,V).

% val_move(S,P,D,M,V): P can get a value of V or better with move M.
val_move(S,max,D,M,V) :- D>0, D1 is D-1, V1 is V-1,
   legal_move(S,max,M,S1), \+ can_get(S1,min,D1,V1).
val_move(S,min,D,M,V) :- D>0, D1 is D-1, V1 is V+1,
   legal_move(S,min,M,S1), \+ can_get(S1,max,D1,V1).

winval(max,999).      % The value of a state where Max has won
winval(min,-999).     % The value of a state where Max has lost
winval(neither,0).    % The value of a state with a tie
```

Further reducing the search

Even only looking at the game tree to a certain depth, there may still be too many states to consider comfortably.

We can sometimes avoid searching large sections of a game tree without changing the backed up values that will be calculated.



Suppose we want to find out what Max can get in state A.

Along the way, we compute that Min can get 5 (or less) at state B, and then later that Max can get (at least) 3 at state D.

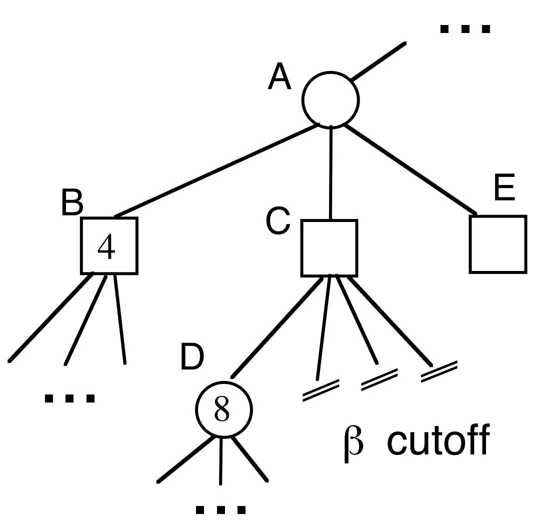
There is no point of continuing to look below state C, since it will be less than B.

So we can go directly to examining state E.

Alpha and beta cutoffs

The cutoff we saw in the previous slide is called an *alpha cutoff*.

There is an analogous one for Min called a *beta cutoff*.



Suppose we want to find out what Min can get in state A.

Along the way, we compute that Max can get (at least) 4 at state B, and that Min can get 8 (or less) at state D.

There is no point of continuing to look below state C, since it will be greater than B.

Again, we can move directly to state E.

These cutoffs can be very significant, and especially if they happen high up in the game tree.

Specifying games for minimax

To be able to use the version of minimax that does not explore the entire tree, we need to provide all of the predicates from before, `player`, `initial_state`, `legal_move`, `game_over`, and one additional one:

- `estval`(*state*, *player*, *value*)

  given that it is the turn of *player* in *state*, *value* is the estimated measure of how good the state is, from the point of view of Max.

Different games will have different methods of estimating how well or how poorly Max is doing at non-final states of the game.

A typical calculation is a *sum* over a number of factors that contribute to the success of a player.

Estimating states in Chess

Estimates of games states is critical for big games like Chess where it is not feasible to search the entire tree.

A typical estimate in Chess is something like:

$$\mathit{material\ advantage} \ + \ \mathit{positional\ advantage}$$

where $\mathit{material\ advantage}$ is of the form

$$9 \cdot Q \ + \ 5 \cdot R \ + \ 3 \cdot N \ + \ 3 \cdot B \ + \ P$$

where Q, R, N, B and P is the advantage in queens, rooks, knights, bishops, and pawns, respectively.

State of the art in Chess

The Chess playing program, *Deep Blue*, was the first to beat a world champion (Kasparov) in 1997.

- it uses minimax with alpha-beta cutoffs, but on a special computer to speed up the generation of moves and the evaluation of states

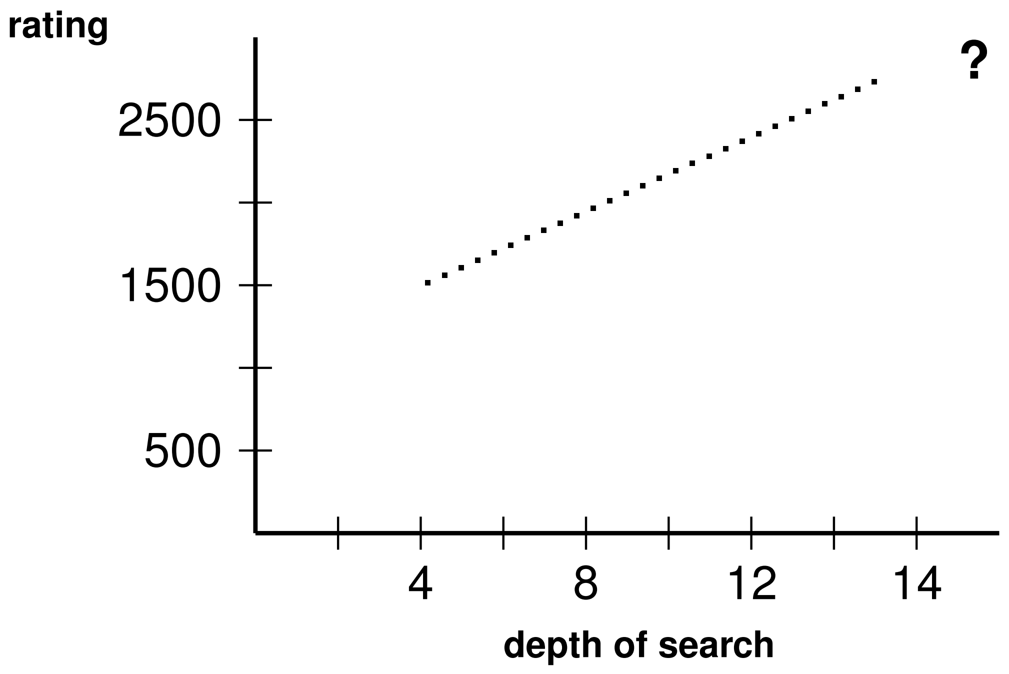
  200 million board positions per second!

- estimates of values of game states uses 8000 separate factors
- explores game trees to about depth 14, with occasional very deep searches
- rating of 2650 in 1996 (grandmaster = 2500)

Note: As of January 2011, the current Chess champion, Magnus Carlsen (born in 1990), had a rating of 2814!

The future of strategic games

Almost all of the development effort in game playing programs goes into speeding up the search.



The US Chess Federation rating of Chess programs seems to be proportional to the search depth.

This effort so far has not helped with *extremely* large games like Go.

11

Case Study: Other Ways of Thinking

Using what you know

For us, thinking has meant using what you know, where:

- *"what you know"* means a KB of atomic and conditional sentences
- *"using"* means applying back-chaining

Even when we said we had to think in *different* ways, it always meant using back-chaining but over a different KB.

But there are other ways to use what is known beyond back-chaining:

- explanation
- learning
- propositional reasoning

These other ways emerge nicely as a form of *thinking about thinking*.

Here: a simple version for thinking about back-chaining.

What is the knowledge about?

So far, we have used Prolog to represent knowledge about a variety of domains: family relations, people in parks, visual scenes *etc.*.

For example, in the family domain, we had:

- the objects: `john`, `sue`, `sam`, ...
- the properties and relationships: `child`, `parent`, `male`, ...
- the facts: `male(john)`, `child(john,sam)`,
  `parent(Y,X) :- child(X,Y).  ...`

As we move to a new domain, we would use

- Prolog terms to stand for the new objects in that domain, and
- Prolog predicates to stand for the new relations in that domain.

Thinking about thinking

Now imagine that we want to think not about the members of some family, but about *back-chaining* itself.

Q: What are the objects in this domain?

A: Knowledge bases, queries, atoms, clauses, ...

Q: What are the relations?

A: The main relation we care about is the one that holds between a KB and a query when the query can be *established* from the KB.

To think about thinking, we need to represent this domain in Prolog!

We need Prolog terms that stand for the objects and Prolog predicates that stand for the relations.

Representing the objects

For simplicity, let us assume that we only want to think about a Prolog that has no variables.

For example, here is a simple clause: `u :- p, b.`

We can use a list (as a term) to represent it: `[u,p,b]`.

Here is a simple knowledge base

```
    a.
    b.
    u :- p, b.
    p :- a.
```

We can represent it using a list of lists: `[[a], [b], [u,p,b], [p,a]]`.

Similarly, a query like `?- p, b.` can be represented by `[p,b]`.

Representing the relations

We need Prolog predicates that stand for the relations we care about.

The only predicate we want is `est(K,Q)` which should hold when the query represented by `Q` can be established by back-chaining from the KB represented by `K`.

In other words, we want this behaviour:

```
?- est([[a],[b],[u,p,b],[p,a]], [p,b]).
Yes

?- est([[a],[b],[u,p,b],[p,a]], [c]).
No
```

Q: How do we get this behaviour?

A: We need to state in Prolog how back-chaining works!

Back-chaining: a mini review

Putting aside variables, negation, equality, and other complications, here is a very terse description of how back-chaining works in Prolog:

The task is to try to establish a query $A_1, \ldots A_n$, given a KB.
The back-chaining procedure is as follows:

1. If $n = 0$, then there is nothing to do, and we return success.
2. Otherwise, we need to find a clause in the KB whose head is A_1 (the first atom of the query) and whose body is $B_1, \ldots B_m$ (where $m \geq 0$), such that we are able to establish (recursively) the query $B_1, \ldots B_m, A_2, \ldots, A_n$.

Based on this, we can write the clauses that characterize when the `est` predicate should hold:

```
% est(K,Q) holds if query Q can be established from KB K.
est(_,[]).
est(K,[A|T]) :- member([A|B],K), append(B,T,Q), est(K,Q).
```

These two simple clauses are all we need to get the behaviour we want!

Another way of thinking

Once we get used to writing clauses like this

```
% est(K,Q) holds if query Q can be established from KB K.
est(_,[]).
est(K,[A|T]) :- member([A|B],K), append(B,T,Q), est(K,Q).
```

we can consider other forms of thinking that diverge from Prolog.

Here is a very simple variant of `est`:

```
% estbf(K,Q) holds if Q can be established from K.
% The method is a breadth-first variant of back-chaining
estbf(_,[]).
estbf(K,[A|T]) :- member([A|B],K), append(T,B,Q), estbf(K,Q).
```

The we get this behaviour:

```
?- est([[a],[p,p]], [p,b]).    % gets stuck in a loop, like Prolog
ERROR: Out of global stack

?- estbf([[a],[p,p]], [p,b]).  % does not get stuck, unlike Prolog!
No
```

A glimpse of explanation and learning

Using `est`, we can consider using what is known in very different ways!

For example, we might want to ask a question like this:

> What would explain Q being true?
>
> = What fact F could I add to my KB to allow Q to be established?

```
?- KB=[[a],[u,p,b],[p,a]], est([[F]|KB],[u]), \+ F=u.
F = b
```

This is a simple form of *explanation*: in the example above, we would say that given what is known, b would explain u being true.

A simple form of *learning* is similar except that we need to use variables:

> What clause (with variables) could I add to my KB to allow observations $O_1, \ldots, O_n$ to be established.

12

Can Computers Really Think?

Ending on a philosophical note . . .

The starting point of the course: the mystery of intelligent behaviour

> People are somehow able to act intelligently because they happen to *know* a lot about the world around them.

Thinking:
> bringing what you know to bear on what you are doing

But how exactly does thinking work?

In this course, we really just studied one idea in detail:

> Thinking can be usefully understood as a computational process.

Computational process:
> manipulating symbolic structures according to a procedure

Review: Intelligent behaviour (Slide 2)

In this course, we will consider what it takes to get a computer to engage in intelligent activities such as

- understanding English sentences;
- making sense of a visual scene;
- planning how to achieve a goal;
- solving puzzles like Sudoku;
- playing games like chess.

What these very different activities all have in common, is that when they are performed by people, they seem to require *thought*.

The problem of scale

While we saw what it takes to get a computer to perform these activities, we also saw that we ran into difficulties for the complex cases:

- the 9×9 Sudoku *vs.* the 4×4
- large visual scenes with many objects
- non-referential noun phrases
- plans with hundreds or thousands of steps
- advanced games like Chess or Go

One of the major challenges of current AI research is to find ways of dealing with more realistic problems, and we got a glimpse of some:

- how to do better than generate and test
- how to represent action and change for large worlds
- how to play games without searching the entire game tree   *etc.*

Can computers think?

Nonetheless it is clear from what we done in the course that we can get computers to engage in activities that require thinking in people.

But does this mean that the computers are *really* thinking?

This question should be an easy one to answer by now!

But it is not.

Some people might argue: Computers cannot possibly think because

- they are not people;
- they do not have bodies;
- they do not souls; *etc.*

How can we resolve this?

The Turing approach

Alan Turing (the first Computer Scientist) suggested the following:

The terms *thinking*, *understanding*, *intelligence*, *etc.* are too *vague* to be worth spending any time arguing about.

The person beside you may appear to be thinking and understanding what you say, but there is no way to say conclusively that he really is.

> He could be just a “zombie” that is somehow faking it.

We should not expect to be able to do any better with computers.

> At the very best, we will only ever be able to say that the computer is thinking or understanding as much the person is.

If we cannot tell the difference in how computers *act* in the long haul from people, the rest is just prejudice (people chauvinism).

The Turing Test

The Turing Test: extended conversation over a teletype.

The conversation between an interrogator and two participants is natural, free flowing, and about *any* topic whatsoever.

Passing the Turing Test: A computer that has the property that no matter how long the conversation, the interrogator cannot tell which of the two participants is a person and which is the computer.

(Put aside for now the question concerning how (or if) we can write a program that will allow a computer to pass the Turing Test.)

Turing's point: we should be willing to say that a computer that passes the test is thinking (or is understanding or is intelligent) as much as the person.

The philosopher John Searle disputes this!
We conclude the course by looking at his argument.

The Chinese Room argument

Imagine a program that can converse intelligently in written Chinese.

= passes the Turing Test

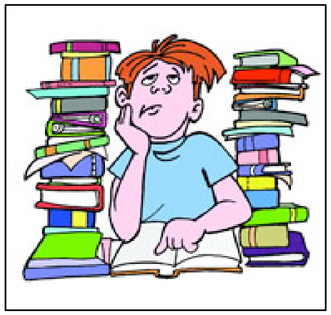
Suppose the program is written in Prolog: `chinese.pl`.

Searle (imaginary version):

- does not understand Chinese;
- but knows Prolog well !

Imagine that Searle is in a room with the text of `chinese.pl` in a book, and performs the job of the Prolog interpreter.

您今天怎么样啊？



Chinese Room

还不错，就是有点头疼。

His external behaviour is the same as that of the computer, passing the Turing Test, without requiring him to understand Chinese!
So Turing is wrong!

The Systems Reply

Reply: It is true that Searle does not understand Chinese.

But the "system" consisting of (Searle + `chinese.pl`) does.

So the intelligent behaviour is not being faked. Turing is right!

Searle's Answer to the Reply: Imagine that Searle memorizes the instructions in the book and then discards it.

Result: Searle (alone now) gets the behaviour right,
but still does not understand Chinese.

So: intelligent behaviour is being faked after all. Turing is wrong!

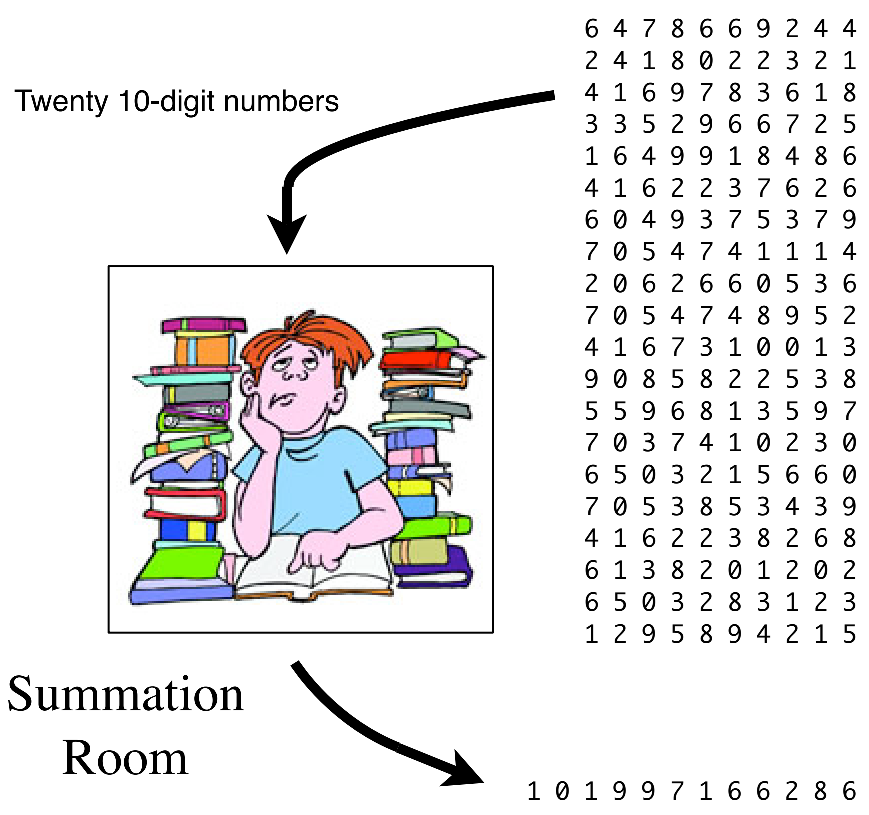
Is this answer to the Systems Reply the last word?

How can we be so sure that Searle does not end up understanding Chinese after learning what is in the book?

Another room: The Summation Room

We imagine the following setup:

- the person in the room does not know how to add
- there is an instruction book that tells him that to do with the input to produce the output
- the input is a list consisting of just twenty 10-digit numbers
- the output is a 12-digit number that happens to be the *sum* of the input numbers



```
6 4 7 8 6 6 9 2 4 4
2 4 1 8 0 2 2 3 2 1
4 1 6 9 7 8 3 6 1 8
3 3 5 2 9 6 6 7 2 5
1 6 4 9 9 1 8 4 8 6
4 1 6 2 2 3 7 6 2 6
6 0 4 9 3 7 5 3 7 9
7 0 5 4 7 4 1 1 1 4
2 0 6 2 6 6 0 5 3 6
7 0 5 4 7 4 8 9 5 2
4 1 6 7 3 1 0 0 1 3
9 0 8 5 8 2 2 5 3 8
5 5 9 6 8 1 3 5 9 7
7 0 3 7 4 1 0 2 3 0
6 5 0 3 2 1 5 6 6 0
7 0 5 3 8 5 3 4 3 9
4 1 6 2 2 3 8 2 6 8
6 1 3 8 2 0 1 2 0 2
6 5 0 3 2 8 3 1 2 3
1 2 9 5 8 9 4 2 1 5
```

```
1 0 1 9 9 7 1 6 6 2 8 6
```

Question: Can a book allow the behaviour here to be faked?

An instruction book à la Searle

The Preface:

> This is a very large book: 10 billion chapters, each with 10 billion sections, each with 10 billion subsections, *etc.* up to depth 20.
>
> Take the first number in the list of 20 and go to that chapter;
> then take the second number in the list and go to that section;
> then take the third number and go to that subsection; *etc.*
>
> After doing this for all the 20 numbers in the list, there will be a number written in the book with at most 12 digits. Write that number on a slip of paper and hand that message back outside the room.

Now suppose that the book is constructed so that the 12-digit number is the sum of the 20 numbers required to get there!

Then this book allows the person in the room to get the right answers without showing him how to add, just like in the Chinese Room!

The catch!

The catch is that this book cannot possibly exist!

It would need to contain $10^{10} \times 10^{10} \times \cdots \times 10^{10} \; = \; 10^{200}$ entries.

But our universe only has about 10^{100} atoms!

However, ...

There is a much smaller book that *will* work:

Have a look at **PROC0**, **PROC1**, **PROC2** *etc.* on Slides 12–18 which we studied at the start of this course.

A book that has the instructions on these slides will allow a person who does not know how to add to get the right answers!

The big difference: anyone memorizing this book is *learning to add*.

This suggests that if the behaviour is complex enough, it cannot be faked!

A final word

In the end, maybe the *real* question about the Turing test, the Chinese Room, the Summation Room etc. is not

> Is a computer that is behaving intelligently really and truly thinking?

but rather

> What does it take to produce that intelligent behaviour?

This is not so much a philosophical question as a *scientific* one!

In this course, we saw very simple forms of intelligent behaviour and how to achieve them.

But this is really just the start. . . .

THE END