

# **Introduction to Symbolic AI: COMP24412**

Allan Ramsay  
School of Computer Science,  
University of Manchester,  
Manchester M13 9PL, UK

# Contents

Course outline	13
Prolog	15
Prolog is different	16
Some practicalities	17
What are Prolog programs made of?	22
Equality, unification & assignment	34
Datastructures	36
Lists	37
Recursion	43
Cut: !	58
Prolog programs as Prolog datastructures	63
Terms	64
Clauses & the database	67
Negation, local cut	71

<b>N-QUEENS</b>	<b>77</b>
Making connections between states of affairs	98
[...] provide precise ways of specifying facts and rules	106
Complex formulae	110
Model theory	116
Truth in an interpretation	120
Proof theory	125
Extensions	150
<b>CHECKPOINT</b>	<b>157</b>
Planning	158
<b>STRIPS</b>	<b>160</b>
Blocks World Example	169
<b>ACTIONS</b>	<b>170</b>
Goal interactions	186
Hierarchical planning	195

Forward chaining	202
Richer Description Languages	207
CHECKPOINT	211
Arrangements of words: grammar and parsing	217
Phrase structure rules	221
Fine-grained constraints	238
Subcategorisation labels	247
Features	252
Categorial grammar	256
Movement	275
CHECKPOINT	282
Parsing	283
Top-down parsing	285
Bottom-up	290
Chart parsing	294

CHECKPOINT	353
Semantics	354
What's it all about?	355
How does language work?	356
Doing something with ideas	358
Complexity	360
Compositionality	364
Compositional=modular	365
Compositionality & ambiguity	366
$\lambda$ -calculus	368
Quantification and reference	381
CHECKPOINT	403
Lexical semantics (I)	404
Events & thematic roles	408
Thematic roles: weak version	411

Thematic roles: strong version	415
CHECKPOINT	422
Adjectives	423
Adjectives	424
CHECKPOINT	437
Meaning postulates	438
Dynamic semantics	442
Interaction with <i>'not'</i>	445
CHECKPOINT	471

Artificial intelligence: getting computers to do tasks that would be regarded as requiring intelligence if a human did them

Seeing the world (Computer vision)

Moving around in the world (Robotics)

Making connections between states of affairs (Reasoning, planning)

Communicating (Natural language processing)

Getting better at any of the above (Machine learning, data mining)

You need domain knowledge for some of these:

- Computer vision: you need to know about light, about reflective properties of surfaces, about 3D geometry and trigonometry
- Robotics: you need to know about mechanics (and you need to be able to see!)
- Communication: you need to know about language (grammar, semantics, ...) and about speech (acoustics, phonetics, ...)



You need general principles for some of them:

- Reasoning: you need to know about logic, about state descriptions, about inference engines
- Learning: you need to know about probability theory, information theory

Tasks that require intelligence seem to come in two flavours:

- Ones that require you to manipulate symbols and structures
- Ones that require you to look for patterns

This course is about tasks that require you to manipulate symbols and structures

Seeing the world (Computer vision)

Moving around in the world (Robotics)

Making connections between states of affairs (Reasoning, planning)

Communicating (Natural language processing, speech processing)

Getting better at any of the above (Machine learning, data mining)

When you're manipulating symbols and structures, you need to be able to work with partial descriptions of entities (like MySQL but better: a complete description of a person might include their name, age, shoe size, height, weight, hair colour, eye colour, address, job, mother's maiden name, primary school, secondary school, . . . . But if I want to find people who are at risk of diabetes then I just want people over the age of 50 who weigh more than 75kg)

When you're trying to find a route from A to B, you might have a list of roads that go from A to somewhere, and you might want to try them each in turn.

So we will want to use a programming language which supports comparing partial descriptions, and which allows you to try out different ways of achieving some goal: **PROLOG**

- Prolog: 2 weeks
- Reasoning
  - Logic and theorem proving: 2 weeks
  - Planning: 2 weeks
- Natural language
  - Grammar and parsing: 2 weeks
  - Meaning: 2 weeks
- Putting it all together: 1 week

# 0 PROLOG

Every other programming language we teach you is just like every other programming language we teach you: Java, Python, C

Every other programming language you will ever use is just like every other programming language you will ever use: Java, Python, C, C++, C#, VB, JavaScript, Fortran, ...

**Prolog is different<sup>1</sup>**. Which is good for you.

---

<sup>1</sup>I usually think that telling people that something is difficult/weird is a bad idea. But Prolog is so different that I think that this time it's right to emphasise it.

## Prolog is different

---

You don't tell it what to do. You list ways of doing things, and ask it to try them out.

You don't store data in variables. You provide partial descriptions of things, and you try to make sure that the objects you are working with fit your descriptions (might not look like that to start with, but bear it in mind)



## Some practicalities

---

There are several Prolog compilers: there are two on lab machines, SWI-Prolog and SICStus Prolog. SWI is free (so you can put it on your own machine), but I prefer SICStus. I will try to ensure that the coursework can be done using either, but there may be mismatches.

There will be programs to try out and exercises to do: these will all live in

```
/opt/info/courses/COMP24412/PROGRAMS
```

Might be worth defining this as an environment variable in your `.bash_profile`

```
COMP24412PROGRAMS=/opt/info/courses/COMP24412/PROGRAMS
```

Then you can just do

```
$ cp $COMP24412PROGRAMS/lecture1.pl .
```

to get a copy of the Prolog programs that go with the first lecture.

There is no very good IDE for Prolog. The best thing to use is Emacs, with suitable support files. This can be easily set up on lab machines running Unix, I haven't been able to do so for Windows.

You may or may not have a file called `.emacs` in your home directory. If you've got one, add the lines

```
(autoload 'run-prolog "prolog" "Start a Prolog sub-process." t)
(autoload 'prolog-mode "prolog" "Major mode for editing Prolog programs." t)
(autoload 'mercury-mode "prolog" "Major mode for editing Mercury programs." t)
(setq prolog-system 'sicstus)
(setq auto-mode-alist (append '(("\\.pl$" . prolog-mode)
                                ("\\.m$" . mercury-mode))
                              auto-mode-alist))

(setq prolog-indent-width 4)
```

at the end, if you haven't got one create one and put these lines in it. Emacs will then work as an IDE for Prolog (it also provides quite a nice IDE for Python, and a reasonable one for LaTeX. I never leave it!)

(cut and paste this code from the online version of the notes or from `prologemacs` in `/opt/info/courses/COMP24412/PROGRAMS`)

There are lots of reasonable Prolog textbooks.

<http://www.learnprolognow.org/lpnpag.php?pageid=online> (*Learn Prolog Now!*: Patrick Blackburn, Johan Bos, and Kristina Striegnitz) is freely available online and is by people who work in this general area, so the examples in their book are relevant.

<http://sicstus.sics.se/sicstus/docs/latest3/html/sicstus.html>  
Manual for the version of Prolog that we will be using (SICS-tus)

# What are Prolog programs made of?

---

You write a Prolog program by writing a collection of facts and rules. You run it by asking a question.

- A fact is made up of a relation name and a set of terms, where the simplest kind of term is just a constant. Relation names ('**predicate names**') and constants begin with a lower-case letter, and then contain a mixture of upper- and lower-case letters, digits and underscores:

```
professor(allan).  
father_of(allan, joe).
```

- Rules have the form `GOAL :- SUBGOAL1, ..., SUBGOALn`, where the goal and the subgoals each consist of a predicate name and a set of variables and terms, where variables start with an upper-case letter, and then contain a mixture upper- and lower-case letters, digits and underscores.

```
male(X) :-  
    father_of(X, Y).
```

This says ‘One way to prove a goal that **matches** `male(X)` is to find a value of `Y` for which you can prove `father_of(X, Y)`’

You put your facts and rules in a file, and you ask the Prolog interpreter to read that file (weirdly called '**consulting**' it).

Then you can ask a question, where a question is again a predicate letter and a set of constants and variables. A question is a **partial description** of what you want to know.

```
| ?- male(allan).
```

```
| ?- male(X). %% Note: this is NOT the same X as in the rule.
```

Variables are used to say *'I don't care what value you put here, so long as it's the same for every instance of this variable'*.

!!!!!! They are **not** places for temporary storage. You cannot update their values !!!!!!



When you ask a question, Prolog says:

1. Do I have a fact that matches that question?
2. Do I have a rule that would let me prove it?
  - (a) Yes. OK, try to prove its subgoals
  - (b) No. See if I could have done anything differently in the steps leading up to here.

```
mother(janet, sam).  
father(jerry, sam).  
mother(dolly, janet).  
father(frank, janet).
```

```
male(X) :-  
    father(X, _).  
female(X) :-  
    mother(X, _).
```

```
parent(X, Y) :-  
    mother(X, Y).  
parent(X, Y) :-  
    father(X, Y).
```

```
ancestor(X, Y) :-  
    parent(X, Y).  
ancestor(X, Z) :-  
    parent(X, Y),  
    ancestor(Y, Z).
```

Examples in standard interpreter: `mother(X, Y), father(frank, X), ancestor(X, Y), ancestor(A, sam), male(X), (ancestor(X, Y), female(X), female(Y))`

Part 1 ('Do I have a rule that would let me prove it? Yes. OK, try to prove its subgoals') is very much like a function call in any other language. You're asking for something to be done: it may cause you to ask for something else, and when you've done it then you're happy.

So you need a stack to manage it, just like the calling stack in any other language. For Prolog this one is often called the **'goal stack'**

Part 2 ('No. See if I could have done anything differently in the steps leading up to here.') has no counterpart in any other language. To do this, you have to remember where you were in the list of possible rules when you last tried to do this task.

You need another stack to manage it. This one is called the '**backtracking stack**' or the '**choice stack**'.

Examples in single-step interpreter: `mother(X, Y)`, `parent(janet, X)`, `ancestor(X, Y)`

```

| ?- prolog(parent(X, Y)).
Things to do before we have a solution
[parent(_422,_442)]
|:
Next goal is parent(_422,_442)
Could be proved using:parent(_746,_747):-mother(_746,_747)
                        parent(_735,_736):-father(_735,_736)
|:

Things to do before we have a solution
[mother(_422,_442)]
Reasons for doing them and alternative solutions
    parent(_422,_442):parent(_422,_442):-mother(_422,_442)
                        parent(_735,_736):-father(_735,_736)
|:

Next goal is mother(_422,_442)
Could be proved using:mother(janet,sam)
                        mother(dolly,janet)
|:

```

Things to do before we have a solution

[]

Reasons for doing them and alternative solutions

mother(janet,sam):mother(janet,sam)

mother(dolly,janet)

parent(janet,sam):parent(janet,sam):-mother(janet,sam)

parent(\_735,\_736):-father(\_735,\_736)

|:

Things to do before we have a solution

[]

Reasons for doing them and alternative solutions

parent(janet,sam):parent(janet,sam):-mother(janet,sam)

parent(\_735,\_736):-father(\_735,\_736)

|:

X = janet,

Y = sam ? ;

Things to do before we have a solution

[]

Reasons for doing them and alternative solutions

mother(dolly,janet):mother(dolly,janet)

parent(dolly,janet):parent(dolly,janet):-mother(dolly,janet)

parent(\_735,\_736):-father(\_735,\_736)

|:



Things to do before we have a solution

[]

Reasons for doing them and alternative solutions

```
parent(dolly,janet):parent(dolly,janet):-mother(dolly,janet)
                    parent(_735,_736):-father(_735,_736)
```

|:

X = dolly,

Y = janet ?

yes

| ?-

# Equality, unification & assignment

---

Variables are used to say *'I don't care what value you put here, so long as it's the same for every instance of this variable'*.

!!!!!! They are **not** places for temporary storage. You cannot update their values !!!!!!

Sometimes it looks as though they are:

| ?- X = 9.

X = 9 ?

yes

It looks as though I'm assigning 9 to X.

But I'm not: I'm asking whether X **could be** the same 9. If you don't know what X is then it could be the same as anything. This idea is known as '**unification**'—are X and 9 '**unifiable**'?

| ?- 9 = X.

X = 9 ?

yes

| ?- X = 9, X = 10.

no

| ?- X = Y, Y = 9, X = Z.

X = 9,

Y = 9,

Z = 9 ?

yes

You can't do very much with a programming language unless you can represent entities, for which you need datastructures.

Prolog has a number of standard datastructures; it lacks some widely used ones; and it makes it extraordinarily easy to define new ones.

Standard datastructures that it provides: [lists](#), strings, numbers, tuples

Standard datastructures that it lacks: arrays

We'll return to how you can make new datastructures [below](#).

Lists are key to Prolog programming. Very flexible, very useful, and the things you do to work with them provide a pattern for every other datatype.

So what is a list? Very simple datatype, consisting of two parts, called the '**head**' and '**tail**'.

You can make one very easily in Prolog (remember— $A=B$  means ‘are A and B unifiable’):

```
| ?- X = [1 | 2] .
```

```
X = [1|2] ?
```

yes

```
| ?- X = [1 | 2] , X = [A | B] .
```

```
A = 1,
```

```
B = 2,
```

```
X = [1|2] ?
```

yes

(unification has got a bit more interesting: two structures can be unified if they are the same shape and their components are unifiable).

And that's it.

More or less.

Although you know that a list is made of a head and a tail, you normally want to use them as ways of storing lots of things together.

That works if you make the second element a list:

```
| ?- X = [1 | [2 | [3 | 4]]].
```

And that's it.

More or less.

Although you know that a list is made of a head and a tail, you normally want to use them as ways of storing lots of things together.

That works if you make the second element a list:

```
| ?- X = [1 | [2 | [3 | 4]]].
```

```
X = [1,2,3|4] ?
```

```
yes
```



That wasn't quite what we wanted, because the last element (4) has been treated a bit differently: we need to mark the end of the list with something like `null`.

```
| ?- X = [1 | [2 | [3 | [4 | []]]]].  
X = [1,2,3,4] ?  
yes
```

So that's what we want when we're making a list. But that was a clumsy way of writing it, so we write:

```
| ?- X = [1,2,3,4].  
X = [1,2,3,4] ?  
yes
```

You **must** remember that this way of writing it is just a shorthand. A list has **two** elements.

```
| ?- X = [1,2,3,4], X = [HD | TL].  
X = [1,2,3,4],  
HD = 1,  
TL = [2,3,4] ?  
yes
```

Core Prolog doesn't have any control structures: no loops, no conditionals

The only thing you can do is define sets of rules and ask the interpreter to try them out in turn.

**Nothing else**

Having sets of rules for doing the same task is like a conditional.

```
brother(allan, alex).  
brother(alex, janet).  
brother(cain, abel).  
brother(nori, ori).  
brother(ori, gloin).  
...
```

You can show that X is Y's brother if X is Allan and Y is Alex or X is Alex and Y is Janet or ...

But how can we do iteration?

To see if something is a member of a list, you want to iterate through the elements of the list and compare them with your target object.

To see if something is a member of a list, see if it's the first item; and then have a look in the tail.

But how can we do iteration?

To see if something is a member of a list, you want to iterate through the elements of the list and compare them with your target object.

To see if something is a member of a list, see if it's the first item; and then have a look in the tail.

```
member(X, [X | T]).  
member(X, [H | T]) :-  
    member(X, T).
```

(**'singleton variables'**)

Recursion is the **only** way to do iteration in Prolog.

Arithmetic: to do sums, you make up an arithmetic expression like  $2+3*6$ , and tell Prolog to work out what it comes to and unify that with your target.

```
| ?- X is 2+3*6.
```

```
X = 20 ?
```

```
yes
```

```
| ?- 20 is 2+3*6.
```

```
yes
```

```
| ?- 19 is 2+3*6.
```

```
no
```

(difference between `is` and `=`).



To do things that involve counting, you have to do recursion.

```
sum(0, 0).  
sum(I, S) :-  
    J is I-1,  
    sum(J, K),  
    S is I+K.
```

Because recursion is the only way to do iteration in Prolog, Prolog interpreters process it **extremely** efficiently<sup>2</sup>. Don't worry about the idea that iteration is more efficient than recursion. In a decent Prolog compiler it's not.

---

<sup>2</sup>Actually, in many cases the compiler turns it into iteration!

Recursion: there must be a base/termination case and a recursive step. When doing things with lists, the base case is nearly always when the list is empty.

We **very** often want to collect things in one list that satisfy some property into another, maybe changing them along the way.

Collect all the positive numbers in some list together. There are three cases to consider.

1. If you haven't got any numbers at all then you haven't got any positive ones
2. If the first number is positive, you want it and any positive numbers that appear in the tail
3. If not, then you just want the positive numbers that appear in the tail

```

/**
    If you haven't got any numbers at all then you haven't got any
    positive ones
    **/
allPos([], []).

/**
    If the first number is positive, you want it and any positive numbers that
    appear in the tail
    **/
allPos([H | T0], [H | T1]) :-
    H > 0,
    allPos(T0, T1).

/**
    If not, then you just want the positive numbers that appear
    in the tail
    **/
allPos([H | T0], T1) :-
    allPos(T0, T1).

```

(run it, spy it, note stupid backtracking behaviour for later)

Find every element of one list that is also in another.

1. If you haven't got any entities at all then you haven't got any that appear in the second list
2. If the first item is in the second list, you want it and any items in the second list that appear in the tail
3. If not, then you just want the items in the second list that appear in the tail

```

/**
    If you haven't got any entities at all then you haven't got any
    that appear in the second list
    **/
intersect([], _L2, []).

/**
    If the first item is in the second list, you want it and any items
    in the second list that appear in the tail
    **/
intersect([H | T1], L2, [H | LN]) :-
    member(H, L2),
    intersect(T1, L2, LN).

/**
    If not, then you just want the items in the second list that
    appear in the tail
    **/
intersect([_H | T1], L2, LN) :-
    intersect(T1, L2, LN).

```



(run it, spy it, note stupid backtracking behaviour for later)

Prolog rules provide a list of alternative ways of doing something. We saw with the family tree rules that sometimes you want to be able to ask for alternative solutions

- the answer you got may have been AN answer, but it may not have been the one you wanted
- you might have wanted ALL the answers

But sometimes we want to say something more like a case statement: if rule 1 works then that's the answer, if rule 2 works then that's the answer, if rule 3 works then that's the answer, ...

```
freq(F, red) :-  
    F < 480.  
freq(F, orange) :-  
    F < 510.  
freq(F, yellow) :-  
    F < 530.  
...
```

You wouldn't want to get multiple answers to the query

```
| ?- freq(470, COLOR).
```

So you 'cut' the backtracking stack.

```
freq1(F, red) :-
```

```
    F < 480,
```

```
    !.
```

```
freq1(F, orange) :-
```

```
    F < 510,
```

```
    !.
```

```
freq1(F, yellow) :-
```

```
    F < 530,
```

```
    !.
```

```
...
```

Once you've gone past a cut, no other choices in this predicate are available to you.

- Once you've gone past the cut in the first rule, you can't try any of the others. If it's red then it's not orange, or yellow, or . . .
- More confusingly, you also can't go back to any choice points that were saved earlier in this clause.

```
prime(1).  
prime(2).  
prime(3).  
prime(5).  
prime(7).
```

```
firstPrimeAfter(N, P) :-  
    prime(P),  
    P > N.
```

```
firstPrimeAfterWithCut(N, P) :-  
    prime(P),  
    P > N,  
    !.
```

# Prolog programs as Prolog datastructures

---

You can very easily get Prolog to build and execute a Prolog program. Clauses are just standard Prolog datastructures, and you can build them, take them apart, execute them, mix them with clauses that you defined in source code files.

A term can be

- a variable
- a constant (begins with lower case, or included in single quotes)
- a number
- a string (included in double quotes: actually a list of ASCII character codes)
- a functor and a set of terms



You can construct/deconstruct terms

```
| ?- functor(firstPrimeAfter(2, P), F, NARGS).  
F = firstPrimeAfter,  
NARGS = 2 ?
```

```
| ?- functor(X, firstPrimeAfter, 2).  
X = firstPrimeAfter(_A,_B) ?
```

```
| ?- arg(1, firstPrimeAfter(2, P), ARG1).  
ARG1 = 2 ?
```

There are list-based versions of this:

```
| ?- firstPrimeAfter(2, X) =.. L.  
L = [firstPrimeAfter,2,X] ?
```

```
| ?- T =.. [f, 5, X].  
T = f(5,X) ?
```

What happens if we do

```
| ?- functor([1,2,3], F, N).  
F = ' . ',  
N = 2 ?
```

```
| ?- [a,b,c] =.. L.  
L = [' . ',a,[b,c]] ?
```

A clause is just a term whose head is `:-` with **two** arguments—the head and the subgoals.

(it might look as though there are varying numbers of subgoals: but `(A, B)` is just a term whose functor is `,` and whose arguments are `A` and `B`. And `A` and `B` could themselves be terms whose functors are the comma, and ...) )

So I can easily make up a term that looks like a clause:

```
| ?- A = (p :- q,r), functor(A, F, N).  
A = (p:-q,r),  
F = :-,  
N = 2 ?
```

How can I tell Prolog to use it as a rule by adding it to the Prolog **'database'**:

```
| ?- A = (p :- q,r), assert(A).
```

or just

```
| ?- assert((p :- q,r)).
```

(note the extra brackets)

It's there now, just as though I had included it in a source file.

```
| ?- assert((p :- q,r)).
```

```
yes
```

```
| ?- assert(q).
```

```
yes
```

```
| ?- assert(r).
```

```
yes
```

```
| ?- p.
```

```
yes
```

This is **useful** and **quite profound**. You can dynamically change the program itself. You can do something quite like it in Python and JavaScript, you cannot do it in Java, C, C++, C#. In this course I care more about the fact that it's useful—in a course on programming languages I would care that it's profound.

The programs we've seen so far have involved trying to prove that something is true.

Sometimes we want to show that something **isn't** true:

- Make sure that some element of some list is bigger than N
- Make sure that no element of some list is bigger than N

First of these is easy

```
someMemberBiggerThanTarget(N, L) :-  
    member(X, L),  
    X > N.
```

Second is more awkward with what we've seen so far (doable, but awkward). Easy way to do it: ask whether there is something that is bigger than N, **hope to fail**.

```
noMemberBiggerThanTarget(N, L) :-  
    \+ (member(X, L),  
        X > N).
```



Hoping that you can't prove something isn't always the same as proving that it's not true.

```
| ?- orphan(X).  
X = dolly ? ;
```

```
person(X) :-  
    parent(X, _).  
person(X) :-  
    parent(_, X).
```

```
orphan(X) :-  
    person(X),  
    \+ parent(_Y, X).
```

I haven't shown that Dolly is an orphan: I've shown that **as far as I know** she's an orphan. You could come up with new information about her which wouldn't contradict anything I've already said, but which would make me change my mind.

This is an important issue: most of the time, adding new information to what you already know won't make you change your mind about something you've already proved. If you infer that I know lots about language from the fact that I'm professor of linguistics, then when you discover that I'm a terrible tennis player you still believe that I know lots about language.

Frameworks in which adding new knowledge can undermine old conclusions are called '**non-monotonic**'. Prolog's treatment of negation as 'inability to prove' or '**negation as failure**' is probably the first example of non-monotonicity that you'll have seen (monotonic—steadily increasing. Usually, adding new knowledge increases the set of things you can prove, without taking any back: we will see this again later)

We don't actually need to include negation. We can achieve the same result by

```
orphan(X) :-  
    parent(_Y, X),  
    !,  
    1 = 2.  
orphan(X) :-  
    person(X).
```

That would be horrible: you wouldn't want to write that every time you wanted something not to be true. But the idea of a goal, like  $1=2$ , that will always fail can be very useful, so we have a predicate `fail` which does this.

Very weird programming language. I've been promising you that there are problems for which it's the best choice, but you don't believe me.

Here's a problem. Not a very real one, but quite tricky.

Put  $N$  queens on an  $N \times N$  chessboard so that none of them can take each other.

(a queen can take a piece if it's on the same row or the same diagonal or the same column)

						Q			
			Q						
	Q								
								Q	
				Q					
									Q
							Q		
					Q				
		Q							
Q									

Solution: define a position  $X, Y$  as being safe if there is no other queen with position  $X', Y'$  where  $X = X'$  or  $Y = Y'$  or  $abs(X - X') = abs(Y - Y')$ .

Try adding queens to successive rows in such a way that each one is safe

```
member(X, [X | _T]).  
member(X, [_H | T]) :-  
    member(X, T).
```

```
unsafe(q(_X0, Y), QUEENS) :-  
    member(q(_X1, Y), QUEENS).  
unsafe(q(X0, Y0), QUEENS) :-  
    member(q(X1, Y1), QUEENS),  
    D is abs(X0-X1),  
    D is abs(Y0-Y1).
```



```
choosePosition(N, N, _) :-  
    !,  
    fail.  
choosePosition(I, _N, I).  
choosePosition(I, N, K) :-  
    J is I+1,  
    choosePosition(J, N, K).  
  
addQueen(QUEENS, N, N, QUEENS) :-  
    !.  
addQueen(QUEENS0, I, N, QUEENS1) :-  
    choosePosition(0, N, Y),  
    \+ unsafe(q(I, Y), QUEENS0),  
    J is I+1,  
    addQueen([q(I, Y) | QUEENS0], J, N, QUEENS1).
```

| ?- nqueens(6, true, Q).

Q	

Q		
	Q	

Q				
	Q			
		Q		

Q				
			Q	
	Q			
		Q		

Q				
			Q	
	Q			
				Q
		Q		

Q					
	Q				
		Q			

Q					
			Q		
	Q				
		Q			

Q			
	Q		

Q			
		Q	
	Q		

Q				
		Q		
	Q			
			Q	

Q				
		Q		
				Q
	Q			
			Q	

Q					
	Q				
		Q			

Q					
			Q		
	Q				
		Q			

Q				
	Q			

Q				
		Q		
	Q			

Q					
		Q			
	Q				
			Q		

Q					
		Q			
				Q	
	Q				
			Q		

Q					
	Q				

Q					
		Q			
	Q				



Q					
		Q			
			Q		
	Q				

Q					
		Q			
	Q				

Q					
			Q		
		Q			
	Q				

Q	

Q			
	Q		

		Q	
Q			
	Q		

		Q	
Q			
			Q
	Q		

		Q		
Q				
			Q	
	Q			
				Q

Q					
	Q				
		Q			

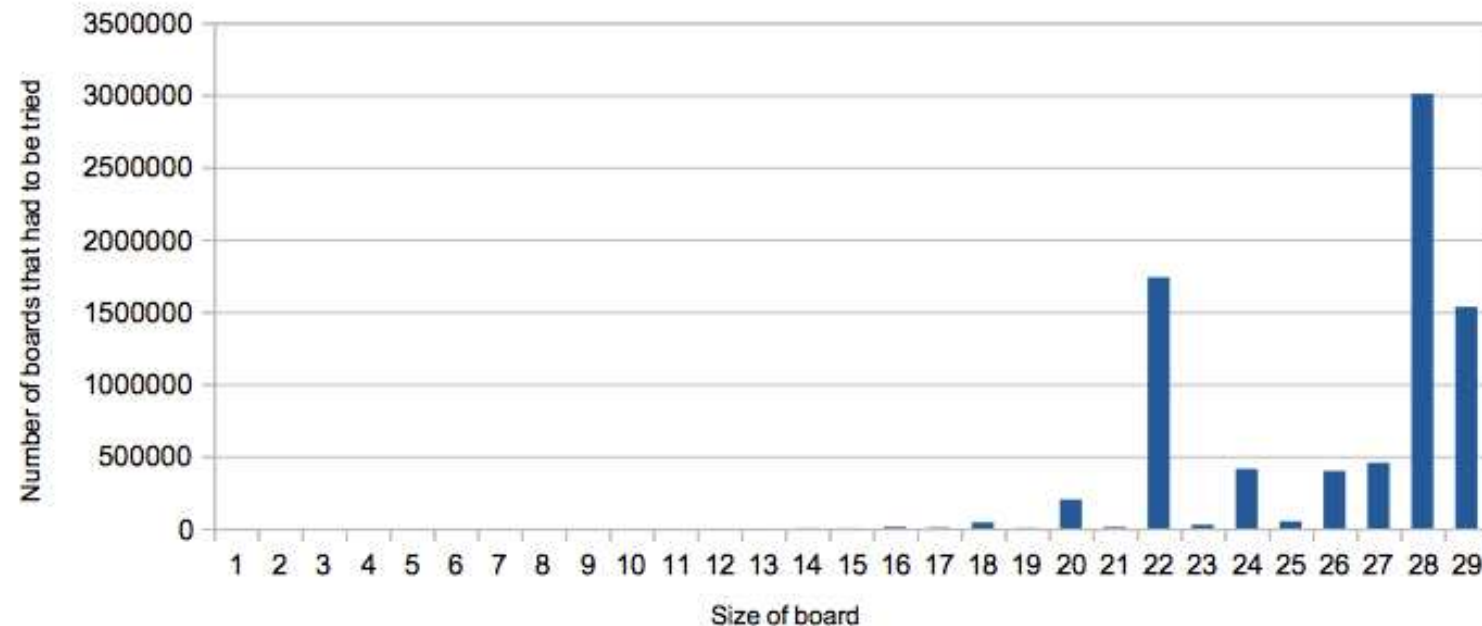
			Q		
Q					
	Q				
		Q			

			Q		
Q					
				Q	
	Q				
		Q			

			Q		
Q					
				Q	
	Q				
					Q
		Q			

(I'll buy the first person who can come up with a solution in Python or Java that is less than twice as long as this a drink (or a chocolate cake))

Complexity is exponential (but odd-sized boards look much easier than even-sized ones). Here's what having exponential complexity means in practice.



1	0	0	8	5	31	9	113	41	102
0	1	2	3	4	5	6	7	8	9
52	261	111	1899	1359	10052	5374	41299	2545	199635
10	11	12	13	14	15	16	17	18	19
8562	1737188	25428	411608	48683	397699	454213	3006298	1532239	
20	21	22	23	24	25	26	27	28	29

I haven't looked for any heuristics, because the point was to see how neatly you can encode the basic algorithm: it's not one that really matters to me, so I haven't put a huge amount of effort into it. No prizes for finding something useful, but I would be interested.



# I REASONING & PLANNING

## Making connections between states of affairs

---

When you first saw me, you knew all sorts of things about me.

- You know I was born before any of you
- You knew that someone believed I knew lots about computer science
- You knew I was going to stand in front of you for an hour and try to persuade you that AI was interesting.

**How did you do that?**

When you first came into the Kilburn Building, you knew all sorts of things about it.

- You knew that you wouldn't get rained on if you were inside it
- You knew there would be lots of people aged between 19 and 25 in it

. . .

**How did you do that?**

You made some observations about the situation you were in.

- Grey-haired man standing at the front of a lecture theatre with a slide saying *'Introduction to Symbolic AI: COMP24412'* on the screen
- large building with *'School of Computer Science'* written on it; . . .

You had access to some **'facts'**

And you knew some things about what the world is like:

- Grey-haired men with creases on their faces are usually in their 60s
- Students are seldom in their 60s
- Large buildings are waterproof

. . .

You had access to some 'rules'

And you were able to link the facts to the rules to get access to implicit information

You were able to do 'inference'

At least some of this is about being able to describe the world, and to exploit patterns in descriptions (**not** patterns in the world).

If I say '*All amblipods have explenditions*' and '*Allan is an amblipod*' then I would hope that everyone in this room would agree that '*Allan has an explendition*' must be true.

At least some of this is about being able to describe the world, and to exploit patterns in descriptions (**not** patterns in the world).

If I say '*All amblipods have explenditions*' and '*Allan is an amblipod*' then I would hope that everyone in this room would agree that '*Allan has an explendition*' must be true.

Even though I've just made the words '*amblipod*' and '*explendition*' up! There's something about the pattern '*All Xs have Ys*' and '*Z is an X*' that lets us infer '*Z has a Y*', no matter what '*X*', '*Y*' and '*Z*' are.



This is one way that people can extract implicit information from facts and rules. The only way???

Logic is an attempt to provide precise ways of specifying facts and rules, and then to study the way they behave.

- How long will it take me carry out some piece of reasoning?
- If I start with accurate descriptions of the world and reliable rules, will all my reasoning lead to true conclusions?
- If something is true, am I guaranteed to be able to prove it?

## [...] provide precise ways of specifying facts and rules

---

Describing facts: there are things, which we can give names to, and they have properties and relations with other things.

**names:** In English we usually use words beginning with upper-case letters as names—‘*Allan*’, ‘*Manchester*’, ...

It's convention. We don't have to do it that way—Arabic doesn't have upper case letters, German uses upper case letters for general nouns.

I'm going to allow any sequence of letters and numbers starting with a lower case letter to be a name: `allanRamsay`, `kilburn2121`, ... I'll call these ‘**constants**’.

**properties and relations:** in natural language we use nouns '*man, house, ...*' and adjectives '*rich, sad, female, ...*' as names for properties, and verbs '*love, give, ...*' and prepositions '*in, on, ...*' for relations.

Not always easy to tell the difference in natural language between nouns, verb & adjectives:

- (1) a. When you've been for a long run, you should have a **drink** of water. (noun)
- b. When you've been for a long run, you should **drink** some water. (verb)
- c. Lots of people carry a **drink** bottle with them when they go for a long run. (adj)

I'm going to allow any sequence of letters and numbers starting with a lower case letter with a numerical superscript to denote a property or a relation:  $\text{man}^1$ ,  $\text{rich}^1$ ,  $\text{love}^2$ ,  $\text{give}^3$ , ... I'll call these '**predicates**'.

**formulae:** I don't want to just be able to name things and relations. I want to be able to say that entities have properties, or that entities are related to other entities:

- (2) a. John is sad. (property)
- b. John loves Mary. (relation)

I'm going to do that by writing a property/relation name, and then the right number of things in brackets. The things in brackets are the '**arguments**' of the property/relation.

$\text{sad}^1(\text{john})$ : John '**satisfies**' the property 'sad', 'sad' '**holds of**' John

$\text{love}^2(\text{john}, \text{mary})$ : John and Mary '**satisfy**' the relation 'love', 'love' '**holds of**' John and Mary.

And then like everyone else I'll get lazy and omit the superscripts. I need them in principle because the  $M$ -place version won't mean exactly, or even roughly, the same as the  $N$ -place one:

- (3) a. John is running very fast. (1-place property)
- b. John is running a small toyshop. (2-place relation)

But if I write it with one argument, then surely I mean the 1-place version, and if I write it with two arguments then surely I mean the 2-place one.

I don't just say simple positive statements. I say that lots of things are true, I say that one out of a set of possible options is true, I say that things are false, ...

- (4) a. '**Conjunction**': John loves Mary and Peter loves Mary.  
b. '**Disjunction**': Mary will marry John or he will run away to America.  
c. '**Negation**': Mary doesn't love John.  
d. '**Implication**': John will be happy if Mary marries him.<sup>3</sup>  
e. '**Universal quantification**': Everyone loves Mary.  
f. '**Existential quantification**': John will marry someone.

---

<sup>3</sup>Probably not true, since she doesn't love him.

Given these ways of combining simple statements, you can say quite complicated things. I've just told a rather sad little story using them.

Natural language has some other ways of putting things together.

But the goal of logic was to 'provide precise ways of specifying facts and rules, and then to study the way they behave.'

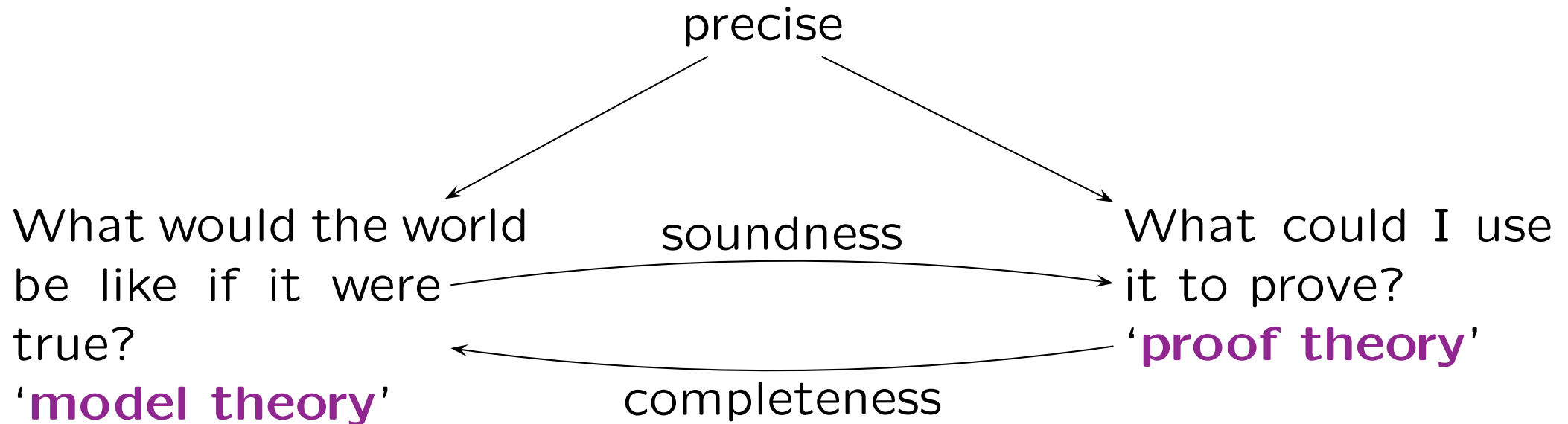
And it's hard to make some of the other things that you can do in natural language precise. So logic often stops here (we will revisit this).



- (5) a. Conjunction: `love(john, mary) & love(peter, mary)`  
b. Disjunction: `marry(mary, john) or runAwayTo(john, america)`  
c. Negation: `not(marry, john)`  
d. Implication: `marry(mary, john) => happy(john)`  
e. Universal quantification: `forall(X, love(X, mary))`  
f. Existential quantification: `exists(X. marry(john, X))`

1. People often write weird symbols for this:  $\& = \wedge$ , `forall` =  $\forall$ . I'm going to want to put these things in programs, so I want simple ASCII characters, not  $\text{\LaTeX}$  symbols; and anyway the weird symbols are weird.
2. My logical formulæ are **much simpler** than my natural language sentences. No tense, made-up relations like `runAwayTo`. We will revisit this as well.

[...] provide precise ways of specifying facts and rules



Logic wouldn't be much use to us if you couldn't use it for describing the world. So model theory seems pretty important.

But I want to be able to extract implicit information. So I need proof theory as well.

**A** logic is a description of a language for which I have both a model theory (so I can describe the world) and a proof theory (so I can reason with my description).

I don't want to do proofs that lead me from true statements about the world to false ones: I want my logic to be '**sound**'

If my goal is always true whenever my assumptions are, I'd like to be able to prove it from them: I'd like my logic to be '**complete**'

A simple-minded way of thinking about properties and relations: start by thinking about the sentence

(6) Everyone who is called Allan Ramsay is descended from Rob Roy.

‘called Allan Ramsay’ denotes a property: I might write this in my logic as

`forall(X, calledAllanRamsay(X) => descendedFrom(X, robRoy))`

The people who satisfy this property don't have anything else in common. They don't all look alike, or weigh the same, or have the same accent. They're just a bunch of people. I might say that called `AllanRamsay` **is** this set of people.

The people who satisfy this property don't have anything else in common. They don't all look alike, or weigh the same, or have the same accent. They're just a bunch of people. I might say that `calledAllanRamsay` **is** this set of people.

That was sort of alright in this case. But I'm going to extend this to cover **every** property and relation.

`brother` = { (Cain, Abel), (Abel, Cain), (Loki, Helblindi), (Helblindi, Loki), (Allan, Janet), ~~(Janet, Allan)~~, ... }

`rich` = { Bill Gates, Alan Sugar, the Queen, ~~Allan~~, ... }

Very unsatisfying. But very precise. `brother(janet, allan)` is true if and only if (iff) the pair of people denoted by the names Janet and Allan is in the set of brothers.

The people denoted by the names Janet and Allan? There are lots of people called Janet and Allan in the world. So there isn't 'a pair of people called Janet and Allan', there are lots of such pairs.

I have to say **which** people called Janet and Allan I mean.



A ‘**model**’ consists of a set of things, and a set of sets of tuples:

A set of things: the people in this room

A set of sets of tuples:  $S1 = \{(me)\}$ ,  $S2 = \text{everyone else}$

An ‘**interpretation**’ consists of an assignment of things to constants, and sets of tuples to predicates.

`allanRamsay = me`, `diegoAbelGarcia=???`, ...

`professor = S1`

`student = S2`

# Truth in an interpretation

---

It's (fairly) common practice to write  $\llbracket a \rrbracket_I$  to denote the individual (set of tuples) that some interpretation  $I$  assigns to the constant (predicate)  $a$ .

We want to think about the circumstances under which  $I$  makes a sentence  $S$  true ( $I$  is a model of  $S$ ,  $S$  satisfies  $I$ ,  $I$  satisfies  $S$ ,  $I \models S$ ).

**Simple sentence:**  $I \models \text{love}(\text{john}, \text{mary})$  iff  $(\llbracket \text{john} \rrbracket_I, \llbracket \text{mary} \rrbracket_I)$  is a member of  $\llbracket \text{love} \rrbracket_I$ .

## Basic combinations:

- $I \models P \ \& \ Q$  iff  $I \models P$  and  $I \models Q$
- $I \models P \text{ or } Q$  iff  $I \models P$  or  $I \models Q$
- $I \models \text{not}(P)$  iff not  $I \models P$
- $I \models P \Rightarrow Q$  iff  $I \models \text{not}(P)$  or  $I \models Q$  (yuk)

## Quantifiers:

- $I \models \text{exists}(X, P(X))$  iff  $\llbracket P(a) \rrbracket_I$  for some constant  $a$ . Note that there could be things in the model that aren't named by any constant. In that case, not  $I \models \text{exists}(X, P(X))$ .
- $I \models \text{forall}(X, P(X))$  iff  $\llbracket P(a) \rrbracket_I$  for every constant  $a$ .

All very clear & precise, even if some bits don't seem very satisfactory. We can use our language for describing the world, which is pretty important.

But in this course I'm actually primarily interested in how we can do things with it. The most elegant language in the world wouldn't be any use to me unless I could write programs that could do inference in it.

So I'm going to introduce the proof theory via a program. A proof is what my program does with the facts and rules at its disposal. I want it to be sound, of course, but I also want it to be implementable. So I will define it by the implementation.

- We have facts:

```
professor(allan) .  
pest(danny) .  
father(allan, steven) .
```

- And we have rules where all variables are universally quantified:

```
professor(X) => clever(X)  
father(X, Y) & professor(X) => nice(Y)
```



That's a bit awkward, because I do want existential quantifiers.

Two men came into the bank. One of them started shouting and the other was waving a gun around. The first one went over the cashiers, while the second fired his gun at the ceiling. Then the second one fired a shot at the alarm, while the first got out a bag with SWAG written on it.

Two men came into the bank. One of them, call him A, started shouting and the other, call him B, was waving a gun around. A went over the cashiers, while B fired his gun at the ceiling. Then B fired a shot at the alarm, while A got out a bag with SWAG written

So long as there's no-one else called A and B in the story, you won't get muddled up. You get this in places like court cases involving undercover security agents, vulnerable witnesses, . . . Make up a name when you need to introduce a new person into what you're doing.

Slightly awkward if the existential quantifier is inside a universal.

(7) I have a mother.

`exists(X, mother(allan, X)).`

`mother(allan, dolly).`

`mother(allan, sk17).`

(8) Every human as a mother.

`forall(X, human(X) => exists(Y, mother(X, Y))).`

`forall(X, human(X) => mother(allan, dolly)). !!!!!`

`forall(X, human(X) => mother(X, mum(X))).`

`forall(X, human(X) => mother(X, sk26(X))).`

These kinds of made-up names are called '**Skolem constants**' and '**Skolem functions**'.

It's extremely easy to implement quite an efficient engine to apply these rules.

1. To prove  $P$ , see if it's a fact.
2. If it's not a fact, look for a rule of the form  $LHS \Rightarrow P'$ , where  $P'$  MATCHES  $P$ .

Prove each element of the left-hand side.

Prove nice(steven)

Is it a fact? No.

Is there a rule whose right-hand side matches it?

Yes:  $\text{father}(X, Y) \ \& \ \text{professor}(X) \Rightarrow \text{nice}(Y)$

Matching it binds  $Y$  to steven, so now we have to prove  $\text{father}(X, \text{steven}) \ \& \ \text{professor}(X)$ .

Prove `father(X, steven)`.

Is it a fact? (or more precisely, is there a fact that MATCHES it?)

Yes—`father(allan, steven)`. Matching that bound `X` to `allan`.

So now we have to prove `professor(allan)`. . . .

Dead useful, dead easy. So much so that it got turned into a programming language: use `,` instead of `&`, `;` instead of `or`, write rules as `P :- Q` rather than `Q => P` and do your inference like this and you've got Prolog.

```
professor(allan).  
pest(danny).  
father(allan, steven).  
clever(X) :- professor(X).  
nice(Y) :- father(X, Y), professor(X).
```



The bit that you can turn into Prolog has conjunction (and) and disjunction (or) in the antecedent of a rule, but only conjunction in the consequent. You can write

$$p(X) \text{ :- } q(X); r(X).$$

but not

$$q(X); r(X) \text{ :- } p(X).$$

First-order logic with no disjunction in the consequent is called '**Horn-clause logic**' (or just Horn logic).

But Horn logic (especially the particular version used in Prolog) has a couple of problems.

Suppose you had a rule like

```
(father(X, Y) & professor(X))  
    => (nice(Y) or spoilt(Y))
```

It would be hard to know whether you could use it for proving `nice(steven)`,  
because you'd have to show that he wasn't spoilt.

It's hard to deal properly with negation. Prolog doesn't. In raw Prolog,  $\text{not}(P)$  and 'I can't prove  $P$ ' are taken to be the same ('**negation as failure**').

(so standing here I would be able to say *'It's not raining'*).

This is surprisingly tricky to overcome: one way is to say that *'It's not raining'*  $\equiv$  *'If it's raining I'm a Dutchman'*, *'She'll never marry you'*  $\equiv$  *'She won't marry you until pigs can fly'*, . . . . In our logic we'll say  $\text{not}(Q) \equiv (Q \Rightarrow \text{absurd})$ . So reasoning about negation is just (!) reasoning about implication.

So how do we prove things like  $p \Rightarrow q$  (and in particular, how do we prove things like `spoilt(steven)  $\Rightarrow$  absurd`).

One way: see whether adding the left-hand side to everything you know lets you prove the right-hand side (this is called '**constructive implication**').

1. To prove  $P$ , see if it's a fact.
2. If it's not a fact, look for a rule of the form  $LHS \Rightarrow P'$ , where  $P'$  MATCHES  $P$ .  
Prove each element of the left-hand side.
3. To prove  $P \Rightarrow Q$ , add  $P$  to the set of facts (assert it) and see if you can now prove  $Q$ . Remember to remove  $P$  again afterwards (retract it).

Last one: disjunction in the antecedent is easy to deal with.

$$(\text{hasBand}(X) \text{ or } \text{hasAthletics}(X)) \ \& \ \text{father}(Y, X) \\ \Rightarrow \text{giveLift}(Y, X),$$

`hasBand(danny).`

`hasAthletics(louisa).`

`father(allan, danny).`

`father(allan, louisa).`

To prove  $(\text{hasBand}(X) \text{ or } \text{hasAthletics}(X))$  you just have to prove one of them.

But disjunction in the consequent (or in a fact) is much harder.

$(\text{hasBand}(X) \text{ or } \text{hasAthletics}(X)) \ \& \ \text{father}(Y, X)$   
 $\Rightarrow \text{giveLift}(Y, X),$   
 $\text{hasAthletics}(\text{joe}) \text{ or } \text{hasBand}(\text{joe}).$

4. To prove that  $P$  follows from  $A$  or  $B$ , show that it follows from each of them.

And that's my favourite theorem prover (or pretty close to it). This one's called 'model generation' (Manthey and Bry 1988). There are loads of others. I like this one because the basic form is so easy to implement in Prolog that adding other weird things to it is manageable.

```
%% First bit is just Prolog: better, in fact, to convert
%% your representation to Prolog, but I'm doing it
%% explicitly here so we can see what's going on.
```

```
prove(X) :-
    fact(X).
```

```
prove(X) :-
    LHS => X,
    prove(LHS).
```

```
prove(A & B) :-
    prove(A),
    prove(B).
```

```
prove(A or B) :-
    prove(A); prove(B).
```



That's an implementation of backward chaining for Horn logic.

Backward chaining  $\approx$  linear input resolution

Horn logic  $\approx$  Prolog.

So it's also an implementation of Prolog in Prolog.

%% And here's the non-Horn stuff

```
prove(X) :-  
    prove(P or Q),  
    (P => X),  
    (Q => X).
```

```
(P => Q) :-  
    assert(P),  
    (prove(Q) ->  
        retract(P);  
        (retract(P), fail)).
```

What can go wrong?

- Infinite loops in the Horn part:

$$p(X) \Rightarrow q(X).$$

$$q(X) \Rightarrow p(X).$$

- Infinite loops in the disjunctive part:

$$p(X) \Rightarrow (p(f(X)) \text{ or } q(X))$$

**You can block both kinds of loop.**

You can catch all such loops. But it might cost you some things that you could prove without the loop catcher.

```

%% Catch loops in the backward chaining part
%% Keep track of what you're already doing, check it
%% every time you start a new step
%% (like the second version of the route finder in lab1)
prove(X, _TRAIL) :-
    fact(X).

prove(X0, TRAIL) :-
    \+ (member(X1, TRAIL), like(X0, X1)), %% definition of 'like'???
    LHS => X,
    prove(LHS, [X | TRAIL]).

prove(A & B, TRAIL) :-
    prove(A, TRAIL),
    prove(B, TRAIL).

prove(A or B, TRAIL) :-
    prove(A, TRAIL); prove(B, TRAIL).

```

```
%% And here's the non-Horn stuff
```

```
prove(X, TRAIL) :-  
    prove(P or Q, TRAIL),  
    (P => X, TRAIL),  
    (Q => X, TRAIL).
```

```
(P => Q, TRAIL) :-  
    \+ (member(A => B, TRAIL), like(P => Q, A => B)),  
    assert(P),  
    (prove(Q) ->  
        retract(P);  
        (retract(P), fail)).
```

You can give definitions of `like` that will block any loops that could arise. But if you do that you'll rule out some legal proofs.

You can give definitions of `like` that block some loops and allow all legal proofs.

But you can't block all loops **and** allow all proofs. The sensible thing to do is to block obvious loops, and use a counter (like in the first version of the route finder) to stop very deep proofs. You've got an implicit counter anyway—the amount of memory on your machine:

```
p :- q.
```

```
q :- p.
```

I don't just want plain first-order logic: I want defaults (*'Most swans are white'*, *'Bruce is a swan'  $\vdash$  'Bruce is white'*), modalities (*'Bruce might be white'*, *'Bruce must be white'*), time, knowledge & belief, ...

I can do some of this by extending the idea from the last version of the engine that I can watch over the proof process and take notes. Keep all the things I want to monitor in a **'label'** (Gabbay 1996).



```
%% The TRAIL is just one bit of information that I want to key an
%% eye on during the proof: stick them all in a 'label'
prove(X, _LABEL) :-
    fact(X).

prove(X0, label(TRAIL, ...)) :-
    \+ (member(X1, label(TRAIL, ...)), like(X0, X1)), %% definition of 'like'???
    LHS => X,
    prove(LHS, label([X | TRAIL], ...)).

prove(A & B, label(TRAIL, ...)) :-
    prove(A, label(TRAIL, ...)),
    prove(B, label(TRAIL, ...)).

prove(A or B, label(TRAIL, ...)) :-
    prove(A, label(TRAIL, ...)); prove(B, label(TRAIL, ...)).
```

```
prove(X, label(TRAIL, ...)) :-  
    prove(P or Q, label(TRAIL, ...)),  
    (P => X, label(TRAIL, ...)),  
    (Q => X, label(TRAIL, ...)).
```

```
(P => Q, label(TRAIL, ...)) :-  
    \+ (member(A => B, label(TRAIL, ...)), like(P => Q, A => B)),  
    assert(P),  
    (prove(Q) ->  
        retract(P);  
        (retract(P), fail)).
```

Defaults (proof theory): you can use a rule about  $\text{most}(X :: \{p(X)\}, q(X))$  to infer  $q(a)$  from  $p(a)$  unless you have reason not to.

Use default rules freely until you've finished the proof. Then make sure that what you've used doesn't contradict things you actually know, and also that it isn't self-contradictory (Reiter 1980).

(Defaults (model theory):  $\text{most}(X :: \{p(X)\}, q(X))$  is true if more than half the things that satisfy  $p$  also satisfy  $q$ ,  $|p \cap q| > 0.5 \times |p|$ .)

Won't work for infinite sets (*'The gaps between most pairs of prime numbers are greater than 100'*), very difficult to connect to the proof theory above.)

```

prove(P) :-
    prove(P, label([], []/DEFAULTS)),
    \+ prove(DEFAULTS => absurd, [], []/[]).

prove(X0, label(TRAIL, DEF0/DEFN)) :-
    \+ (member(X1, label(TRAIL, TRAIL)), like(X0, X1)), %% definition
    LHS => X,
    prove(LHS, label([X | TRAIL], DEF0/DEFN)).

...
%% If you need to use a default rule, remember it
%% for the end of the proof
prove(P, label(TRAIL, DEF0/DEF1)) :-
    most(X, Q => P),
    prove(Q, label(TRAIL, [P | DEF0], DEF1)).

```

The original version of `prove/2` could get into loops, and we can't possibly detect every loop.

But this one involves a call of

```
\+ prove(DEFAULTS => absurd, [], []/[]).
```

This could in turn get caught in a loop; this is worse. If I really wanted to I could get my inference engine to enumerate all provable formulae, and just wait till the answer turned up (in fact the original presentation of this algorithm did exactly that). But suppose it never turns up (which is what I want here). Then I'll wait forever.

- Why do I care about inference?
- Basic model generation algorithm: do Horn clauses backwards, just like Prolog; do non-Horn clauses forwards, showing that each disjunct leads to a proof.
- ‘Constructive’ treatment of negation by  $\text{not}(p) \equiv (p \Rightarrow \text{absurd})$
- Labels: annotating your proof with extralogical information—useful for loop-checking and for extensions beyond simple first-order logic.

- The world is not as you would like it to be.
- You are capable of acting
- How do you change it to be how you would like it?



## Tasks:

- Describing goals:
  - notation for describing the world (first-order logic ???)
  - goals: partial description.
- Describing actions:
  - describing effects. What do we do about negative effects.  
What do we do about conservative effects?
  - describing preconditions.
  - describing durations, motor actions, . . .

Planning in general is unmanageable. Choose a simpler version (STRIPS (Fikes and Nilsson 1971)).

## 1. **The world:**

- (a) Finite set of facts. No rules, no disjunction, no negations.
- (b) Closed world assumption (CWA): if it's not known to be true it must be false.

2. **Goal states:** also a list of facts. But now only partial, so no CWA.

3. **Actions:** All that matters is (a) how they change the world, and (b) when they can be performed. In particular, no duration, no motor control.

Planning in general is unmanageable. Choose a simpler task.

You can plan to retire by the age of 40, you can plan to get me to make you a cup of tea, you can plan to persuade me that Gurdjieff was right, . . .

You can plan to pick up a wooden block and put it somewhere else. Toy problems or experimental simplification???

## Representing the world:

- testing whether something is true
- testing whether something is false
- testing whether there is **anything** that satisfies a given property
- testing whether **everything** satisfies a given property
- changing your mind (imagining what would happen if you performed some action, backtracking when you realise that's wrong)

Representing the world: as a collection of facts, as a simulation, . . .

- A collection of facts: (quite) easy to set up, flexible, coarse-grained
- A simulation: set of purpose-built datastructures and a direct mapping between (parts of) the world and (parts of) the set of datastructures (e.g. a 3D array where putting the letter  $A$  at point  $\langle x, y, z \rangle$  means that the corresponding point in the real world is occupied by some part of the block  $A$ ). As fine-grained as you like, might be hard to set up, likely to be hard to inspect and manipulate

Not a bad idea to have both: but we'll concentrate on the first for now.

A collection of facts: list, doubly-linked list, hash table, association list, . . .

- list: easy to add facts, slightly awkward deleting them, easy to see if something's true (just do membership test), very easy to keep different contexts. Seeing if it's false: assume that it's false if you can't show that it's true, or explicitly look to see that its negation is true?
- hash table: easy to add and delete facts, easy to see if something's true, much harder to maintain contexts. Same issues about finding out whether something is false.

- The Prolog database provides a combination of hash table and list: you get directly to a list containing all the facts with a given predicate. Might be a good bet, but the task of maintaining different contexts might outweigh its other advantages. Wait and see.

Representation of action schemas: all that matters is (a) how an action can change the world, and (b) when it can be performed. In particular, no duration, no motor control.

How can an action change the world? By making things become true or false. Making things false: add `not(on(a, b))`, or delete `on(a, b)`?

When can an action be performed: when the right things are true or false.

Falsity: the classic solution is to say that anything which is not known to be true is false – the ‘**closed world assumption**’. **It’s a hack.** People try to dress it up, but basically it’s a hack.



```
action(<name>,  
      pre(<P1, ..., Pn>),  
      add(<A1, ..., Aj>),  
      delete(<D1, ..., Dj>))
```

- The '**name**' is just there for the programmer. The system doesn't use it.
- The '**preconditions**' are a list of propositions that are required to be true or false. Do we allow negations, disjunctions, quantification in here?

If we **don't** allow negation then we may need antonyms: I can get round the fact that I can't have '*not(John knows that Mary loves him)*' by using '*John is ignorant of the fact that Mary loves him*'. These 'duals' can be maddening, but if you don't have negation then you can't avoid them.

- The ‘**add**’ list is a list of things which will become true when this action is performed. Even I don’t want disjunctions and quantification here. The effects of an action have to be concrete, or it’s not an action.
- The ‘**delete**’ list is a list of things that will become false when the action is performed. Using a delete list goes hand-in-hand with assuming that anything that is not known is false. In order to **make** something false, simply remove it from the list of things that are true.

If you’re using duals, then if you are adding the positive one you’ll have to retract the negative, and vice versa.

# Blocks World Example

---

Initial state:

`[on(a, b), on(b, table), on(c, table),  
clear(a), clear(c), handempty]`

Goal state: `[on(b, c)]`

Where's a? Is the hand empty? Where, indeed, is c?

```
action(raise,  
      pre([low]),  
      add([high]),  
      delete([low]))
```

```
action(lower,  
      pre([high]),  
      add([low]),  
      delete([high]))
```

```
action(move(X, Y),  
      pre([high, over(X)]),  
      add([over(Y)]),  
      delete([over(X)]))
```

```
action(grasp(X),  
    pre([handempty, low, over(X), clear(X), on(X, Y)]),  
    add(holding(X), over(Y), clear(Y))),  
    delete([handempty, over(X), on(X, Y)]))
```

```
action(ungrasp(X),  
    pre([holding(X), over(Y), low, clear(Y)]),  
    add(handempty, on(X, Y), over(X)]),  
    delete(holding(X), over(Y), clear(Y)]))
```

## Backward chaining algorithm

- 1 Choose an unsatisfied goal  
(non-deterministic)
  - 1.1 Find a member of the current list of goals which isn't a member of the current description of the world  
(non-deterministic)

That's OK so long as the goal list is simply a set of fully instantiated facts. So in the example, the only goal is `on(b, c)`. This is not a member of the list of facts that characterise the world, so we assume that it is unsatisfied

Non-deterministic: there might have been other goals which weren't currently true. We might have been better advised to choose one of them instead. So we'll have to **remember** the one we chose, and possibly **backtrack** to choose another.

- 1 Choose an unsatisfied goal  
(non-deterministic)
- 2 Find an action that brings it about  
(non-deterministic)
  - 2.1 Find one whose effects contain it.



So in the current case, with `on(b, c)` as our chosen goal, we find that

```
action(ungrasp(b),  
      pre([holding(b), over(c), low, clear(c)]),  
      add(handempty, on(b, c), over(b)]),  
      delete(holding(b), over(c), clear(c)]))
```

will do what we want. Or do we? How did `x` and `y` turn into `b` and `c`? The action was an action **schema**, not an action. We need some way of matching variables and constants, and of instantiating them. Use unification (Prolog does this for free).

Just include actions as written above as Prolog 'facts'. Then to find an action which will achieve a given goal, do

```
find_action(G, action(NAME, pre(PRE), add(ADD), delete(DELETE))) :-  
    action(NAME, pre(PRE), add(ADD), delete(DELETE)),  
    member(G, ADD).
```

So in the example

```
find_action(on(b, c), A).
```

```
A = action(ungrasp(b),  
           pre([holding(b), over(c), low, clear(c)]),  
           add([handempty, on(b, c), over(b)]),  
           delete([holding(b), over(c), clear(c)])) ?
```

- 1 Choose an unsatisfied goal  
(non-deterministic)
- 2 Find an action that brings it about  
(non-deterministic)
- 3 Check its preconditions
  - 3.1 If they're all true, perform it
  - 3.2 Otherwise add the ones that aren't to  
the list of goals and go to 1

Perform it?

Perform it in your internal model. That's all you can do when you're **making** the plan

To perform an action, you add its add-list to the current world and delete its delete-list.

But in practice you have to run the plan in the real world. Where you will discover that your initial model was wrong, that the action description was wrong, and that I've moved your block since you last looked (the '**frame problem**': most planning algorithms assume that nothing changes unless the planner changes it. Very similar to the '**negation-as-failure**' treatment of negation in Prolog).

Observing the world is also an action. Your robot may have a camera on it, permanently monitoring the scene. But you've got to weave the interpretation of the visual input into the planning algorithm. And unless the camera has a  $360^\circ$  field of view, you may have to include actions for controlling where it should point.

If you're going to do that, then you're anticipating the potential for including actions whose preconditions refer to what you know, rather than what is true; and actions whose effects concern obtaining information about the world.

Otherwise add the ones that aren't to the list of goals?

Fair enough. But where should we add them: on the front, on the back, ... ?

Why are we adding them? Because they are preconditions for doing the next action. Why do we want to perform this action. Because it will bring about our current goal. So we need to deal with **its** preconditions before we think about anything else. Better to think about the planner as follows:

- 1 Choose an unsatisfied goal  
(non-deterministic)
- 2 Find an action that brings it about  
(non-deterministic)
- 3 Make a sub-plan to ensure that its preconditions  
are satisfied.
- 4 Perform it (you can, since its preconditions  
are satisfied). Go to 1.

```

plan(GOALS, WORLD0, PLAN) :-
    %% dummy initial action to get started
    plan(action(dummy, pre([]), add([]), delete([])),
          GOALS, WORLD0, _WORLD1, [], PLAN).

plan(action(NAME, PRE, add(ADD), delete(DEL)), GOALS, W0, W1, PLAN, [NAME | PLAN]) :-
    all_true(GOALS, W0),
    perform(action(NAME, PRE, add(ADD), delete(DEL)), W0, W1).

plan(ACTION, GOALS, W0, W2, PLAN0, PLAN2) :-
    choose(GOALS, W0, G),
    find_action(G, action(NAME, pre(PRE), add(ADD), delete(DEL))),
    plan(action(NAME, pre(PRE), add(ADD), delete(DEL)), PRE, W0, W1, PLAN0, PLAN1),
    plan(ACTION, GOALS, W1, W2, PLAN1, PLAN2).

all_true([], _W).
all_true([H | T], W) :-
    member(H, W),
    all_true(T, W).

```



And here it is in action (output is in reverse order—easier that way):

```
?- plan([on(b, c)], [on(b, a), on(c, table), handempty, high, over(b), clear(a), clear(b), clear(c)], P).
```

```
on(b,c) is not true
```

```
Chosen ungrasp(b) to help
```

```
holding(b) is not true
```

```
Chosen grasp(b) to help
```

```
low is not true
```

```
Chosen lower to help
```

```
[high] all true: performing lower
```

```
Before: [on(b,a),on(c,table),handempty,high,over(b),clear(a),clear(b),clear(c)]
```

```
After: [low,on(b,a),on(c,table),handempty,over(b),clear(a),clear(b),clear(c)]
```

```
[clear(b),over(b),handempty,low,on(b,a)] all true: performing grasp(b)
```

```
Before: [low,on(b,a),on(c,table),handempty,over(b),clear(a),clear(b),clear(c)]
```

```
After: [holding(b),over(a),clear(a),low,on(c,table),clear(a),clear(b),clear(c)]
```

```
over(c) is not true
```

```
Chosen move(_4775,c) to help
```

```
high is not true
```

```
Chosen raise to help
```

```
[low] all true: performing raise
```

```
Before: [holding(b),over(a),clear(a),low,on(c,table),clear(a),clear(b),clear(c)]
```

```
After: [high,holding(b),over(a),clear(a),on(c,table),clear(a),clear(b),clear(c)]
```

```
[high,over(a)] all true: performing move(a,c)
```

```
Before: [high,holding(b),over(a),clear(a),on(c,table),clear(a),clear(b),clear(c)]
```

```
After: [over(c),high,holding(b),clear(a),on(c,table),clear(a),clear(b),clear(c)]
```

low is not true  
Chosen lower to help

[high] all true: performing lower

Before: [over(c),high,holding(b),clear(a),on(c,table),clear(a),clear(b),clear(c)]

After: [low,over(c),holding(b),clear(a),on(c,table),clear(a),clear(b),clear(c)]

[holding(b),over(c),low,clear(c)] all true: performing ungrasp(b)

Before: [low,over(c),holding(b),clear(a),on(c,table),clear(a),clear(b),clear(c)]

After: [handempty,on(b,c),over(b),low,clear(a),on(c,table),clear(a),clear(b)]

[on(b,c)] all true: performing dummy

Before: [handempty,on(b,c),over(b),low,clear(a),on(c,table),clear(a),clear(b)]

After: [handempty,on(b,c),over(b),low,clear(a),on(c,table),clear(a),clear(b)]

P = [dummy,ungrasp(b),lower,move(a,c),raise,grasp(b),lower] ?

That was very hard work: to achieve A and B, find some way of achieving A and then find some way of achieving B: but because of the generic nature of the actions schemas, we can re-use it in other situations.

Can't we?

Two minimal blocks world actions:

```
action(unstack(A, B),  
      pre([on(A, B)]),  
      add([on(A, table), clear(B)]),  
      delete([on(A, B)])).
```

```
action(stack(A, C),  
      pre([on(A, table), clear(A), clear(C)]),  
      add([on(A, C)]),  
      delete([on(A, table), clear(C)])).
```

and two very simple tasks:

task9(P):-

```
    plan([on(b, c), on(a, b)],  
          [on(a, c), clear(b), clear(a), on(b, table)],  
          P).
```

task10(P):-

```
    plan([on(a, b), on(b, c)],  
          [on(a, c), clear(b), clear(a), on(b, table)],  
          P).
```

```

| ?- task9(P).
on(b,c) is untrue in [on(a,c),clear(b),clear(a),on(b,table)]
stack(b,c) would make it true
  clear(c) is untrue in [on(a,c),clear(b),clear(a),on(b,table)]
  unstack(_1111,c) would make it true
    All true: [on(a,c)]
    Performing unstack(a,c)
    Before: [on(a,c),clear(b),clear(a),on(b,table)]
    After:  [on(a,table),clear(c),clear(b),clear(a),on(b,table)]
    All true: [on(b,table),clear(b),clear(c)]
    Performing stack(b,c)
    Before: [on(a,table),clear(c),clear(b),clear(a),on(b,table)]
    After:  [on(b,c),on(a,table),clear(b),clear(a)]
on(a,b) is untrue in [on(b,c),on(a,table),clear(b),clear(a)]
stack(a,b) would make it true
  All true: [on(a,table),clear(a),clear(b)]
  Performing stack(a,b)
  Before: [on(b,c),on(a,table),clear(b),clear(a)]
  After:  [on(a,b),on(b,c),clear(a)]
All true: [on(b,c),on(a,b)]
Performing dummy
Before: [on(a,b),on(b,c),clear(a)]
After:  [on(a,b),on(b,c),clear(a)]
P = [dummy,stack(a,b),stack(b,c),unstack(a,c)] ?

```

```
| ?- task10(P).
on(a,b) is untrue in [on(a,c),clear(b),clear(a),on(b,table)]
stack(a,b) would make it true
  on(a,table) is untrue in [on(a,c),clear(b),clear(a),on(b,table)]
  unstack(a,_1093) would make it true
    All true: [on(a,c)]
    Performing unstack(a,c)
    Before: [on(a,c),clear(b),clear(a),on(b,table)]
    After:  [on(a,table),clear(c),clear(b),clear(a),on(b,table)]
    All true: [on(a,table),clear(a),clear(b)]
    Performing stack(a,b)
    Before: [on(a,table),clear(c),clear(b),clear(a),on(b,table)]
    After:  [on(a,b),clear(c),clear(a),on(b,table)]
on(b,c) is untrue in [on(a,b),clear(c),clear(a),on(b,table)]
stack(b,c) would make it true
  clear(b) is untrue in [on(a,b),clear(c),clear(a),on(b,table)]
  unstack(_3980,b) would make it true
    All true: [on(a,b)]
```

```

    Performing unstack(a,b)
    Before: [on(a,b),clear(c),clear(a),on(b,table)]
    After:  [on(a,table),clear(b),clear(c),clear(a),on(b,table)]
    All true: [on(b,table),clear(b),clear(c)]
    Performing stack(b,c)
    Before: [on(a,table),clear(b),clear(c),clear(a),on(b,table)]
    After:  [on(b,c),on(a,table),clear(b),clear(a)]
    on(a,b) is untrue in [on(b,c),on(a,table),clear(b),clear(a)]
    stack(a,b) would make it true
    All true: [on(a,table),clear(a),clear(b)]
    Performing stack(a,b)
    Before: [on(b,c),on(a,table),clear(b),clear(a)]
    After:  [on(a,b),on(b,c),clear(a)]
    All true: [on(a,b),on(b,c)]
    Performing dummy
    Before: [on(a,b),on(b,c),clear(a)]
    After:  [on(a,b),on(b,c),clear(a)]

    P = [dummy,stack(a,b),stack(b,c),unstack(a,b),stack(a,b),unstack(a,c)] ?

```



Solution(1): check the preconditions before you perform the action.

- 1 Choose an unsatisfied goal  
(non-deterministic)  
(make sure you're not already working on this)
- 2 Find an action that brings it about  
(non-deterministic)
- 3 Make a sub-plan to ensure that its preconditions are satisfied. (\*\* Make sure that it doesn't undo anything important \*\*)
- 4 Perform it (\*\* Check that the preconditions really are satisfied \*\*). Go to 1.

The tricky bit is keeping track of which things are important.  
The core of this program is as follows:

```
plan(action(NAME, PRE, add(ADD), delete(DEL)), GOALS, W0-W1,
      PLAN-[NAME | PLAN], _PROTECTED, _TRYING) :-
    all_true(GOALS, W0),
    perform(action(NAME, PRE, add(ADD), delete(DEL)), W0, W1).

plan(ACTION, GOALS, W0-W2, PLAN0-PLAN2, PROTECTED, TRYING) :-
    choose(GOALS, W0, G),
    \+ member(G, TRYING),          % Make sure you're not in a loop
    find_action(G, action(NAME, pre(PRE), add(ADD), delete(DEL))),
    plan(action(NAME, pre(PRE), add(ADD), delete(DEL)),
          PRE, W0-W1, PLAN0-PLAN1, PROTECTED, [G | TRYING], I1),
    all_true(PROTECTED, W1),       % Make sure you don't undo anything important
    plan(ACTION, GOALS, W1-W2, PLAN1-PLAN2, [G | PROTECTED], TRYING, I0).
```

I've extended the set of actions so that you can avoid putting things on the table:

```
action(unstack(A, B),
      pre([isblock(A), isblock(B), clear(A), on(A, B)]),
      add([on(A, table), clear(B)]),
      delete([on(A, B)]))

action(move(A, C),
      pre([isblock(A), isblock(B), isblock(C), on(A, B), clear(A), clear(C)]),
      add([on(A, C), clear(B)]),
      delete([on(A, B), clear(C)]))

action(stack(A, C),
      pre([isblock(A), isblock(C), on(A, table), clear(A), clear(C)]),
      add([on(A, C)]),
      delete([on(A, table), clear(C)]))
```

To get away with this, I've had to explicitly say which things are blocks. Otherwise we start trying to clear the table.

Some preconditions are more important than others: say how much they matter (most important ones = 0, next most = 1, ... ).(Sacerdoti 1974)

```
action(raise,  
    pre([low, 2]),  
    add([high]),  
    delete([low])).
```

```
action(lower,  
    pre([high, 2]),  
    add([low]),  
    delete([high])).
```

```
action(move(X, Y),  
    pre([high, 2], {over(X), 1}),  
    add([over(Y)]),  
    delete([over(X)])).
```

```
action(grasp(X),  
    pre([handempty, 0], {low, 2}, {over(X), 1},  
        {clear(X), 0}, {on(X, Y), 0}),  
    add([holding(X), over(Y), clear(Y)]),  
    delete([handempty, over(X), on(X, Y)])).
```

```
action(ungrasp(X),  
    pre([holding(X), 0], {over(Y), 1}, {low, 2}, {clear(Y), 0}),  
    add([handempty, on(X, Y), over(X)]),  
    delete([holding(X), over(Y), clear(Y)])).
```

```

noah1(PLAN, N) :-
    plan([on(b, c), 0],
        [on(b, a), on(c, table), handempty, high, over(b), clear(a),
         clear(b), clear(c)], PLAN, N).

| ?- noah1(P, 0).
on(b,c) is untrue in [on(b,a),on(c,table),handempty,high,over(b),clear(a),clear(b),clear(c)]
ungrasp(b) would make it true
    holding(b) is untrue in [on(b,a),on(c,table),handempty,high,over(b),clear(a),clear(b),clear(c)]
    grasp(b) would make it true
        All true: [{handempty,0},{low,2},{over(b),1},{clear(b),0},{on(b,a),0}]
        Performing grasp(b)
        Before: [on(b,a),on(c,table),handempty,high,over(b),clear(a),clear(b),clear(c)]
        After:  [holding(b),over(a),clear(a),on(c,table),high,clear(a),clear(b),clear(c)]
        All true: [{holding(b),0},{over(c),1},{low,2},{clear(c),0}]
        Performing ungrasp(b)
        Before: [holding(b),over(a),clear(a),on(c,table),high,clear(a),clear(b),clear(c)]
        After:  [handempty,on(b,c),over(b),over(a),clear(a),on(c,table),high,clear(a),clear(b)]
        All true: [{on(b,c),0}]
        Performing dummy
        Before: [handempty,on(b,c),over(b),over(a),clear(a),on(c,table),high,clear(a),clear(b)]
        After:  [handempty,on(b,c),over(b),over(a),clear(a),on(c,table),high,clear(a),clear(b)]

P = [dummy,ungrasp(b),grasp(b)] ?

```

```

| ?- noah1(P, 1).
on(b,c) is untrue in [on(b,a),on(c,table),handempty,high,over(b),clear(a),clear(b),clear(c)]
ungrasp(b) would make it true
  holding(b) is untrue in [on(b,a),on(c,table),handempty,high,over(b),clear(a),clear(b),clear(c)]
  grasp(b) would make it true
    All true: [{handempty,0},{low,2},{over(b),1},{clear(b),0},{on(b,a),0}]
    Performing grasp(b)
      Before: [on(b,a),on(c,table),handempty,high,over(b),clear(a),clear(b),clear(c)]
      After:  [holding(b),over(a),clear(a),on(c,table),high,clear(a),clear(b),clear(c)]
    over(c) is untrue in [holding(b),over(a),clear(a),on(c,table),high,clear(a),clear(b),clear(c)]
    move(_3538,c) would make it true
      All true: [{high,2},{over(a),1}]
      Performing move(a,c)
        Before: [holding(b),over(a),clear(a),on(c,table),high,clear(a),clear(b),clear(c)]
        After:  [over(c),holding(b),clear(a),on(c,table),high,clear(a),clear(b),clear(c)]
      All true: [{holding(b),0},{over(c),1},{low,2},{clear(c),0}]
      Performing ungrasp(b)
        Before: [over(c),holding(b),clear(a),on(c,table),high,clear(a),clear(b),clear(c)]
        After:  [handempty,on(b,c),over(b),clear(a),on(c,table),high,clear(a),clear(b)]
      All true: [{on(b,c),0}]
      Performing dummy
        Before: [handempty,on(b,c),over(b),clear(a),on(c,table),high,clear(a),clear(b)]
        After:  [handempty,on(b,c),over(b),clear(a),on(c,table),high,clear(a),clear(b)]

P = [dummy,ungrasp(b),move(a,c),grasp(b)] ?

```



```

| ?- noah1(P, 2).
on(b,c) is untrue in [on(b,a),on(c,table),handempty,high,over(b),clear(a),clear(b),clear(c)]
ungrasp(b) would make it true
  holding(b) is untrue in [on(b,a),on(c,table),handempty,high,over(b),clear(a),clear(b),clear(c)]
  grasp(b) would make it true
    low is untrue in [on(b,a),on(c,table),handempty,high,over(b),clear(a),clear(b),clear(c)]
    lower would make it true
      All true: [{high,2}]
      Performing lower
        Before: [on(b,a),on(c,table),handempty,high,over(b),clear(a),clear(b),clear(c)]
        After:  [low,on(b,a),on(c,table),handempty,over(b),clear(a),clear(b),clear(c)]
      All true: [{handempty,0},{low,2},{over(b),1},{clear(b),0},{on(b,a),0}]
      Performing grasp(b)
        Before: [low,on(b,a),on(c,table),handempty,over(b),clear(a),clear(b),clear(c)]
        After:  [holding(b),over(a),clear(a),low,on(c,table),clear(a),clear(b),clear(c)]
    over(c) is untrue in [holding(b),over(a),clear(a),low,on(c,table),clear(a),clear(b),clear(c)]
    move(_4988,c) would make it true
      high is untrue in [holding(b),over(a),clear(a),low,on(c,table),clear(a),clear(b),clear(c)]
      raise would make it true
        All true: [{low,2}]
        Performing raise
          Before: [holding(b),over(a),clear(a),low,on(c,table),clear(a),clear(b),clear(c)]
          After:  [high,holding(b),over(a),clear(a),on(c,table),clear(a),clear(b),clear(c)]
        All true: [{high,2},{over(a),1}]
        Performing move(a,c)
          Before: [high,holding(b),over(a),clear(a),on(c,table),clear(a),clear(b),clear(c)]
          After:  [over(c),high,holding(b),clear(a),on(c,table),clear(a),clear(b),clear(c)]
      low is untrue in [over(c),high,holding(b),clear(a),on(c,table),clear(a),clear(b),clear(c)]

```

```

lower would make it true
  All true: [{high,2}]
  Performing lower
    Before: [over(c),high,holding(b),clear(a),on(c,table),clear(a),clear(b),clear(c)]
    After:  [low,over(c),holding(b),clear(a),on(c,table),clear(a),clear(b),clear(c)]
  All true: [{holding(b),0},{over(c),1},{low,2},{clear(c),0}]
  Performing ungrasp(b)
    Before: [low,over(c),holding(b),clear(a),on(c,table),clear(a),clear(b),clear(c)]
    After:  [handempty,on(b,c),over(b),low,clear(a),on(c,table),clear(a),clear(b)]
  All true: [{on(b,c),0}]
  Performing dummy
    Before: [handempty,on(b,c),over(b),low,clear(a),on(c,table),clear(a),clear(b)]
    After:  [handempty,on(b,c),over(b),low,clear(a),on(c,table),clear(a),clear(b)]

P = [dummy,ungrasp(b),lower,move(a,c),raise,grasp(b),lower] ?

```

I'm doing the whole plan at different levels: what you really want is to make the plan at level 0, and then use level 1 to fill in the gaps, and then level 2 to fill in the gaps in that, ...

```
noah2(PLAN, N) :-
```

```
    plan([on(b, c), 0], [on(a, b), 0],  
          [on(b, a), on(c, table), handempty, high, over(b), on(a, table),  
            clear(b), clear(c)], PLAN, N).
```

```
| ?- noah2(P, 0).
```

```
...
```

```
P = [dummy, ungrasp(a), grasp(a), ungrasp(b), grasp(b)]
```

```
| ?- noah2(P, 1).
```

```
...
```

```
P = [dummy, ungrasp(a), move(table, b), grasp(a), move(b, a), ungrasp(b), move(a, c), grasp(b)] ?
```

```
| ?- noah2(P, 2).
```

```
...
```

```
P = [dummy, ungrasp(a), lower, move(table, b), raise, grasp(a), lower, move(b, a),  
      raise, ungrasp(b), lower, move(a, c), raise, grasp(b), lower]
```

## Register swapping:

```
action(name(store(R1, R2)),  
      pre([contents(R1, X), contents(R2, Y)]),  
      effects([not(contents(R2, Y)), contents(R2, X)]))).
```

```
swap_registers(PLAN) :-  
    GOALS = [contents(r1, 1), contents(r2, 0)],  
    W = [contents(r1, 0), contents(r2, 1), contents(r3, 2)],  
    forwards(GOALS, W, PLAN).
```

- Nothing you can do, even using goal protection. You've got to include a move that isn't actually anything to do with either goal

Think of anything you could possibly do. See what happens, think of anything you could possibly do in that situation. Best done breadth-first (to do things in breadth-first in Prolog, it's a good idea to use the database).

So the basic algorithm is

1. Think of things you could do in the current situation
    - (a) for each of them, add it to your current partial plan and put that in the database
- Pick the **oldest** partial plan in the database.
- (a) Simulate it
  - (b) Think of things you could do in that situation, and ...

When you try it, you find that the actions need much better descriptions:

```
action(raise,  
    pre([low]),  
    add([high]),  
    delete([low])).
```

```
action(lower,  
    pre([high]),  
    add([low]),  
    delete([high])).
```

```
action(grasp(X),  
    pre([block(X), block(Y), {\+ X = Y}, handempty, low, over(X), clear(X), on(X, Y)]),  
    add([holding(X), over(Y), clear(Y)]),  
    delete([handempty, over(X), on(X, Y)]))).
```

```
action(ungrasp(X),  
    pre([block(X), block(Y), {\+ X = Y}, holding(X), over(Y), low, clear(Y)]),  
    add([handempty, on(X, Y), over(X)]),  
    delete([holding(X), over(Y), clear(Y)]))).
```

```
action(move(X, Y),  
    pre([block(X), block(Y), {\+ X = Y}, high, over(X)]),  
    add([over(Y)]),  
    delete([over(X)]))).
```

```
lower
move(b,a)
move(b,c)
grasp(b) lower
lower move(b,a)
move(a,c) move(b,a)
lower move(b,c)
move(c,a) move(b,c)
raise grasp(b) lower
ungrasp(b) grasp(b) lower
lower move(a,c) move(b,a)
lower move(c,a) move(b,c)
move(a,b) raise grasp(b) lower
move(a,c) raise grasp(b) lower
raise ungrasp(b) grasp(b) lower
lower move(a,b) raise grasp(b) lower
...
move(b,c) raise ungrasp(b) grasp(b) lower
lower move(b,c) move(a,b) raise grasp(b) lower
ungrasp(b) lower move(a,c) raise grasp(b) lower
lower move(c,b) move(a,c) raise grasp(b) lower
lower move(b,a) raise ungrasp(b) grasp(b) lower
move(a,c) move(b,a) raise ungrasp(b) grasp(b) lower
lower move(b,c) raise ungrasp(b) grasp(b) lower
move(c,a) move(b,c) raise ungrasp(b) grasp(b) lower
ungrasp(b) lower move(b,c) move(a,b) raise grasp(b) lower
```

Lots of partial plans: dead cheap to make them though.

But I did have to put in some extras: notably, you have to check that you're not in a loop.

Do this by making sure that you don't recreate a previous state. But that means that you've got to be able to directly compare states – to know that `[on(a, b), on(b, table)]` is the same state as `[on(b, table), on(a, b)]`



# Richer Description Languages

---

Allowing rules and implications, e.g.

```
action(unstack(X, Y),  
      pre([on(X, Y), clear(X), stable(Y)]),  
      effects([not(on(X, Y))]))
```

```
stable(Y) if on(Y, table)  
stable(Y) if support1(Y, S1), support2(Y, S2),  
          c_o_g(Y, C), between(C, S1, S2)
```

```
c_o_g(Y, C) if ...
```

Then in order to take  $X$  off  $Y$  you may need to put  $X'$  on it first.

In general, given an unsatisfied goal  $G$  you need to find an action  $action(A, P, E)$  such that  $E \vdash G$ , rather than merely  $member(G, E)$ .

One way to deal with this is by using a theorem prover to see whether the your goals are **provable** rather than just looking to see whether they are explicitly listed.

That will tell me which preconditions are currently true: but how do I find actions that will make something provable? I can't now just match my goals against the list of effects: if I want to have A **above** B, then putting it on C will work if C is on B, but not otherwise. This is a version of the '**ramification problem**': my actions may have consequences which are not immediately visible just by inspection (**Field and Ramsay 2004**).

Turn my action into a '**hypothetical**' proof rule, record the hypotheses that get used during a proof in the label.

I can't have an action whose consequences are that X is above Z.

`on(X, Y) & above(Y, Z) => above(X, Z).`

`hypothetical(do(stack(X, Y))) => on(X, Y).`

Add a rule to the inference engine:

```
prove(hypothetical(H), label(TRAIL, DEF/DEF, HYPS/[H | HYPS]))).
```

Then at the end of a proof you've got a list of actions which would make the thing you're trying to prove proveable. Which is what you need.

## CHECKPOINT

---

- Planning = rational action to achieve your goals
- You have to be able to represent your actions, and to simulate their effects
- You can do it backwards or forwards, depth first or breadth first
- Sequencing matters and is difficult: does non-linear planning sidestep this?
- Planning at different levels of detail can simplify matters
- Actions can have consequences which are not immediately visible. To see if an action will make some goal provable, you have to invoke your theorem prover & use hypothetical proof rules.

## **II NATURAL LANGUAGE PROCESSING**

This course is about tasks that require you manipulate symbols and structures

Seeing the world (Computer vision)

Moving around in the world (Robotics)

Making connections between states of affairs (Reasoning, planning)

Communicating (Natural language processing, speech processing)

Getting better at any of the above (Machine learning, data mining)

Words convey simple ideas. Sequences of words convey complex ideas.



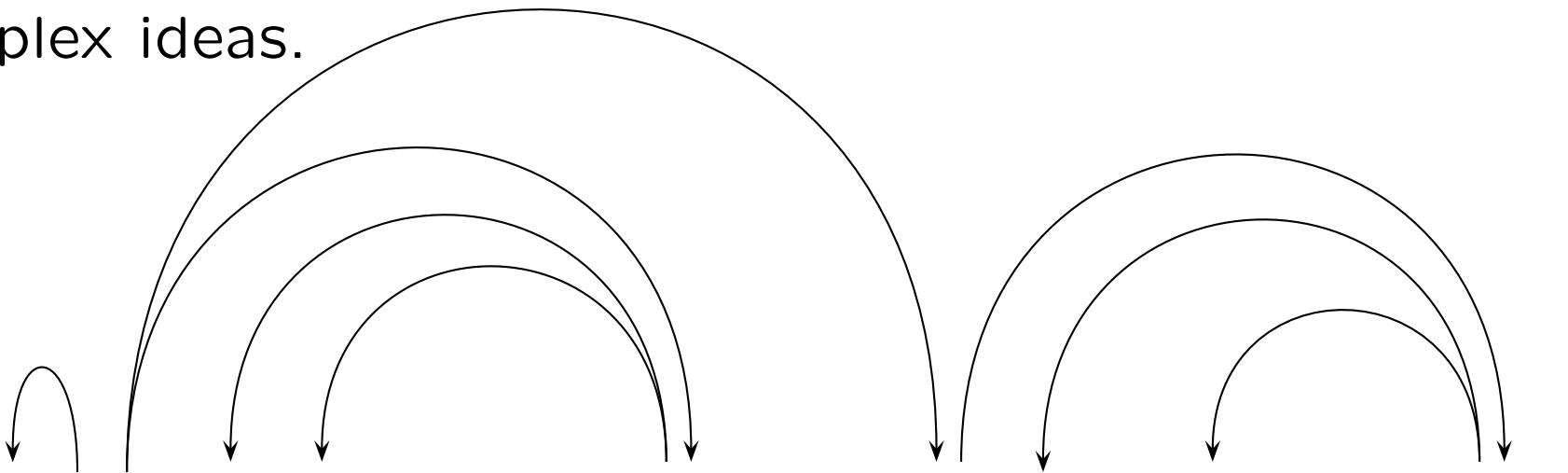
Words convey simple ideas. Sequences of words convey complex ideas.

1(a) I saw the distant mountains with a powerful telescope

(b) I saw the old man with a big nose

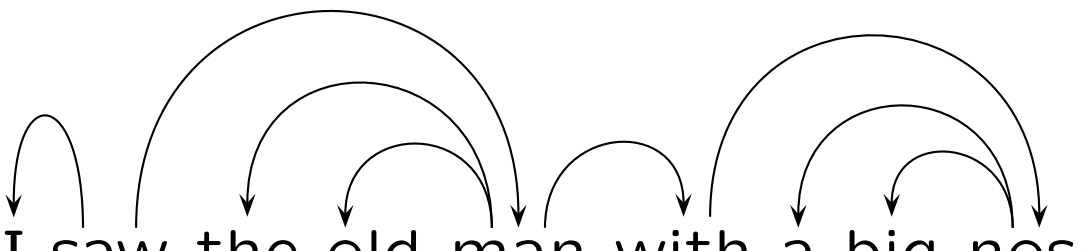
Words convey simple ideas. Arrangements of words convey complex ideas.

1(a) I saw the distant mountains with a powerful telescope



The diagram for sentence (a) shows arcs connecting words to illustrate complex ideas. There are four arcs connecting 'I' to 'mountains', three arcs connecting 'saw' to 'telescope', and two arcs connecting 'distant' to 'powerful'.

(b) I saw the old man with a big nose



The diagram for sentence (b) shows arcs connecting words to illustrate complex ideas. There are four arcs connecting 'I' to 'man', three arcs connecting 'saw' to 'nose', and two arcs connecting 'old' to 'big'.

So we need to know about arrangements of words

So we need to know about arrangements of words

and about how words convey simple ideas

So we need to know about arrangements of words

and about how words convey simple ideas

and about how arrangements of words convey complex ideas

# Arrangements of words: grammar and parsing

---

There are well-known types of words:

- nouns: '*man*', '*cat*', '*dog*', '*house*', '*ball*', ... These typically denote **things**.
- verbs: '*sleep*', '*walk*', '*believe*', ... Typically denote things with temporal properties (actions, states, events)
- adjective & adverbs: '*loud*', '*loudly*', '*old*', ... Add information to nouns and adverbs (would have been better if they'd been called adnouns and adverbs)

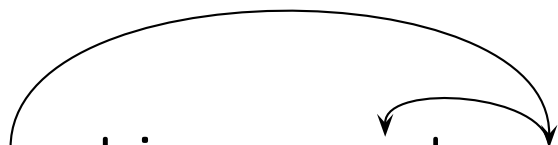
- determiners: '*the*', '*a*', '*every*', ... Nouns describe the **type** of thing you're talking about, determiners tell you what to do with that description. So '*the*' says 'Remember the thing of this kind we were talking about just now', '*a*' says 'I'm introducing something of this kind into the conversation', ... Also known as articles and specifiers

- prepositions: *'in', 'on', 'at', ...* Provide more information about an entity or an event by linking it to some other item: *'man with a big nose', 'slept in the park'*. So their function is a bit like adjectives and adverbs, but they work by linking the thing you're talking about to something else. In some languages (English!) you get postpositions:

(9) I saw him in the park



(10) I saw him a week ago





- pronouns: *'he', 'I', 'this', 'it', ...* Point to something which is very clearly identifiable: it might be something you've been talking about – *'John loves Mary, but **she** doesn't fancy **him**'*, might be something in the external world – *'Let me carry **that** for you'*.
- names: *'John', 'London', 'dawn', ...* Very like pronouns: point to something which is easily identifiable by its name (people often treat them like constants, but they're not: if you go into the national gallery in Scotland and ask about Allan Ramsay they won't think you're talking about me). They're a bit like shorthand for *'the person whose name is 'Allan Ramsay''*, where 'Allan Ramsay' is just a string rather than a symbol.

## Phrase structure rules

---

I've decided that '*John*', '*I*', '*the man*' can all play the same kinds of roles: you can substitute one of these for another and most of the time you'll get a meaningful sentence.

So I need a name for this kind of phrase, and I need to describe what it looks like:

np ==> [name].  
np ==> [det, noun].  
np ==> [pronoun].

name ==> [john].  
det ==> [the].  
man ==> [man].  
pronoun ==> [it].  
pronoun ==> [he].

Rules like this are called ‘**context-free**’ rules. What this means is that when you’re trying to work out whether some phrase matches one of the labels on the right-hand side of a rule you cannot look at any of the surrounding items.

What???

- (11) a. john sleeps.  
      b. the man sleeps.  
      c. it sleeps.  
      d. he sleeps.

The NPs ‘*john*’, ‘*the man*’, ‘*it*’, ‘*he*’ all fit neatly in front of ‘*sleeps*’.

- (12) a. i saw john.  
b. i saw the man.  
c. i saw it.  
d. \* i saw he.

The NPs '*john*', '*the man*', '*it*' fit neatly following '*saw*', but '*he*' doesn't.

It's an NP, but it doesn't fit into this context. It's the wrong **kind** of NP to use here.

But my rule doesn't let me talk about minor variations. NP is the name for a kind of phrase, and that's that. It's an atomic label. No room for anything more subtle.

Clearly (12d) is a problem. We'll come back to it later. For now I'm going to make a basic grammar with atomic symbols, so we've got a jumping-off point.

Anywhere I can put '*the man*', I can put '*the old man*'.

np ==> [name].

np ==> [det, noun].

np ==> [det, adj, noun].

np ==> [pronoun].

det ==> [the].

noun ==> [man].

adj ==> [old].

It'll work, but it's not very appealing. I don't want to write two rules, one mentioning an adjective and one not. The point is that you can have an optional adjective in the np ==> det, noun rule.

Well, actually, you can have several adjectives:

(13) a. I saw the man.

b. I saw the old man.

c. I saw the bad-tempered old man.

d. I saw the rich bad-tempered old man.

. . .

Two ways of doing this:

- Add ways of talking about choice and repetition: ? (0 or 1), \* (0 or more), + (1 or more)

`np ==> det?, adj*, noun.`

- Write your rules recursively:

`np ==> [det, nn].`

`nn ==> [adj, nn].`

`nn ==> [noun].`



Using  $?$ ,  $*$ ,  $+$  looks like it might be more concise, and is more efficient.

But we're going to use the recursive version.

- Some things are **impossible** to describe without recursion, so our algorithms will have to cope with it anyway.
- And other things turn out to be neater with recursion, particularly once we start trying to deal with things like the difference between *'I like her'* and *'me likes she'*.

But it remains true that if you **could** do without recursion you could write more efficient programs.

- (14) a. I slept.  
b. I saw her.  
c. I gave her a book.  
d. I know she read it.

The other half of a sentence seems to form a group as well, made up of the verb and whatever else is needed.

$s \Rightarrow [np, vp]$ .

$vp \Rightarrow [verb]$ .

$vp \Rightarrow [verb, np]$ .

$vp \Rightarrow [verb, np, np]$ .

$vp \Rightarrow [verb, s]$ .

The other half of a sentence seems to form a group as well, made up of the verb and whatever else is needed.

`s ==> [np, vp].`

`vp ==> [verb].`

`vp ==> [verb, np].`

`vp ==> [verb, np, np].`

`vp ==> [verb, s]. %% unavoidable recursion`

- (15) a. (she would marry him)  
b. (Martin [expected (she would marry him)])  
c. (she [believed (Martin [expected (she would marry him)])])  
d. (she [said (she [believed (Martin [expected (she would marry him)])])])  
e. (I [think (she [said (she [believed (Martin [expected (she would marry him)])])])])].

We've seen adding adjectives to a noun. We can do the same with prepositional phrases (PPs).

(16) The man with an umbrella put it down.

np ==> [det, nn].

pp ==> [prep, np].

nn ==> [nn, pp].

pp ==> [prep, np].

np ==> [det, adj\*, n, pp\*].

(17) The man with an umbrella with broken spokes put it down.

We can do the same with VPs.

(18) a. He slept.

b. He slept in the park.

vp ==> [verb] .

vp ==> [verb, np] .

...

vp ==> [vp, pp] .

(19) a. I saw him in the park on Saturday.

b. I saw the man in the park on Saturday.

This is where my problem with '*I saw the man in the park*' comes from. This is an example of '**ambiguity**'. There will be others! <sup>4</sup>

---

<sup>4</sup>The number of possible '**attachment sites**' for a PP in expressions like this follows the Fibonacci series in the exercise.



s ==> [np, vp] .

np ==> [pronoun] .

np ==> [name] .

np ==> [det, nn] .

nn ==> [noun] .

nn ==> [adj, nn] .

nn ==> [nn, pp] .

pp ==> [prep, np] .

vp ==> [verb] .

vp ==> [verb, np] .

vp ==> [verb, np, np] .

vp ==> [verb, s] .

vp ==> [vp, pp] .

det ==> [a] .  
det ==> [the] .

noun ==> [man] .  
noun ==> [park] .  
noun ==> [peach] .

adj ==> [old] .

verb ==> [eats] .  
verb ==> [sleeps] .

prep ==> [in] .

- (20) a. I saw her.  
b. me saw her.  
c. I saw she.  
d. I expect she will do it  
e. I expect her will do it  
f. I expect she to do it  
g. I expect her to do it

What's wrong with the ones that sound wrong?

There are positions where you expect '*he*', '*she*', '*i*' and ones where you expect '*him*', '*her*', '*me*'. The thing that's playing the most active role in the event gets to be the '**subject**', and subjects usually get '**subject case**' (also known as nominative case: I will try to stick to subject case). Most other things get '**object case**' ('**accusative case**').

(situation is more complicated in other languages: roughly speaking, languages seem to have gradually traded case markers for prepositions; and languages with lots of overt case marking allow free-er word orders)

First thought: try something like

s ==> [npsubjcase, vp].	pp ==> [prep, npobjcase].
npsubjcase ==> [pronounsubjcase].	vp ==> [verb].
npsubjcase ==> [name].	vp ==> [verb, npobjcase].
npsubjcase ==> [det, nn].	
npobjcase ==> [pronounobjcase].	pronounsubjcase ==> [i].
npobjcase ==> [name].	pronounobjcase ==> [her].
npobjcase ==> [det, nn].	pronounsubjcase ==> [it].
	pronounobjcase ==> [it].
nn ==> [noun].	pronounsubjcase ==> [she].
nn ==> [adj, nn].	name ==> [martin].
nn ==> [nn, pp].	

It'll work. But I've got more np rules than before: the ones that say

npsubjcase ==> [name].	nn ==> [noun].
npsubjcase ==> [det, nn].	nn ==> [adj, nn].
npobjcase ==> [name].	nn ==> [nn, pp].
npobjcase ==> [det, nn].	pp ==> [prep, npobjcase].

are particularly annoying looking. A bit better might be

npsubjcase ==> [np].	nn ==> [noun].
npobjcase ==> [np].	nn ==> [adj, nn].
np ==> [name].	nn ==> [nn, pp].
np ==> [det, nn].	pp ==> [prep, npobjcase].

- (21) a. i. I sleep.  
ii. I sleeps.  
iii. she sleep.  
iv. she sleeps.
- b. i. I am sleeping.  
ii. he is sleeping.  
iii. we are sleeping.

This time we need something like

s ==> [np1sing, vp1sing].	vp1sing ==> [verb1sing].
s ==> [np3sing, vp3sing].	vp1sing ==> [verb1sing, np].
s ==> [npother, vpother].	vp1sing ==> [verb1sing, np, np].
np1sing ==> [pronoun1sing].	vp3sing ==> [verb3sing].
np3sing ==> [pronoun3sing].	vp3sing ==> [verb3sing, np].
npother ==> [pronounother].	vp3sing ==> [verb3sing, np, np].
np3sing ==> [detsing, nnsing].	vp1sing ==> [verb1sing].
npother ==> [detplural, nnplural].	vp1sing ==> [verb1sing, np].
	vp1sing ==> [verb1sing, np, np].
np ==> [np1sing].	
np ==> [np3sing].	
np ==> [npother].	



This will also work. Just about. But we're going to have to combine it with the other rules for dealing with case.

s ==> [np1singsubjcase, vp1sing].

s ==> [np3singsubjcase, vp3sing].

s ==> [npothersubjcase, vpothet].

np1singsubjcase ==> [pronoun1singsubjcase].

np3singsubjcase ==> [pronoun3singsubjcase].

npothersubjcase ==> [pronounothersubjcase].

np3singsubjcase ==> [detsing, nnsing].

npothersubjcase ==> [detplural, nnplural].

npobjcase ==> [pronounobjcase].

npobjcase ==> [detsing, nnsing].

npobjcase ==> [detplural, nnplural].

pronoun1singsubjcase ==> [i].

pronounobjcase ==> [me].

pronoun3singsubjcase ==> [he].

pronounobjcase ==> [him].

pronoun3singsubjcase ==> [it].

pronounobjcase ==> [it].

This is getting **really** unwieldy. The facts that we are trying to account for are not in doubt. The problem is that using atomic names for labels means that we have to use different names for things that aren't very different.

(22) a. i. He died.

ii. She came.

iii. He murdered her.

iv. \* He murdered.

b. i. She gave him a present.

ii. She gave it to him.

iii. \* She gave.

iv. \* She gave him.

v. \* She gave it.

- c. i. I know she left.
- ii. I expect she will come back.
- iii. I expect her to come back.
- iv. I want to see her.
- v. \* I know her to have left.
- vi. \* I want she will come back.

Several of the starred ones aren't actually all that bad, but not every verb happily takes part in every construction.

```
vp ==> [iverb]. %% intransitive: sleep, die, ...  
vp ==> [tverb, np]. %% transitive: eat, hit, ...  
vp ==> [dverb, np, np]. %% ditransitive: give, make, ...  
vp ==> [scverb, s]. %% scomp: know, expect
```

But they're **atomic** names. I can't see that they're all verbs, because I can't see inside the names.

(and anyway, they're not fine-grained enough.

- (23) a. i. I gave her a present.  
ii. I made her a cup of tea.  
iii. I gave it to her.  
iv. I made it for her.  
v. \* I gave it for her.  
vi. \* I made it to her.

- (24) a. i. I saw him eating it.  
ii. \* I saw him to eat it.  
iii. I saw he had eaten it.  
i. \* I expect him eating it.  
ii. I expect him to eat it.  
iii. I expect he had eaten it.  
i. \* I want him eating it.  
ii. I want him to eat it.  
iii. \* I want he had eaten it.)



'saw' is a verb; it's past tense; it will agree with any subject; it's transitive.

Write as a bundle of 'features':

```
{cat=verb,  
  agree=_,  
  tense=past,  
  subcat=transitive}
```

Then in my rules I mention those features that I'm interested in.

S ==> [NP, VP] :-

cat@S -- s,

cat@NP -- np,

cat@VP -- vp,

case@NP -- subj,

agree@NP -- agree@VP.

NP ==> [DET, NN] :-

cat@NP -- np,

cat@DET -- det,

cat@NN -- nn,

agree@NP -- agree@DET,

agree@NP -- agree@NN.

NP ==> [PRO] :-

cat@NP -- np,

cat@PRO -- pronoun,

[agree, case]@NP -- [agree, case]@PRO.

```

word('I', PRON) :-
    cat@PRON -- pronoun,
    case@PRON -- subj,
    agree@PRON -- firstsing.
word('it', PRON) :-
    cat@PRON -- pronoun,
    agree@PRON -- thirdsing.
word('a', DET) :-
    cat@DET -- det,
    agree@DET -- thirdsing.
word('the', DET) :-
    cat@DET -- det.
word('cat', NOUN) :-
    cat@NOUN -- noun,
    agree@NOUN -- thirdsing.
word('cats', NOUN) :-
    cat@NOUN -- noun,
    agree@NOUN -- thirdplural.
word('sheep', NOUN) :-
    cat@NOUN -- noun.

```

Much smaller set of rather verbose rules.

At any point I mention just those aspects of a word or phrase that I'm interested in at that point.

Labels are no longer atomic. That might make my algorithms more complex.

The category is just another feature. I don't say `np{agree=firstsing, case=subject}`, I say `{agree=firstsing, case=subject, cat=np}`.

Information is carried around by '**unification**' (Prolog was actually invented for parsing).

I have to have millions of rules for different type of VPs, and then I have to say which of these rules a particular verb takes part in.

```
%% sleeps
VP ==> [V] :-
    [agree, vform]@VP -- [agree, vform]@V,
    cat@VP -- vp,
    cat@V -- verb,
    subcat@V -- intrans.
```

```
%% eats it
VP ==> [V, NP] :-
    [agree, vform]@VP -- [agree, vform]@V,
    cat@VP -- vp,
    cat@V -- verb,
    cat@NP -- np,
    subcat@V -- trans.
```

```
%% gave her a present, made her a cake
VP ==> [V, NP1, NP2] :-
    [agree, vform]@VP -- [agree, vform]@V,
    cat@VP -- vp,
    cat@V -- verb,
    cat@NP1 -- np,
    cat@NP2 -- np,
    subcat@V -- ditrans(_).
```

```
%% gave it to her, made it for her, * gave it fo her, * made it to he  
VP ==> [V, NP, PP] :-  
    [agree, vform]@VP -- [agree, vform]@V,  
    cat@VP -- vp,  
    cat@V -- verb,  
    cat@NP -- np,  
    cat@PP -- pp,  
    prep@PP -- PREP,  
    subcat@V -- ditrans(PREP).
```



```

%% I like to sleep, I like sleeping, I want to sleep, * I want sleep:
VP0 ==> [V, VP1] :-
    [agree, vform]@VP -- [agree, vform]@V,
    cat@VP0 -- vp,
    cat@V -- verb,
    cat@VP1 -- vp,
    vform@VP1 -- VFORM1,
    subcat@V -- vpcomp(VFORM1).

```

```
%% I expect she will marry him, I expect her to marry him,  
%% I know she will marry him, * I know her to marry him  
VP ==> [V, S] :-  
    [agree, vform]@VP -- [agree, vform]@V,  
    cat@VP0 -- vp,  
    cat@V -- verb,  
    cat@S -- s,  
    vform@S -- VFORM1,  
    subcat@V -- scomp(VFORM1).
```

And then you have to say which rules apply to each verb (most verbs will fit more than one rule):

```
word('sleep', X) :-  
    cat@X -- verb,  
    subcat@X -- intrans.
```

```
%% I've already eaten  
word('eat', X) :-  
    cat@X -- verb,  
    subcat@X -- intrans.
```

```
%% the unicorn ate some plum cake  
word('eat', X) :-  
    cat@X -- verb,  
    subcat@X -- trans.
```

```
%% my mum made a cake
word('make', X) :-
    cat@X -- verb,
    subcat@X -- trans.
```

```
%% my mum made me a cake
word('make', X) :-
    cat@X -- verb,
    subcat@X -- ditrans(for).
```

```
%% I was expecting his arrival
word('expect', X) :-
    cat@X -- verb,
    subcat@X -- trans.
```

```
%% I expected him to arrive
word('expect', X) :-
    cat@X -- verb,
    subcat@X -- scomp(toForm).
```

```
%% I expected he would be able to see me
word('expect', X) :-
    cat@X -- verb,
    subcat@X -- scomp(tensed).
```

You can't avoid saying what arguments a verb takes. Different verbs take slightly different sets of arguments, and you just have to say what they do.

But having loads of different rules, each with a fairly meaningless name, and then saying which rule each verb takes part in is very annoying.

Especially if you find that you've got some verb that takes a completely idiosyncratic set of arguments, so you have to write a rule for that verb and then say that it takes part in that rule.

What you want to say is: each verb requires some arguments, and when it's got them then you've got a VP.

```
VP ==> [V | ARGS] :-  
    cat@VP -- vp,  
    cat@V -- verb,  
    args@VERB -- ARGS.
```



Then you might define a type of verb, say a transitive verb, by saying

```
tverb(V) :-  
    cat@V -- verb,  
    args@X -- [OBJ],  
    cat@OBJ -- np.
```

And then provide instances of it:

```
word(eat, X) :-  
    tverb(X).
```

**It's the same information as before: just that everything's specified within the word, with just a very skeletal rule**

Slightly funny technically, because the value of a feature is now a list of signs, rather than just a label like `vp` or `subjcase`, but nothing to stop us doing that if we want.

I can push this idea a bit further: a verb hasn't actually got everything it needs until it gets its subject. So I'd really like a rule like

```
S ==> [V | ARGS] :-  
    cat@S -- s,  
    cat@V -- verb,  
    args@V -- ARGS.
```

```
tverb(V) :-  
    cat@V -- verb,  
    args@X -- [SUBJ, OBJ],  
    cat@SUBJ -- np,  
    cat@OBJ -- np.
```

That's not quite going to work, because I've got to note that the subject comes before the verb.

```
tverb(V) :-  
    cat@V -- verb,  
    args@X -- [OBJ, SUBJ],  
    cat@SUBJ -- np,  
    dir@SUBJ -- before,  
    cat@OBJ -- np,  
    dir@OBJ -- after.
```

I need a slightly different set of rules now:

```
X0 ==> [Y, X1] :-  
    cat@X -- cat@X1,  
    args@X1 -- [Y | ARGS@X0],  
    dir@Y -- before.
```

```
X0 ==> [X1, Y] :-  
    cat@X -- cat@X1,  
    args@X1 -- [Y | ARGS@X0],  
    dir@Y -- after.
```

## Worked examples

Very neat

Very flexible

Linguistic generalisations:

English:  $v[\overrightarrow{OBJ}, \overleftarrow{SUBJ}]$

Arabic:  $v[\overrightarrow{SUBJ}, \overleftarrow{OBJ}]$

Persian:  $v[\overleftarrow{OBJ}, \overrightarrow{SUBJ}]$

If it worked it would be all you needed

The grammars we've seen so far have specified two things at once: what pieces a phrase is made out of, and where they appear.

$s \Rightarrow [np, vp]$ .

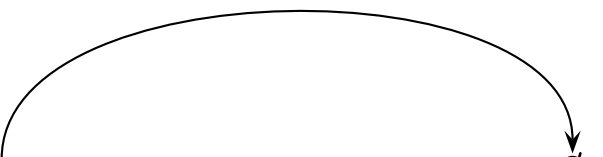
A sentence is made out of an NP followed by a VP.



Sometimes the pieces don't come in the order in which you expect them:

(25) a. I enjoyed the main course, but I thought the pudding was disgusting

b. I enjoyed the main course, but the pudding I thought  $\emptyset$  was disgusting

A curved arrow originates from the empty set symbol ( $\emptyset$ ) at the end of the sentence in example (25) b and points back to the word 'pudding'.

Different languages allow different amounts of this.

In a lot of languages, you can shift phrases to the beginning or end of the utterance for various reasons:

- Emphasis (as above)

Some movements are quite local: you can permute the elements of a German sentence around the verb, you can put the subject of an Arabic sentence in front of the verb or behind it.

Some items can be moved arbitrarily far: *'Who did you say you thought she wanted me to meet □?'*. '**Long-distance dependencies**' of this kind are particularly difficult to handle.

- Ambiguity reduction

(26) a. I believe that she loves me with all my heart.

- Ambiguity reduction

(26) a. I believe that she loves me with all my heart.

b. I believe with all my heart that she loves me  $\emptyset$ .



- Ambiguity reduction

(26) a. I believe that she loves me with all my heart.

b. I believe with all my heart that she loves me  $\emptyset$ .



(27) a. I saw the man in the park.

- Ambiguity reduction

(26) a. I believe that she loves me with all my heart.

b. I believe with all my heart that she loves me  $\emptyset$ .



(27) a. I saw the man in the park.

b. In the park I saw the man  $\emptyset$ .



In many languages, you can permute the order of the arguments and modifiers: German is quite nice here—in main sentences you can have any constituent you like in initial position so long as the verb is the second item.

(28) a. Ich sah den Mann im Park.

b. den Mann sah Ich im Park.

c. im Park sah Ich den Mann.

(29) (same phenomenon in English)

a. An old man sat on the bus.

b. On the bus sat an old man.

In all the languages I know anything about, you have to shift  
'**WH-marked**' items to the front:

- (30) a. Who did you see  $\emptyset$ ?  
b. Where did you say you thought you saw him  $\emptyset$ ?  
c. I saw the woman who you said you had given it to  $\emptyset$ .  
d. I saw the woman who you said  $\emptyset$  had given it to you.
- 
- The diagram illustrates WH movement in four sentences. In sentence (a), an arrow points from the null operator  $\emptyset$  to the front position. In sentence (b), an arrow points from the null operator  $\emptyset$  to the front position. In sentence (c), an arrow points from the null operator  $\emptyset$  to the front position. In sentence (d), an arrow points from the null operator  $\emptyset$  to the front position.



I will return to this later if we get time: for now, just remember that you can't do the local permutation ones easily with rewrite rules of the kind we've seen, and you can't do the long-distance ones at all with them.

- Why do I care about grammar?
- What makes a grammar '**context-free**' ?
- What can't I do with atomic labels? Agreement, case-marking, subcat frames, verb forms
- How does using features solve these problems?
- The structure of labels in categorial grammar
- The problems that '**out of order items**' cause for phrase structure grammars ('**marked orders**', '**non-canonical orders**'). Not the solutions.

Why do I want a grammar? Because I want to know the relationships between words and phrases.

So I also need a program which can find the relationships between words and phrases.

Two simple algorithms to get started. Then the one that works.

Test them initially on our basic grammar. They will work for better grammars, but we we'll use the simple one to start with for clarity.

$s \Rightarrow [np, vp].$

$np \Rightarrow [pronoun].$

$np \Rightarrow [name].$

$np \Rightarrow [det, nn].$

$nn \Rightarrow [noun].$

$nn \Rightarrow [adj, nn].$

$nn \Rightarrow [nn, pp].$

$pp \Rightarrow [prep, np].$

$vp \Rightarrow [verb].$

$vp \Rightarrow [verb, np].$

$vp \Rightarrow [verb, np, np].$

$vp \Rightarrow [verb, s].$

$vp \Rightarrow [vp, pp].$

I think I've got an S. To verify that, I'll see if I've got the things that you need in order to make an S.

To make an S, you need an NP and VP. So I'll see if I've got an NP, and if I find one of those then I'll see if the rest of what I've got makes a VP.

To make an NP, you need ...

Finding a constituent consumes part of the text, so after I've found something I want to know what the remaining text is.

(this is **extremely** like the Horn part of our theorem prover. You could catch the loops the same way that we do there)

%% at the end I want there to be nothing unconsumed

```
td(X, TXT0) :-  
    td(X, TXT0, []).
```

%% Is the current word actually an X?

```
td(X, [H | T], T) :-  
    word(H, X).
```

%% Do we have a rule for making Xs?

```
td(X, T0, T1) :-  
    X ==> RHS,  
    tdAll(RHS, T0, T1).
```

```
tdAll([], T, T).
```

```
tdAll([H | T], TXT0, TXTN) :-  
    td(H, TXT0, TXT1),  
    tdAll(T, TXT1, TXTN).
```

That told me whether my text was a sentence: but it didn't show me the structure.

%% at the end I want there to be nothing unconsumed

```
tdp(X, TXT0, TREE) :-  
    tdp(X, TXT0, [], TREE).
```

%% Is the current word actually an X?

```
tdp(X, [H | T], T, {H, X}) :-  
    word(H, X).
```

%% Do we have a rule for making Xs?

```
tdp(X, T0, T1, {X, RHS1}) :-  
    X ==> RHS0,  
    tdpAll(RHS0, T0, T1, RHS1).
```

```
tdpAll([], T, T, []).
```

```
tdpAll([H | T0], TXT0, TXTN, [R | T1]) :-  
    tdp(H, TXT0, TXT1, R),  
    tdpAll(T0, TXT1, TXTN, T1).
```



## Problems with top-down parsing:

- don't look at the text till we get to leaf node. So we go through the same sequence of moves every time we are looking for an NP, even though they are completely the wrong thing to do
- get stuck in left recursion

Do we have the right-hand side of any rule?

If so, replace it by the left-hand side.

This one is easiest to write using the Prolog database. Very like the 'split' stage of the theorem prover.

```
bu(TEXT) :-  
    retractall(edge(_, _, _, _)),  
    addAllWords(TEXT, 0),  
    rewrite.
```

```
addAllWords([], _N).  
addAllWords([H | T], I) :-  
    J is I+1,  
    word(H, X),  
    assert(edge(I, J, X, H)),  
    addAllWords(T, J).
```

```
rewrite :-  
    X ==> RHS0,  
    findParts(RHS0, RHS1, I, J),  
    \+ edge(I, J, X, RHS1),  
    assert(edge(I, J, X, RHS1)),  
    rewrite.
```

## Problems with bottom-up parsing

- easily misled by lexical ambiguity

## Choice points:

- Which rule shall I use?

Top-down: I've got lots of rules that will let me make an NP. Which one should I use?

Bottom-up: I've got lots of rules whose right-hand sides are matched. Which one should I use?

- Which version of a word shall I use?

Top-down: not really a problem

Bottom-up: different versions of the same word will satisfy different rules.

In my bottom-up parser, I stuck things that I'd found in the database.

What about putting things I'd tried to find in there. Then if ever I thought of trying to find the same thing again, I'd know that if it existed I would already have found it.

Fundamental rule of chart parsing: if you've got an  $X$  that needs  $[H \mid T]$  to make a complete  $X$ , you can combine them to make an  $X$  that needs  $T$  to complete itself

```
chart(TEXT) :-  
    retractall(edge(_, _, _, _, _)),  
    addAllWords(TEXT, 0).  
  
addAllWords([], _N).  
addAllWords([H | T], I) :-  
    J is I+1,  
    word(H, X),  
    (newEdge(edge(I, J, X, H, [])); true),  
    addAllWords(T, J).  
  
newEdge(EDGE) :-  
    \+ EDGE,  
    assert(EDGE),  
    combine(EDGE).
```

```
combine(edge(I, J, L0, DTRS0, [H | T])) :-  
    edge(J, K, H, DTRS1, []),  
    append(DTRS0, [DTRS1], DTRS),  
    newEdge(edge(I, K, L0, DTRS, T)).
```

```
combine(edge(J, K, H, DTRS1, [])) :-  
    edge(I, J, L0, DTRS0, [H | T]),  
    append(DTRS0, [DTRS1], DTRS),  
    newEdge(edge(I, K, L0, DTRS, T)).
```

```
combine(edge(J, K, H, DTRS1, [])) :-  
    LHS ==> [H | RHS],  
    newEdge(edge(J, K, LHS, [DTRS1], RHS)).
```



Traced example for

(31) I know the man saw her

Lots of slides: you should get the picture after about 10.

Adding lexical edge {0, 1, pronoun, i, [], 0}

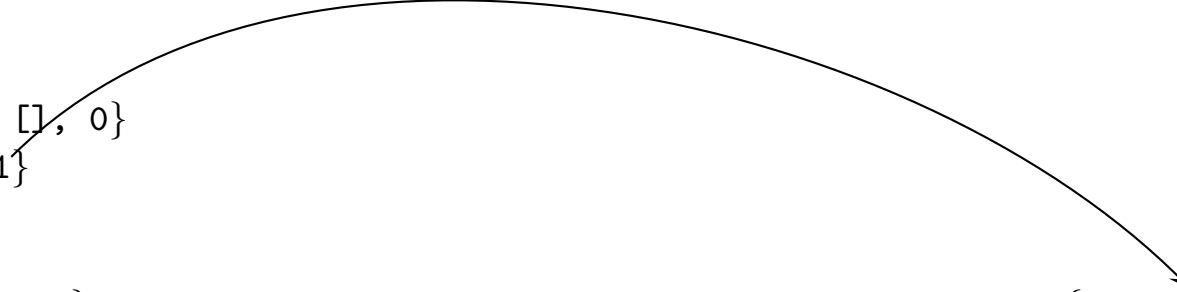
{0, 1, pronoun, i, [], 0}

Adding {0,1,np,i,[],1} because start of np==>[pronoun] has been matched by {0,1,pronoun,i,[],0}

$\{0, 1, \text{pronoun}, i, [], 0\}$

$\{0, 1, \text{np}, i, [], 1\}$

Adding  $\{0, 1, s, i, [\text{vp}], 2\}$  because start of  $s \Rightarrow [\text{np}, \text{vp}]$  has been matched by  $\{0, 1, \text{np}, i, [], 1\}$



{0, 1, pronoun, i, [], 0}

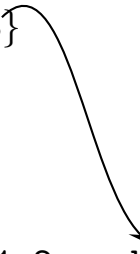
{0, 1, np, i, [], 1}

{0, 1, s, i, [vp], 2}

Adding lexical edge {1, 2, verb, know, [], 3}

{0, 1, pronoun, i, [], 0}  
{0, 1, np, i, [], 1}  
{0, 1, s, i, [vp], 2}

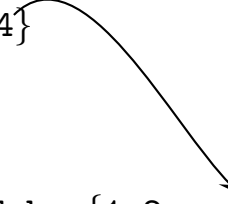
{1, 2, verb, know, [], 3}



Adding {1,2,vp,know,[],4} because start of vp==>[verb] has been matched by {1,2,verb,know,[],3}

{0, 1, pronoun, i, [], 0}  
{0, 1, np, i, [], 1}  
{0, 1, s, i, [vp], 2}

{1, 2, verb, know, [], 3}  
{1, 2, vp, know, [], 4}




Adding {1,2,vp,know,[pp],5} because start of vp==>[vp,pp] has been matched by {1,2,vp,know,[],4}

{0, 1, pronoun, i, [], 0}  
{0, 1, np, i, [], 1}  
{0, 1, s, i, [vp], 2}  
{1, 2, verb, know, [], 3}

{1, 2, vp, know, [], 4}  
{1, 2, vp, know, [pp], 5}

Adding {0,2,s,[i,known],[],6}: {0,1,s,i,[vp],2} wanted {1,2,vp,known,[],4}





{0, 1, pronoun, i, [], 0}  
{0, 1, np, i, [], 1}  
{0, 1, s, i, [vp], 2}  
{1, 2, verb, know, [], 3}

{1, 2, vp, know, [], 4}  
{1, 2, vp, know, [pp], 5}  
{0, 2, s, [i, know], [], 6}

Adding {1,2,vp,knownp],7} because start of vp==>[verb,np] has been matched by {1,2,verb,knownp],3}

{0, 1, pronoun, i, [], 0}  
{0, 1, np, i, [], 1}  
{0, 1, s, i, [vp], 2}  
{1, 2, verb, know, [], 3}  
{1, 2, vp, know, [], 4}

{1, 2, vp, know, [pp], 5}  
{0, 2, s, [i, know], [], 6}  
{1, 2, vp, know, [np], 7}

Adding {1,2,vp,know,[np,np],8} because start of vp==>[verb,np,np] has been matched by {1,2,verb,know,[],3}

{0, 1, pronoun, i, [], 0}  
{0, 1, np, i, [], 1}  
{0, 1, s, i, [vp], 2}  
{1, 2, verb, know, [], 3}  
{1, 2, vp, know, [], 4}

{1, 2, vp, know, [pp], 5}  
{0, 2, s, [i, know], [], 6}  
{1, 2, vp, know, [np], 7}  
{1, 2, vp, know, [np, np], 8}

Adding {1,2,vp,know,[s],9} because start of vp==>[verb,s] has been matched by {1,2,verb,know,[],3}

{0, 1, pronoun, i, [], 0}  
{0, 1, np, i, [], 1}  
{0, 1, s, i, [vp], 2}  
{1, 2, verb, know, [], 3}  
{1, 2, vp, know, [], 4}  
{1, 2, vp, know, [pp], 5}

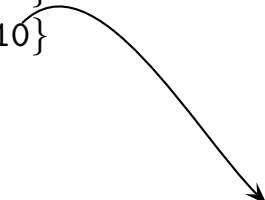
{0, 2, s, [i, know], [], 6}  
{1, 2, vp, know, [np], 7}  
{1, 2, vp, know, [np, np], 8}  
{1, 2, vp, know, [s], 9}

Adding lexical edge {2, 3, det, the, [], 10}

{0, 1, pronoun, i, [], 0}  
{0, 1, np, i, [], 1}  
{0, 1, s, i, [vp], 2}  
{1, 2, verb, know, [], 3}  
{1, 2, vp, know, [], 4}  
{1, 2, vp, know, [pp], 5}

{0, 2, s, [i, know], [], 6}  
{1, 2, vp, know, [np], 7}  
{1, 2, vp, know, [np, np], 8}  
{1, 2, vp, know, [s], 9}  
{2, 3, det, the, [], 10}

Adding {2,3,np,the,[nn],11} because start of np==>[det,nn] has been matched by {2,3,det,the,[],10}



{0, 1, pronoun, i, [], 0}  
{0, 1, np, i, [], 1}  
{0, 1, s, i, [vp], 2}  
{1, 2, verb, know, [], 3}  
{1, 2, vp, know, [], 4}  
{1, 2, vp, know, [pp], 5}  
{0, 2, s, [i, know], [], 6}

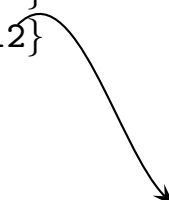
{1, 2, vp, know, [np], 7}  
{1, 2, vp, know, [np, np], 8}  
{1, 2, vp, know, [s], 9}  
{2, 3, det, the, [], 10}  
{2, 3, np, the, [nn], 11}

Adding lexical edge {3, 4, noun, man, [], 12}

{0, 1, pronoun, i, [], 0}  
{0, 1, np, i, [], 1}  
{0, 1, s, i, [vp], 2}  
{1, 2, verb, know, [], 3}  
{1, 2, vp, know, [], 4}  
{1, 2, vp, know, [pp], 5}  
{0, 2, s, [i, know], [], 6}

{1, 2, vp, know, [np], 7}  
{1, 2, vp, know, [np, np], 8}  
{1, 2, vp, know, [s], 9}  
{2, 3, det, the, [], 10}  
{2, 3, np, the, [nn], 11}  
{3, 4, noun, man, [], 12}

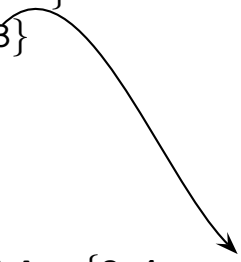
Adding {3,4,nn,man,[],13} because start of nn==>[noun] has been matched by {3,4,noun,man,[],12}



{0, 1, pronoun, i, [], 0}  
{0, 1, np, i, [], 1}  
{0, 1, s, i, [vp], 2}  
{1, 2, verb, know, [], 3}  
{1, 2, vp, know, [], 4}  
{1, 2, vp, know, [pp], 5}  
{0, 2, s, [i, know], [], 6}  
{1, 2, vp, know, [np], 7}

{1, 2, vp, know, [np, np], 8}  
{1, 2, vp, know, [s], 9}  
{2, 3, det, the, [], 10}  
{2, 3, np, the, [nn], 11}  
{3, 4, noun, man, [], 12}  
{3, 4, nn, man, [], 13}

Adding {3,4,nn,man,[pp],14} because start of nn==>[nn,pp] has been matched by {3,4,nn,man,[],13}





{0, 1, pronoun, i, [], 0}  
 {0, 1, np, i, [], 1}  
 {0, 1, s, i, [vp], 2}  
 {1, 2, verb, know, [], 3}  
 {1, 2, vp, know, [], 4}  
 {1, 2, vp, know, [pp], 5}  
 {0, 2, s, [i, know], [], 6}  
 {1, 2, vp, know, [np], 7}

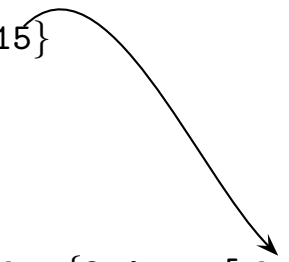
{1, 2, vp, know, [np, np], 8}  
 {1, 2, vp, know, [s], 9}  
 {2, 3, det, the, [], 10}  
 {2, 3, np, the, [nn], 11}  
 {3, 4, noun, man, [], 12}  
 {3, 4, nn, man, [], 13}  
 {3, 4, nn, man, [pp], 14}

Adding {2,4,np,[the,man],[],15}: {2,3,np,the,[nn],11} wanted {3,4,nn,man,[],13}

{0, 1, pronoun, i, [], 0}  
{0, 1, np, i, [], 1}  
{0, 1, s, i, [vp], 2}  
{1, 2, verb, know, [], 3}  
{1, 2, vp, know, [], 4}  
{1, 2, vp, know, [pp], 5}  
{0, 2, s, [i, know], [], 6}  
{1, 2, vp, know, [np], 7}  
{1, 2, vp, know, [np, np], 8}

{1, 2, vp, know, [s], 9}  
{2, 3, det, the, [], 10}  
{2, 3, np, the, [nn], 11}  
{3, 4, noun, man, [], 12}  
{3, 4, nn, man, [], 13}  
{3, 4, nn, man, [pp], 14}  
{2, 4, np, [the, man], [], 15}

Adding {2,4,s,[the,man],[vp],16} because start of s==>[np,vp] has been matched by {2,4,np,[the,man],[],15}



{0, 1, pronoun, i, [], 0}  
{0, 1, np, i, [], 1}  
{0, 1, s, i, [vp], 2}  
{1, 2, verb, know, [], 3}  
{1, 2, vp, know, [], 4}  
{1, 2, vp, know, [pp], 5}  
{0, 2, s, [i, know], [], 6}  
{1, 2, vp, know, [np], 7}  
{1, 2, vp, know, [np, np], 8}

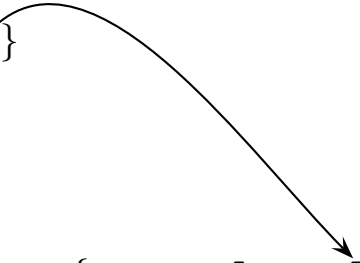
{1, 2, vp, know, [s], 9}  
{2, 3, det, the, [], 10}  
{2, 3, np, the, [nn], 11}  
{3, 4, noun, man, [], 12}  
{3, 4, nn, man, [], 13}  
{3, 4, nn, man, [pp], 14}  
{2, 4, np, [the, man], [], 15}  
{2, 4, s, [the, man], [vp], 16}

Adding {1,4,vp,[know,[the,man]],[],17}: {1,2,vp,know,[np],7} wanted {2,4,np,[the,man],[],15}

{0, 1, pronoun, i, [], 0}  
{0, 1, np, i, [], 1}  
{0, 1, s, i, [vp], 2}  
{1, 2, verb, know, [], 3}  
{1, 2, vp, know, [], 4}  
{1, 2, vp, know, [pp], 5}  
{0, 2, s, [i, know], [], 6}  
{1, 2, vp, know, [np], 7}  
{1, 2, vp, know, [np, np], 8}  
{1, 2, vp, know, [s], 9}

{2, 3, det, the, [], 10}  
{2, 3, np, the, [nn], 11}  
{3, 4, noun, man, [], 12}  
{3, 4, nn, man, [], 13}  
{3, 4, nn, man, [pp], 14}  
{2, 4, np, [the, man], [], 15}  
{2, 4, s, [the, man], [vp], 16}  
{1, 4, vp, [know, [the, man]], [], 17}

Adding {1,4,vp,[know,[the,man]], [pp], 18} because start of vp==>[vp,pp] has been matched by {1,4,vp,[know,[the,man]], []



{0, 1, pronoun, i, [], 0}  
 {0, 1, np, i, [], 1}  
 {0, 1, s, i, [vp], 2}  
 {1, 2, verb, know, [], 3}  
 {1, 2, vp, know, [], 4}  
 {1, 2, vp, know, [pp], 5}  
 {0, 2, s, [i, know], [], 6}  
 {1, 2, vp, know, [np], 7}  
 {1, 2, vp, know, [np, np], 8}  
 {1, 2, vp, know, [s], 9}

{2, 3, det, the, [], 10}  
 {2, 3, np, the, [nn], 11}  
 {3, 4, noun, man, [], 12}  
 {3, 4, nn, man, [], 13}  
 {3, 4, nn, man, [pp], 14}  
 {2, 4, np, [the, man], [], 15}  
 {2, 4, s, [the, man], [vp], 16}  
 {1, 4, vp, [know, [the, man]], [], 17}  
 {1, 4, vp, [know, [the, man]], [pp], 18}

Adding {0,4,s,[i,[know,[the,man]]],[],19}: {0,1,s,i,[vp],2} wanted {1,4,vp,[know,[the,man]],[],17}

{0, 1, pronoun, i, [], 0}  
 {0, 1, np, i, [], 1}  
 {0, 1, s, i, [vp], 2}  
 {1, 2, verb, know, [], 3}  
 {1, 2, vp, know, [], 4}  
 {1, 2, vp, know, [pp], 5}  
 {0, 2, s, [i, know], [], 6}  
 {1, 2, vp, know, [np], 7}  
 {1, 2, vp, know, [np, np], 8}  
 {1, 2, vp, know, [s], 9}  
 {2, 3, det, the, [], 10}

{2, 3, np, the, [nn], 11}  
 {3, 4, noun, man, [], 12}  
 {3, 4, nn, man, [], 13}  
 {3, 4, nn, man, [pp], 14}  
 {2, 4, np, [the, man], [], 15}  
 {2, 4, s, [the, man], [vp], 16}  
 {1, 4, vp, [know, [the, man]], [], 17}  
 {1, 4, vp, [know, [the, man]], [pp], 18}  
 {0, 4, s, [i, [know, [the, man]]], [], 19}

Adding {1, 4, vp, [know, [the, man]], [np], 20}: {1, 2, vp, know, [np, np], 8} wanted {2, 4, np, [the, man], [], 15}

```

{0, 1, pronoun, i, [], 0}
{0, 1, np, i, [], 1}
{0, 1, s, i, [vp], 2}
{1, 2, verb, know, [], 3}
{1, 2, vp, know, [], 4}
{1, 2, vp, know, [pp], 5}
{0, 2, s, [i, know], [], 6}
{1, 2, vp, know, [np], 7}
{1, 2, vp, know, [np, np], 8}
{1, 2, vp, know, [s], 9}
{2, 3, det, the, [], 10}

```

```

{2, 3, np, the, [nn], 11}
{3, 4, noun, man, [], 12}
{3, 4, nn, man, [], 13}
{3, 4, nn, man, [pp], 14}
{2, 4, np, [the, man], [], 15}
{2, 4, s, [the, man], [vp], 16}
{1, 4, vp, [know, [the, man]], [], 17}
{1, 4, vp, [know, [the, man]], [pp], 18}
{0, 4, s, [i, [know, [the, man]]], [], 19}
{1, 4, vp, [know, [the, man]], [np], 20}

```

Adding lexical edge {4, 5, verb, saw, [], 21}

```

{0, 1, pronoun, i, [], 0}
{0, 1, np, i, [], 1}
{0, 1, s, i, [vp], 2}
{1, 2, verb, know, [], 3}
{1, 2, vp, know, [], 4}
{1, 2, vp, know, [pp], 5}
{0, 2, s, [i, know], [], 6}
{1, 2, vp, know, [np], 7}
{1, 2, vp, know, [np, np], 8}
{1, 2, vp, know, [s], 9}
{2, 3, det, the, [], 10}
{2, 3, np, the, [nn], 11}

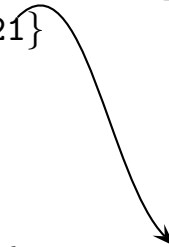
```

```

{3, 4, noun, man, [], 12}
{3, 4, nn, man, [], 13}
{3, 4, nn, man, [pp], 14}
{2, 4, np, [the, man], [], 15}
{2, 4, s, [the, man], [vp], 16}
{1, 4, vp, [know, [the, man]], [], 17}
{1, 4, vp, [know, [the, man]], [pp], 18}
{0, 4, s, [i, [know, [the, man]]], [], 19}
{1, 4, vp, [know, [the, man]], [np], 20}
{4, 5, verb, saw, [], 21}

```

Adding {4,5,vp,saw,[],22} because start of vp==>[verb] has been matched by {4,5,verb,saw,[],21}

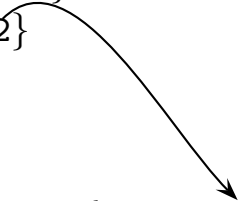




{0, 1, pronoun, i, [], 0}  
 {0, 1, np, i, [], 1}  
 {0, 1, s, i, [vp], 2}  
 {1, 2, verb, know, [], 3}  
 {1, 2, vp, know, [], 4}  
 {1, 2, vp, know, [pp], 5}  
 {0, 2, s, [i, know], [], 6}  
 {1, 2, vp, know, [np], 7}  
 {1, 2, vp, know, [np, np], 8}  
 {1, 2, vp, know, [s], 9}  
 {2, 3, det, the, [], 10}  
 {2, 3, np, the, [nn], 11}

{3, 4, noun, man, [], 12}  
 {3, 4, nn, man, [], 13}  
 {3, 4, nn, man, [pp], 14}  
 {2, 4, np, [the, man], [], 15}  
 {2, 4, s, [the, man], [vp], 16}  
 {1, 4, vp, [know, [the, man]], [], 17}  
 {1, 4, vp, [know, [the, man]], [pp], 18}  
 {0, 4, s, [i, [know, [the, man]]], [], 19}  
 {1, 4, vp, [know, [the, man]], [np], 20}  
 {4, 5, verb, saw, [], 21}  
 {4, 5, vp, saw, [], 22}

Adding {4,5,vp,saw,[pp],23} because start of vp==>[vp,pp] has been matched by {4,5,vp,saw,[],22}



{0, 1, pronoun, i, [], 0}  
 {0, 1, np, i, [], 1}  
 {0, 1, s, i, [vp], 2}  
 {1, 2, verb, know, [], 3}  
 {1, 2, vp, know, [], 4}  
 {1, 2, vp, know, [pp], 5}  
 {0, 2, s, [i, know], [], 6}  
 {1, 2, vp, know, [np], 7}  
 {1, 2, vp, know, [np, np], 8}  
 {1, 2, vp, know, [s], 9}  
 {2, 3, det, the, [], 10}  
 {2, 3, np, the, [nn], 11}  
 {3, 4, noun, man, [], 12}

{3, 4, nn, man, [], 13}  
 {3, 4, nn, man, [pp], 14}  
 {2, 4, np, [the, man], [], 15}  
 {2, 4, s, [the, man], [vp], 16}  
 {1, 4, vp, [know, [the, man]], [], 17}  
 {1, 4, vp, [know, [the, man]], [pp], 18}  
 {0, 4, s, [i, [know, [the, man]]], [], 19}  
 {1, 4, vp, [know, [the, man]], [np], 20}  
 {4, 5, verb, saw, [], 21}  
 {4, 5, vp, saw, [], 22}  
 {4, 5, vp, saw, [pp], 23}

Adding {2,5,s,[[the,man],saw],[],24}: {2,4,s,[the,man],[vp],16} wanted {4,5,vp,saw,[],22}

{0, 1, pronoun, i, [], 0}  
 {0, 1, np, i, [], 1}  
 {0, 1, s, i, [vp], 2}  
 {1, 2, verb, know, [], 3}  
 {1, 2, vp, know, [], 4}  
 {1, 2, vp, know, [pp], 5}  
 {0, 2, s, [i, know], [], 6}  
 {1, 2, vp, know, [np], 7}  
 {1, 2, vp, know, [np, np], 8}  
 {1, 2, vp, know, [s], 9}  
 {2, 3, det, the, [], 10}  
 {2, 3, np, the, [nn], 11}  
 {3, 4, noun, man, [], 12}

{3, 4, nn, man, [], 13}  
 {3, 4, nn, man, [pp], 14}  
 {2, 4, np, [the, man], [], 15}  
 {2, 4, s, [the, man], [vp], 16}  
 {1, 4, vp, [know, [the, man]], [], 17}  
 {1, 4, vp, [know, [the, man]], [pp], 18}  
 {0, 4, s, [i, [know, [the, man]]], [], 19}  
 {1, 4, vp, [know, [the, man]], [np], 20}  
 {4, 5, verb, saw, [], 21}  
 {4, 5, vp, saw, [], 22}  
 {4, 5, vp, saw, [pp], 23}  
 {2, 5, s, [[the, man], saw], [], 24}

Adding {1,5,vp,[know,[[the,man],saw]],[],25}: {1,2,vp,know,[s],9} wanted {2,5,s,[[the,man],saw],[],24}

```

{0, 1, pronoun, i, [], 0}
{0, 1, np, i, [], 1}
{0, 1, s, i, [vp], 2}
{1, 2, verb, know, [], 3}
{1, 2, vp, know, [], 4}
{1, 2, vp, know, [pp], 5}
{0, 2, s, [i,know], [], 6}
{1, 2, vp, know, [np], 7}
{1, 2, vp, know, [np,np], 8}
{1, 2, vp, know, [s], 9}
{2, 3, det, the, [], 10}
{2, 3, np, the, [nn], 11}
{3, 4, noun, man, [], 12}
{3, 4, nn, man, [], 13}

```

```

{3, 4, nn, man, [pp], 14}
{2, 4, np, [the,man], [], 15}
{2, 4, s, [the,man], [vp], 16}
{1, 4, vp, [know,[the,man]], [], 17}
{1, 4, vp, [know,[the,man]], [pp], 18}
{0, 4, s, [i,[know,[the,man]]], [], 19}
{1, 4, vp, [know,[the,man]], [np], 20}
{4, 5, verb, saw, [], 21}
{4, 5, vp, saw, [], 22}
{4, 5, vp, saw, [pp], 23}
{2, 5, s, [[the,man],saw], [], 24}
{1, 5, vp, [know,[[the,man],saw]], [], 25}

```

Adding {1,5,vp,[know,[[the,man],saw]], [pp],26} because start of vp==>[vp,pp] has been matched by {1,5,vp,[know,[[the,m

{0, 1, pronoun, i, [], 0}  
 {0, 1, np, i, [], 1}  
 {0, 1, s, i, [vp], 2}  
 {1, 2, verb, know, [], 3}  
 {1, 2, vp, know, [], 4}  
 {1, 2, vp, know, [pp], 5}  
 {0, 2, s, [i, know], [], 6}  
 {1, 2, vp, know, [np], 7}  
 {1, 2, vp, know, [np, np], 8}  
 {1, 2, vp, know, [s], 9}  
 {2, 3, det, the, [], 10}  
 {2, 3, np, the, [nn], 11}  
 {3, 4, noun, man, [], 12}  
 {3, 4, nn, man, [], 13}

{3, 4, nn, man, [pp], 14}  
 {2, 4, np, [the, man], [], 15}  
 {2, 4, s, [the, man], [vp], 16}  
 {1, 4, vp, [know, [the, man]], [], 17}  
 {1, 4, vp, [know, [the, man]], [pp], 18}  
 {0, 4, s, [i, [know, [the, man]]], [], 19}  
 {1, 4, vp, [know, [the, man]], [np], 20}  
 {4, 5, verb, saw, [], 21}  
 {4, 5, vp, saw, [], 22}  
 {4, 5, vp, saw, [pp], 23}  
 {2, 5, s, [[the, man], saw], [], 24}  
 {1, 5, vp, [know, [[the, man], saw]], [], 25}  
 {1, 5, vp, [know, [[the, man], saw]], [pp], 26}

Adding {0, 5, s, [i, [know, [[the, man], saw]]], [], 27}: {0, 1, s, i, [vp], 2} wanted {1, 5, vp, [know, [[the, man], saw]], [], 25}

```

{0, 1, pronoun, i, [], 0}
{0, 1, np, i, [], 1}
{0, 1, s, i, [vp], 2}
{1, 2, verb, know, [], 3}
{1, 2, vp, know, [], 4}
{1, 2, vp, know, [pp], 5}
{0, 2, s, [i, know], [], 6}
{1, 2, vp, know, [np], 7}
{1, 2, vp, know, [np, np], 8}
{1, 2, vp, know, [s], 9}
{2, 3, det, the, [], 10}
{2, 3, np, the, [nn], 11}
{3, 4, noun, man, [], 12}
{3, 4, nn, man, [], 13}
{3, 4, nn, man, [pp], 14}

```

```

{2, 4, np, [the, man], [], 15}
{2, 4, s, [the, man], [vp], 16}
{1, 4, vp, [know, [the, man]], [], 17}
{1, 4, vp, [know, [the, man]], [pp], 18}
{0, 4, s, [i, [know, [the, man]]], [], 19}
{1, 4, vp, [know, [the, man]], [np], 20}
{4, 5, verb, saw, [], 21}
{4, 5, vp, saw, [], 22}
{4, 5, vp, saw, [pp], 23}
{2, 5, s, [[the, man], saw], [], 24}
{1, 5, vp, [know, [[the, man], saw]], [], 25}
{1, 5, vp, [know, [[the, man], saw]], [pp], 26}
{0, 5, s, [i, [know, [[the, man], saw]]], [], 27}

```

Adding {4,5,vp,saw,[np],28} because start of vp==>[verb,np] has been matched by {4,5,verb,saw,[],21}

```

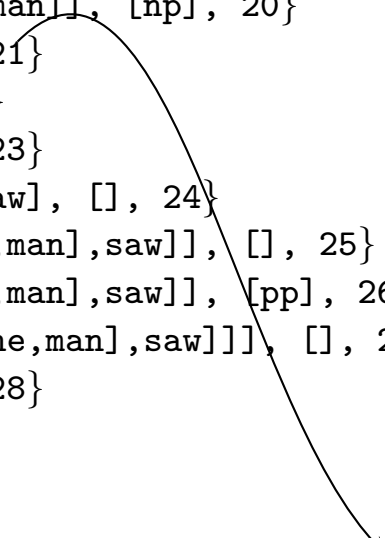
{0, 1, pronoun, i, [], 0}
{0, 1, np, i, [], 1}
{0, 1, s, i, [vp], 2}
{1, 2, verb, know, [], 3}
{1, 2, vp, know, [], 4}
{1, 2, vp, know, [pp], 5}
{0, 2, s, [i, know], [], 6}
{1, 2, vp, know, [np], 7}
{1, 2, vp, know, [np, np], 8}
{1, 2, vp, know, [s], 9}
{2, 3, det, the, [], 10}
{2, 3, np, the, [nn], 11}
{3, 4, noun, man, [], 12}
{3, 4, nn, man, [], 13}
{3, 4, nn, man, [pp], 14}

```

```

{2, 4, np, [the, man], [], 15}
{2, 4, s, [the, man], [vp], 16}
{1, 4, vp, [know, [the, man]], [], 17}
{1, 4, vp, [know, [the, man]], [pp], 18}
{0, 4, s, [i, [know, [the, man]]], [], 19}
{1, 4, vp, [know, [the, man]], [np], 20}
{4, 5, verb, saw, [], 21}
{4, 5, vp, saw, [], 22}
{4, 5, vp, saw, [pp], 23}
{2, 5, s, [[the, man], saw], [], 24}
{1, 5, vp, [know, [[the, man], saw]], [], 25}
{1, 5, vp, [know, [[the, man], saw]], [pp], 26}
{0, 5, s, [i, [know, [[the, man], saw]]], [], 27}
{4, 5, vp, saw, [np], 28}

```



Adding {4,5,vp,saw,[np,np],29} because start of vp=>[verb,np,np] has been matched by {4,5,verb,saw,[],21}

```

{0, 1, pronoun, i, [], 0}
{0, 1, np, i, [], 1}
{0, 1, s, i, [vp], 2}
{1, 2, verb, know, [], 3}
{1, 2, vp, know, [], 4}
{1, 2, vp, know, [pp], 5}
{0, 2, s, [i,know], [], 6}
{1, 2, vp, know, [np], 7}
{1, 2, vp, know, [np,np], 8}
{1, 2, vp, know, [s], 9}
{2, 3, det, the, [], 10}
{2, 3, np, the, [nn], 11}
{3, 4, noun, man, [], 12}
{3, 4, nn, man, [], 13}
{3, 4, nn, man, [pp], 14}
{2, 4, np, [the,man], [], 15}

```

```

{2, 4, s, [the,man], [vp], 16}
{1, 4, vp, [know,[the,man]], [], 17}
{1, 4, vp, [know,[the,man]], [pp], 18}
{0, 4, s, [i,[know,[the,man]]], [], 19}
{1, 4, vp, [know,[the,man]], [np], 20}
{4, 5, verb, saw, [], 21}
{4, 5, vp, saw, [], 22}
{4, 5, vp, saw, [pp], 23}
{2, 5, s, [[the,man],saw], [], 24}
{1, 5, vp, [know,[[the,man],saw]], [], 25}
{1, 5, vp, [know,[[the,man],saw]], [pp], 26}
{0, 5, s, [i,[know,[[the,man],saw]]], [], 27}
{4, 5, vp, saw, [np], 28}
{4, 5, vp, saw, [np,np], 29}

```

Adding {4,5,vp,saw,[s],30} because start of vp==>[verb,s] has been matched by {4,5,verb,saw,[],21}



```

{0, 1, pronoun, i, [], 0}
{0, 1, np, i, [], 1}
{0, 1, s, i, [vp], 2}
{1, 2, verb, know, [], 3}
{1, 2, vp, know, [], 4}
{1, 2, vp, know, [pp], 5}
{0, 2, s, [i, know], [], 6}
{1, 2, vp, know, [np], 7}
{1, 2, vp, know, [np, np], 8}
{1, 2, vp, know, [s], 9}
{2, 3, det, the, [], 10}
{2, 3, np, the, [nn], 11}
{3, 4, noun, man, [], 12}
{3, 4, nn, man, [], 13}
{3, 4, nn, man, [pp], 14}
{2, 4, np, [the, man], [], 15}
{2, 4, s, [the, man], [vp], 16}
{1, 4, vp, [know, [the, man]], [], 17}
{1, 4, vp, [know, [the, man]], [pp], 18}
{0, 4, s, [i, [know, [the, man]]], [], 19}
{1, 4, vp, [know, [the, man]], [np], 20}
{4, 5, verb, saw, [], 21}
{4, 5, vp, saw, [], 22}
{4, 5, vp, saw, [pp], 23}
{2, 5, s, [[the, man], saw], [], 24}
{1, 5, vp, [know, [[the, man], saw]], [], 25}
{1, 5, vp, [know, [[the, man], saw]], [pp], 26}
{0, 5, s, [i, [know, [[the, man], saw]]], [], 27}
{4, 5, vp, saw, [np], 28}
{4, 5, vp, saw, [np, np], 29}
{4, 5, vp, saw, [s], 30}

```

Adding lexical edge {5, 6, pronoun, her, [], 31}

```

{0, 1, pronoun, i, [], 0}
{0, 1, np, i, [], 1}
{0, 1, s, i, [vp], 2}
{1, 2, verb, know, [], 3}
{1, 2, vp, know, [], 4}
{1, 2, vp, know, [pp], 5}
{0, 2, s, [i,know], [], 6}
{1, 2, vp, know, [np], 7}
{1, 2, vp, know, [np,np], 8}
{1, 2, vp, know, [s], 9}
{2, 3, det, the, [], 10}
{2, 3, np, the, [nn], 11}
{3, 4, noun, man, [], 12}
{3, 4, nn, man, [], 13}
{3, 4, nn, man, [pp], 14}
{2, 4, np, [the,man], [], 15}
{2, 4, s, [the,man], [vp], 16}

```

```

{1, 4, vp, [know,[the,man]], [], 17}
{1, 4, vp, [know,[the,man]], [pp], 18}
{0, 4, s, [i,[know,[the,man]]], [], 19}
{1, 4, vp, [know,[the,man]], [np], 20}
{4, 5, verb, saw, [], 21}
{4, 5, vp, saw, [], 22}
{4, 5, vp, saw, [pp], 23}
{2, 5, s, [[the,man],saw], [], 24}
{1, 5, vp, [know,[[the,man],saw]], [], 25}
{1, 5, vp, [know,[[the,man],saw]], [pp], 26}
{0, 5, s, [i,[know,[[the,man],saw]]], [], 27}
{4, 5, vp, saw, [np], 28}
{4, 5, vp, saw, [np,np], 29}
{4, 5, vp, saw, [s], 30}
{5, 6, pronoun, her, [], 31}

```

Adding {5,6,np,her,[],32} because start of np==>[pronoun] has been matched by {5,6,pronoun,her,[],31}

{0, 1, pronoun, i, [], 0}  
 {0, 1, np, i, [], 1}  
 {0, 1, s, i, [vp], 2}  
 {1, 2, verb, know, [], 3}  
 {1, 2, vp, know, [], 4}  
 {1, 2, vp, know, [pp], 5}  
 {0, 2, s, [i, know], [], 6}  
 {1, 2, vp, know, [np], 7}  
 {1, 2, vp, know, [np, np], 8}  
 {1, 2, vp, know, [s], 9}  
 {2, 3, det, the, [], 10}  
 {2, 3, np, the, [nn], 11}  
 {3, 4, noun, man, [], 12}  
 {3, 4, nn, man, [], 13}  
 {3, 4, nn, man, [pp], 14}  
 {2, 4, np, [the, man], [], 15}  
 {2, 4, s, [the, man], [vp], 16}

{1, 4, vp, [know, [the, man]], [], 17}  
 {1, 4, vp, [know, [the, man]], [pp], 18}  
 {0, 4, s, [i, [know, [the, man]]], [], 19}  
 {1, 4, vp, [know, [the, man]], [np], 20}  
 {4, 5, verb, saw, [], 21}  
 {4, 5, vp, saw, [], 22}  
 {4, 5, vp, saw, [pp], 23}  
 {2, 5, s, [[the, man], saw], [], 24}  
 {1, 5, vp, [know, [[the, man], saw]], [], 25}  
 {1, 5, vp, [know, [[the, man], saw]], [pp], 26}  
 {0, 5, s, [i, [know, [[the, man], saw]]], [], 27}  
 {4, 5, vp, saw, [np], 28}  
 {4, 5, vp, saw, [np, np], 29}  
 {4, 5, vp, saw, [s], 30}  
 {5, 6, pronoun, her, [], 31}  
 {5, 6, np, her, [], 32}

Adding {5,6,s,her,[vp],33} because start of s==>[np,vp] has been matched by {5,6,np,her,[],32}

{0, 1, pronoun, i, [], 0}  
 {0, 1, np, i, [], 1}  
 {0, 1, s, i, [vp], 2}  
 {1, 2, verb, know, [], 3}  
 {1, 2, vp, know, [], 4}  
 {1, 2, vp, know, [pp], 5}  
 {0, 2, s, [i, know], [], 6}  
 {1, 2, vp, know, [np], 7}  
 {1, 2, vp, know, [np, np], 8}  
 {1, 2, vp, know, [s], 9}  
 {2, 3, det, the, [], 10}  
 {2, 3, np, the, [nn], 11}  
 {3, 4, noun, man, [], 12}  
 {3, 4, nn, man, [], 13}  
 {3, 4, nn, man, [pp], 14}  
 {2, 4, np, [the, man], [], 15}  
 {2, 4, s, [the, man], [vp], 16}  
 {1, 4, vp, [know, [the, man]], [], 17}

{1, 4, vp, [know, [the, man]], [pp], 18}  
 {0, 4, s, [i, [know, [the, man]]], [], 19}  
 {1, 4, vp, [know, [the, man]], [np], 20}  
 {4, 5, verb, saw, [], 21}  
 {4, 5, vp, saw, [], 22}  
 {4, 5, vp, saw, [pp], 23}  
 {2, 5, s, [[the, man], saw], [], 24}  
 {1, 5, vp, [know, [[the, man], saw]], [], 25}  
 {1, 5, vp, [know, [[the, man], saw]], [pp], 26}  
 {0, 5, s, [i, [know, [[the, man], saw]]], [], 27}  
 {4, 5, vp, saw, [np], 28}  
 {4, 5, vp, saw, [np, np], 29}  
 {4, 5, vp, saw, [s], 30}  
 {5, 6, pronoun, her, [], 31}  
 {5, 6, np, her, [], 32}  
 {5, 6, s, her, [vp], 33}

Adding {4,6,vp,[saw,her],[],34}: {4,5,vp,saw,[np],28} wanted {5,6,np,her,[],32}

```

{0, 1, pronoun, i, [], 0}
{0, 1, np, i, [], 1}
{0, 1, s, i, [vp], 2}
{1, 2, verb, know, [], 3}
{1, 2, vp, know, [], 4}
{1, 2, vp, know, [pp], 5}
{0, 2, s, [i, know], [], 6}
{1, 2, vp, know, [np], 7}
{1, 2, vp, know, [np, np], 8}
{1, 2, vp, know, [s], 9}
{2, 3, det, the, [], 10}
{2, 3, np, the, [nn], 11}
{3, 4, noun, man, [], 12}
{3, 4, nn, man, [], 13}
{3, 4, nn, man, [pp], 14}
{2, 4, np, [the, man], [], 15}
{2, 4, s, [the, man], [vp], 16}
{1, 4, vp, [know, [the, man]], [], 17}

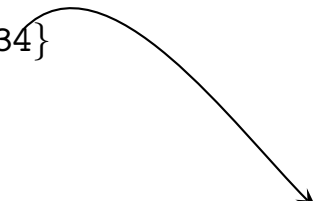
```

```

{1, 4, vp, [know, [the, man]], [pp], 18}
{0, 4, s, [i, [know, [the, man]]], [], 19}
{1, 4, vp, [know, [the, man]], [np], 20}
{4, 5, verb, saw, [], 21}
{4, 5, vp, saw, [], 22}
{4, 5, vp, saw, [pp], 23}
{2, 5, s, [[the, man], saw], [], 24}
{1, 5, vp, [know, [[the, man], saw]], [], 25}
{1, 5, vp, [know, [[the, man], saw]], [pp], 26}
{0, 5, s, [i, [know, [[the, man], saw]]], [], 27}
{4, 5, vp, saw, [np], 28}
{4, 5, vp, saw, [np, np], 29}
{4, 5, vp, saw, [s], 30}
{5, 6, pronoun, her, [], 31}
{5, 6, np, her, [], 32}
{5, 6, s, her, [vp], 33}
{4, 6, vp, [saw, her], [], 34}

```

Adding {4,6,vp,[saw,her],[pp],35} because start of vp==>[vp,pp] has been matched by {4,6,vp,[saw,her],[],34}



{0, 1, pronoun, i, [], 0}	{0, 4, s, [i,[know,[the,man]]], [], 19}
{0, 1, np, i, [], 1}	{1, 4, vp, [know,[the,man]], [np], 20}
{0, 1, s, i, [vp], 2}	{4, 5, verb, saw, [], 21}
{1, 2, verb, know, [], 3}	{4, 5, vp, saw, [], 22}
{1, 2, vp, know, [], 4}	{4, 5, vp, saw, [pp], 23}
{1, 2, vp, know, [pp], 5}	{2, 5, s, [[the,man],saw], [], 24}
{0, 2, s, [i,know], [], 6}	{1, 5, vp, [know,[[the,man],saw]], [], 25}
{1, 2, vp, know, [np], 7}	{1, 5, vp, [know,[[the,man],saw]], [pp], 26}
{1, 2, vp, know, [np,np], 8}	{0, 5, s, [i,[know,[[the,man],saw]]], [], 27}
{1, 2, vp, know, [s], 9}	{4, 5, vp, saw, [np], 28}
{2, 3, det, the, [], 10}	{4, 5, vp, saw, [np,np], 29}
{2, 3, np, the, [nn], 11}	{4, 5, vp, saw, [s], 30}
{3, 4, noun, man, [], 12}	{5, 6, pronoun, her, [], 31}
{3, 4, nn, man, [], 13}	{5, 6, np, her, [], 32}
{3, 4, nn, man, [pp], 14}	{5, 6, s, her, [vp], 33}
{2, 4, np, [the,man], [], 15}	{4, 6, vp, [saw,her], [], 34}
{2, 4, s, [the,man], [vp], 16}	{4, 6, vp, [saw,her], [pp], 35}
{1, 4, vp, [know,[the,man]], [], 17}	
{1, 4, vp, [know,[the,man]], [pp], 18}	

Adding {2,6,s,[[the,man],[saw,her]],[],36}: {2,4,s,[the,man],[vp],16} wanted {4,6,vp,[saw,her],[],34}

{0, 1, pronoun, i, [], 0}  
 {0, 1, np, i, [], 1}  
 {0, 1, s, i, [vp], 2}  
 {1, 2, verb, know, [], 3}  
 {1, 2, vp, know, [], 4}  
 {1, 2, vp, know, [pp], 5}  
 {0, 2, s, [i, know], [], 6}  
 {1, 2, vp, know, [np], 7}  
 {1, 2, vp, know, [np, np], 8}  
 {1, 2, vp, know, [s], 9}  
 {2, 3, det, the, [], 10}  
 {2, 3, np, the, [nn], 11}  
 {3, 4, noun, man, [], 12}  
 {3, 4, nn, man, [], 13}  
 {3, 4, nn, man, [pp], 14}  
 {2, 4, np, [the, man], [], 15}  
 {2, 4, s, [the, man], [vp], 16}  
 {1, 4, vp, [know, [the, man]], [], 17}  
 {1, 4, vp, [know, [the, man]], [pp], 18}

{0, 4, s, [i, [know, [the, man]]], [], 19}  
 {1, 4, vp, [know, [the, man]], [np], 20}  
 {4, 5, verb, saw, [], 21}  
 {4, 5, vp, saw, [], 22}  
 {4, 5, vp, saw, [pp], 23}  
 {2, 5, s, [[the, man], saw], [], 24}  
 {1, 5, vp, [know, [[the, man], saw]], [], 25}  
 {1, 5, vp, [know, [[the, man], saw]], [pp], 26}  
 {0, 5, s, [i, [know, [[the, man], saw]]], [], 27}  
 {4, 5, vp, saw, [np], 28}  
 {4, 5, vp, saw, [np, np], 29}  
 {4, 5, vp, saw, [s], 30}  
 {5, 6, pronoun, her, [], 31}  
 {5, 6, np, her, [], 32}  
 {5, 6, s, her, [vp], 33}  
 {4, 6, vp, [saw, her], [], 34}  
 {4, 6, vp, [saw, her], [pp], 35}  
 {2, 6, s, [[the, man], [saw, her]], [], 36}

Adding {1, 6, vp, [know, [[the, man], [saw, her]]], [], 37}: {1, 2, vp, know, [s], 9} wanted {2, 6, s, [[the, man], [saw, her]], [], 36}

{0, 1, pronoun, i, [], 0}	{1, 4, vp, [know,[the,man]], [np], 20}
{0, 1, np, i, [], 1}	{4, 5, verb, saw, [], 21}
{0, 1, s, i, [vp], 2}	{4, 5, vp, saw, [], 22}
{1, 2, verb, know, [], 3}	{4, 5, vp, saw, [pp], 23}
{1, 2, vp, know, [], 4}	{2, 5, s, [[the,man],saw], [], 24}
{1, 2, vp, know, [pp], 5}	{1, 5, vp, [know,[[the,man],saw]], [], 25}
{0, 2, s, [i,know], [], 6}	{1, 5, vp, [know,[[the,man],saw]], [pp], 26}
{1, 2, vp, know, [np], 7}	{0, 5, s, [i,[know,[[the,man],saw]]], [], 27}
{1, 2, vp, know, [np,np], 8}	{4, 5, vp, saw, [np], 28}
{1, 2, vp, know, [s], 9}	{4, 5, vp, saw, [np,np], 29}
{2, 3, det, the, [], 10}	{4, 5, vp, saw, [s], 30}
{2, 3, np, the, [nn], 11}	{5, 6, pronoun, her, [], 31}
{3, 4, noun, man, [], 12}	{5, 6, np, her, [], 32}
{3, 4, nn, man, [], 13}	{5, 6, s, her, [vp], 33}
{3, 4, nn, man, [pp], 14}	{4, 6, vp, [saw,her], [], 34}
{2, 4, np, [the,man], [], 15}	{4, 6, vp, [saw,her], [pp], 35}
{2, 4, s, [the,man], [vp], 16}	{2, 6, s, [[the,man],[saw,her]], [], 36}
{1, 4, vp, [know,[the,man]], [], 17}	{1, 6, vp, [know,[[the,man],[saw,her]]], [], 37}
{1, 4, vp, [know,[the,man]], [pp], 18}	
{0, 4, s, [i,[know,[the,man]]], [], 19}	

Adding {1,6,vp,[know,[[the,man],[saw,her]]],[pp],38} because start of vp==>[vp,pp] has been matched by {1,6,vp,[know,[



{0, 1, pronoun, i, [], 0}	{1, 4, vp, [know, [the, man]], [np], 20}
{0, 1, np, i, [], 1}	{4, 5, verb, saw, [], 21}
{0, 1, s, i, [vp], 2}	{4, 5, vp, saw, [], 22}
{1, 2, verb, know, [], 3}	{4, 5, vp, saw, [pp], 23}
{1, 2, vp, know, [], 4}	{2, 5, s, [[the, man], saw], [], 24}
{1, 2, vp, know, [pp], 5}	{1, 5, vp, [know, [[the, man], saw]], [], 25}
{0, 2, s, [i, know], [], 6}	{1, 5, vp, [know, [[the, man], saw]], [pp], 26}
{1, 2, vp, know, [np], 7}	{0, 5, s, [i, [know, [[the, man], saw]]], [], 27}
{1, 2, vp, know, [np, np], 8}	{4, 5, vp, saw, [np], 28}
{1, 2, vp, know, [s], 9}	{4, 5, vp, saw, [np, np], 29}
{2, 3, det, the, [], 10}	{4, 5, vp, saw, [s], 30}
{2, 3, np, the, [nn], 11}	{5, 6, pronoun, her, [], 31}
{3, 4, noun, man, [], 12}	{5, 6, np, her, [], 32}
{3, 4, nn, man, [], 13}	{5, 6, s, her, [vp], 33}
{3, 4, nn, man, [pp], 14}	{4, 6, vp, [saw, her], [], 34}
{2, 4, np, [the, man], [], 15}	{4, 6, vp, [saw, her], [pp], 35}
{2, 4, s, [the, man], [vp], 16}	{2, 6, s, [[the, man], [saw, her]], [], 36}
{1, 4, vp, [know, [the, man]], [], 17}	{1, 6, vp, [know, [[the, man], [saw, her]]], [], 37}
{1, 4, vp, [know, [the, man]], [pp], 18}	{1, 6, vp, [know, [[the, man], [saw, her]]], [pp], 38}
{0, 4, s, [i, [know, [the, man]]], [], 19}	

Adding {0, 6, s, [i, [know, [[the, man], [saw, her]]]], [], 39}: {0, 1, s, i, [vp], 2} wanted {1, 6, vp, [know, [[the, man], [saw, her]]], []}

One using categorial grammar

```
| ?- chart([i, saw, the, man], P).
```

```
Adding lexical edge edge(0, 1, sign(syntax(head(cat(np),A,B,C),args([])),D,E), i, [], 0)
```

edge(0,1,sign(syntax(head(cat(np))),args([]))),i,[],0))

Adding edge(0,1,sign(syntax(args)),i,[sign(syntax(args([sign(syntax(head(cat(np))),args([])),structure(dir(before)))|\_2  
because start of

(sign(syntax(args))

==> [sign(syntax(head(cat(np))), args([])),  
structure(dir(before))),

sign(syntax(args([sign(syntax(head(cat(np))), args([])),

structure(dir(before))) | A]])) has been matched by

```
edge(0,1,sign(syntax(head(cat(np)),args([]))),i,[],0))
```

```
edge(0,1,sign(syntax(args)),i,[sign(syntax(args([sign(syntax(head(cat(np)),args([])),structure(dir(before)))|A])))],1)
```

```
Adding lexical edge edge(1, 2, sign(syntax(head(cat(v),A,B,C),args([sign(syntax(head(cat(np),D,E,F),args([])),structure
```

```
edge(0,1,sign(syntax(head(cat(np))),args([]))),i,[],0))
```

```
edge(0,1,sign(syntax(args)),i,[sign(syntax(args([sign(syntax(head(cat(np))),args([])),structure(dir(before)))|A]))],1)
```

```
edge(1,2,sign(syntax(head(cat(v))),args([sign(syntax(head(cat(np))),args([])),structure(dir(after))),sign(syntax(head(ca
```

Adding edge(1,2,sign(syntax(head(cat(v))),args([sign(syntax(head(cat(np))),args([])),structure(dir(before))))),saw,[si  
because start of

```
(sign(syntax(head(cat(v))),
```

```
    args([sign(syntax(head(cat(np))), args([])),  
          structure(dir(before))]))))
```

```
==> [sign(syntax(head(cat(v)),  
    args([sign(syntax(head(cat(np))), args([])),  
          structure(dir(after))),  
    sign(syntax(head(cat(np))), args([])),  
    structure(dir(before)))]),  
    semantics(tverb(see))),  
sign(syntax(head(cat(np))), args([])),  
structure(dir(after)))] has been matched by
```

```

edge(0,1,sign(syntax(head(cat(np))),args([]))),i,[],0))
edge(0,1,sign(syntax(args)),i,[sign(syntax(args([sign(syntax(head(cat(np))),args([])),structure(dir(before)))|A)))]),1)
edge(1,2,sign(syntax(head(cat(v))),args([sign(syntax(head(cat(np))),args([])),structure(dir(after))),sign(syntax(head(ca
edge(1,2,sign(syntax(head(cat(v))),args([sign(syntax(head(cat(np))),args([])),structure(dir(before)))))),saw,[sign(synt

Adding edge(1,2,sign(syntax(args)),saw,[sign(syntax(args([sign(syntax(head(cat(v))),args([sign(syntax(head(cat(np))),arg
because start of
(sign(syntax(args))
==> [sign(syntax(head(cat(v)),
      args([sign(syntax(head(cat(np))), args([])),
        structure(dir(after))),
        sign(syntax(head(cat(np))), args([])),
        structure(dir(before)))))),
      structure(dir(before)),
      semantics(tverb(see))),
sign(syntax(args([sign(syntax(head(cat(v)),
      args([sign(syntax(head(cat(np))),
        args([])),
        structure(dir(after))),
        sign(syntax(head(cat(np))),
        args([])),
        structure(dir(before)))))),
      structure(dir(before)),
      semantics(tverb(see))) | A]])) has been matched by

```



```

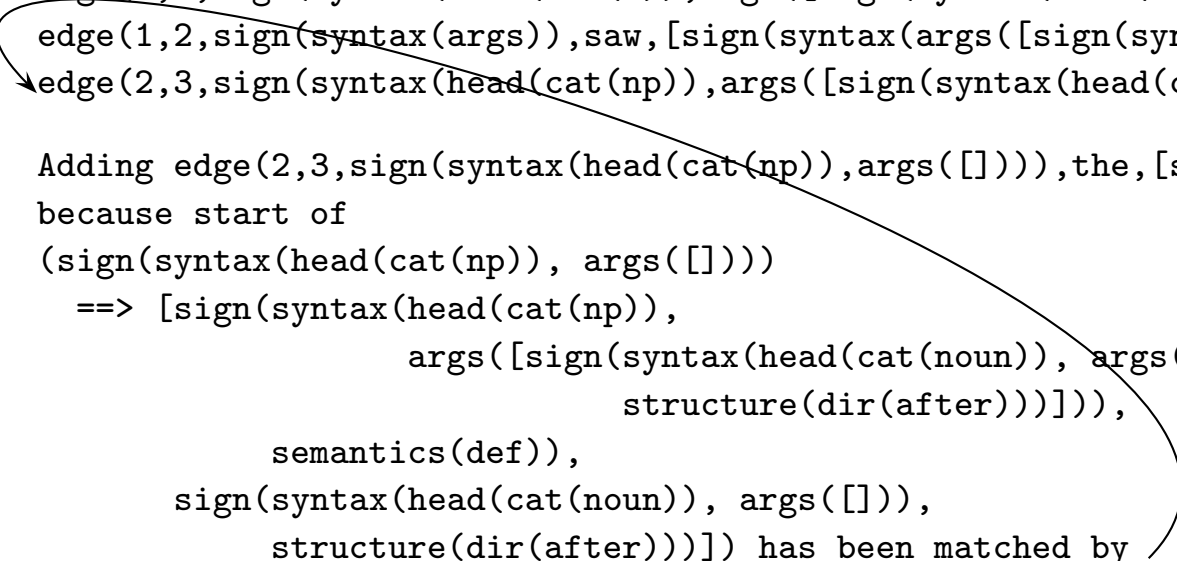
edge(0,1,sign(syntax(head(cat(np)),args([]))),i,[],0))
edge(0,1,sign(syntax(args)),i,[sign(syntax(args([sign(syntax(head(cat(np)),args([])),structure(dir(before)))|A]))),1)
edge(1,2,sign(syntax(head(cat(v)),args([sign(syntax(head(cat(np)),args([])),structure(dir(after))),sign(syntax(head(ca
edge(1,2,sign(syntax(head(cat(v)),args([sign(syntax(head(cat(np)),args([])),structure(dir(before)))))),saw,[sign(synt
edge(1,2,sign(syntax(args)),saw,[sign(syntax(args([sign(syntax(head(cat(v)),args([sign(syntax(head(cat(np)),args([])),
Adding lexical edge edge(2, 3, sign(syntax(head(cat(np),A,B,C),args([sign(syntax(head(cat(noun),D,E,F),args([])),struc

```

```

edge(0,1,sign(syntax(head(cat(np))),args([]))),i,[],0))
edge(0,1,sign(syntax(args)),i,[sign(syntax(args([sign(syntax(head(cat(np))),args([])),structure(dir(before)))|A))))],1)
edge(1,2,sign(syntax(head(cat(v))),args([sign(syntax(head(cat(np))),args([])),structure(dir(after))),sign(syntax(head(ca
edge(1,2,sign(syntax(head(cat(v))),args([sign(syntax(head(cat(np))),args([])),structure(dir(before)))))),saw,[sign(synt
edge(1,2,sign(syntax(args)),saw,[sign(syntax(args([sign(syntax(head(cat(v))),args([sign(syntax(head(cat(np))),args([])),
edge(2,3,sign(syntax(head(cat(np))),args([sign(syntax(head(cat(noun))),args([])),structure(dir(after)))))),semantics(def

```



Adding edge(2,3,sign(syntax(head(cat(np))),args([]))),the,[sign(syntax(head(cat(noun))),args([])),structure(dir(after)))  
because start of

```

(sign(syntax(head(cat(np))), args([])))
==> [sign(syntax(head(cat(np))),
        args([sign(syntax(head(cat(noun))), args([])),
              structure(dir(after))))]),
      semantics(def)),
sign(syntax(head(cat(noun))), args([])),
structure(dir(after)))] has been matched by

```



```

edge(0,1,sign(syntax(head(cat(np))),args([]))),i,[],0))
edge(0,1,sign(syntax(args)),i,[sign(syntax(args([sign(syntax(head(cat(np))),args([])),structure(dir(before)))|A)))]),1)
edge(1,2,sign(syntax(head(cat(v))),args([sign(syntax(head(cat(np))),args([])),structure(dir(after))),sign(syntax(head(ca
edge(1,2,sign(syntax(head(cat(v))),args([sign(syntax(head(cat(np))),args([])),structure(dir(before)))))),saw,[sign(synt
edge(1,2,sign(syntax(args)),saw,[sign(syntax(args([sign(syntax(head(cat(v))),args([sign(syntax(head(cat(np))),args([])),
edge(2,3,sign(syntax(head(cat(np))),args([sign(syntax(head(cat(noun))),args([])),structure(dir(after)))))),semantics(def
edge(2,3,sign(syntax(head(cat(np))),args([]))),the,[sign(syntax(head(cat(noun))),args([])),structure(dir(after)))]),6))

```

Adding edge(2,3,sign(syntax(args)),the,[sign(syntax(args([sign(syntax(head(cat(np))),args([sign(syntax(head(cat(noun))),

because start of

```

(sign(syntax(args))
==> [sign(syntax(head(cat(np))),
      args([sign(syntax(head(cat(noun))), args([])),
            structure(dir(after)))])),
      structure(dir(before)),
      semantics(def)),
sign(syntax(args([sign(syntax(head(cat(np))),
                    args([sign(syntax(head(cat(noun))),
                          args([])),
                          structure(dir(after)))])),
      structure(dir(before)),
      semantics(def)) | A)))])) has been matched by

```

```

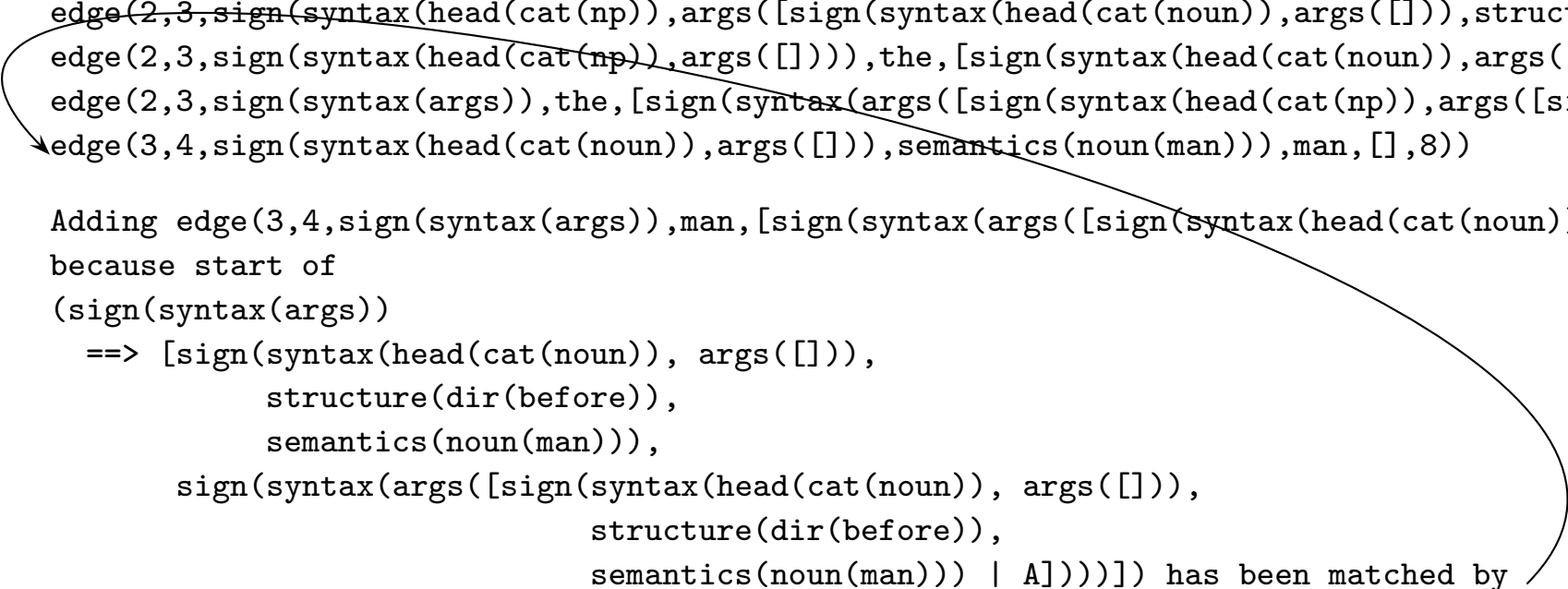
edge(0,1,sign(syntax(head(cat(np))),args([]))),i,[],0))
edge(0,1,sign(syntax(args)),i,[sign(syntax(args([sign(syntax(head(cat(np))),args([])),structure(dir(before)))|A))))],1)
edge(1,2,sign(syntax(head(cat(v))),args([sign(syntax(head(cat(np))),args([])),structure(dir(after))),sign(syntax(head(ca
edge(1,2,sign(syntax(head(cat(v))),args([sign(syntax(head(cat(np))),args([])),structure(dir(before)))))),saw,[sign(synt
edge(1,2,sign(syntax(args)),saw,[sign(syntax(args([sign(syntax(head(cat(v))),args([sign(syntax(head(cat(np))),args([])),
edge(2,3,sign(syntax(head(cat(np))),args([sign(syntax(head(cat(noun))),args([])),structure(dir(after)))))),semantics(def
edge(2,3,sign(syntax(head(cat(np))),args([]))),the,[sign(syntax(head(cat(noun))),args([])),structure(dir(after))],6))
edge(2,3,sign(syntax(args)),the,[sign(syntax(args([sign(syntax(head(cat(np))),args([sign(syntax(head(cat(noun))),args([
Adding lexical edge edge(3, 4, sign(syntax(head(cat(noun),A,B,C),args([])),D,semantics(noun(man))), man, [], 8)

```

```

edge(0,1,sign(syntax(head(cat(np))),args([]))),i,[],0))
edge(0,1,sign(syntax(args)),i,[sign(syntax(args([sign(syntax(head(cat(np))),args([])),structure(dir(before)))|A)))]),1)
edge(1,2,sign(syntax(head(cat(v))),args([sign(syntax(head(cat(np))),args([])),structure(dir(after))),sign(syntax(head(ca
edge(1,2,sign(syntax(head(cat(v))),args([sign(syntax(head(cat(np))),args([])),structure(dir(before)))))),saw,[sign(synt
edge(1,2,sign(syntax(args)),saw,[sign(syntax(args([sign(syntax(head(cat(v))),args([sign(syntax(head(cat(np))),args([])),
edge(2,3,sign(syntax(head(cat(np))),args([sign(syntax(head(cat(noun))),args([])),structure(dir(after))))),semantics(def
edge(2,3,sign(syntax(head(cat(np))),args([]))),the,[sign(syntax(head(cat(noun))),args([])),structure(dir(after)))]),6))
edge(2,3,sign(syntax(args)),the,[sign(syntax(args([sign(syntax(head(cat(np))),args([sign(syntax(head(cat(noun))),args([]
edge(3,4,sign(syntax(head(cat(noun))),args([])),semantics(noun(man))),man,[],8))

```



Adding edge(3,4,sign(syntax(args)),man,[sign(syntax(args([sign(syntax(head(cat(noun))),args([])),structure(dir(before))
because start of
(sign(syntax(args))
=> [sign(syntax(head(cat(noun))), args([])),
structure(dir(before)),
semantics(noun(man))),
sign(syntax(args([sign(syntax(head(cat(noun))), args([])),
structure(dir(before)),
semantics(noun(man))) | A)))])) has been matched by

```

edge(0,1,sign(syntax(head(cat(np))),args([]))),i,[],0))
edge(0,1,sign(syntax(args)),i,[sign(syntax(args([sign(syntax(head(cat(np))),args([])),structure(dir(before)))|A))))],1)
edge(1,2,sign(syntax(head(cat(v))),args([sign(syntax(head(cat(np))),args([])),structure(dir(after))),sign(syntax(head(ca
edge(1,2,sign(syntax(head(cat(v))),args([sign(syntax(head(cat(np))),args([])),structure(dir(before)))))),saw,[sign(synt
edge(1,2,sign(syntax(args)),saw,[sign(syntax(args([sign(syntax(head(cat(v))),args([sign(syntax(head(cat(np))),args([])),
edge(2,3,sign(syntax(head(cat(np))),args([sign(syntax(head(cat(noun))),args([])),structure(dir(after)))))),semantics(def
edge(2,3,sign(syntax(head(cat(np))),args([]))),the,[sign(syntax(head(cat(noun))),args([])),structure(dir(after))],6))
edge(2,3,sign(syntax(args)),the,[sign(syntax(args([sign(syntax(head(cat(np))),args([sign(syntax(head(cat(noun))),args([]
edge(3,4,sign(syntax(head(cat(noun))),args([])),semantics(noun(man))),man,[],8))
edge(3,4,sign(syntax(args)),man,[sign(syntax(args([sign(syntax(head(cat(noun))),args([])),structure(dir(before))),semant

Adding edge(2,4,sign(syntax(head(cat(np))),args([]))),[the,man],[],10)
because edge(2,3,sign(syntax(head(cat(np))),args([]))),the,[sign(syntax(head(cat(noun))),args([])),structure(dir(after))
wanted edge(3,4,sign(syntax(head(cat(noun))),args([])),structure(dir(after)),semantics(noun(man))),man,[],8)

```

```

edge(0,1,sign(syntax(head(cat(np))),args([]))),i,[],0))
edge(0,1,sign(syntax(args)),i,[sign(syntax(args([sign(syntax(head(cat(np))),args([])),structure(dir(before)))|A)))]),1)
edge(1,2,sign(syntax(head(cat(v))),args([sign(syntax(head(cat(np))),args([])),structure(dir(after))),sign(syntax(head(ca
edge(1,2,sign(syntax(head(cat(v))),args([sign(syntax(head(cat(np))),args([])),structure(dir(before)))))),saw,[sign(synt
edge(1,2,sign(syntax(args)),saw,[sign(syntax(args([sign(syntax(head(cat(v))),args([sign(syntax(head(cat(np))),args([])),
edge(2,3,sign(syntax(head(cat(np))),args([sign(syntax(head(cat(noun))),args([])),structure(dir(after)))))),semantics(def
edge(2,3,sign(syntax(head(cat(np))),args([]))),the,[sign(syntax(head(cat(noun))),args([])),structure(dir(after))],6))
edge(2,3,sign(syntax(args)),the,[sign(syntax(args([sign(syntax(head(cat(np))),args([sign(syntax(head(cat(noun))),args([]
edge(3,4,sign(syntax(head(cat(noun))),args([]))),semantics(noun(man))),man,[],8))
edge(3,4,sign(syntax(args)),man,[sign(syntax(args([sign(syntax(head(cat(noun))),args([])),structure(dir(before))),semant
edge(2,4,sign(syntax(head(cat(np))),args([]))),[the,man],[],10))

```

Adding edge(2,4,sign(syntax(args)),[the,man],[sign(syntax(args([sign(syntax(head(cat(np))),args([])),structure(dir(befo  
because start of

(sign(syntax(args))

==> [sign(syntax(head(cat(np)), args([])),  
structure(dir(before))),

sign(syntax(args([sign(syntax(head(cat(np)), args([])),

structure(dir(before))) | A)))] has been matched by

```

edge(0,1,sign(syntax(head(cat(np))),args([]))),i,[],0))
edge(0,1,sign(syntax(args)),i,[sign(syntax(args([sign(syntax(head(cat(np))),args([])),structure(dir(before)))|A]))],1)
edge(1,2,sign(syntax(head(cat(v))),args([sign(syntax(head(cat(np))),args([])),structure(dir(after))),sign(syntax(head(ca
edge(1,2,sign(syntax(head(cat(v))),args([sign(syntax(head(cat(np))),args([])),structure(dir(before)))))),saw,[sign(synt
edge(1,2,sign(syntax(args)),saw,[sign(syntax(args([sign(syntax(head(cat(v))),args([sign(syntax(head(cat(np))),args([])),
edge(2,3,sign(syntax(head(cat(np))),args([sign(syntax(head(cat(noun))),args([])),structure(dir(after)))))),semantics(def
edge(2,3,sign(syntax(head(cat(np))),args([]))),the,[sign(syntax(head(cat(noun))),args([])),structure(dir(after))],6))
edge(2,3,sign(syntax(args)),the,[sign(syntax(args([sign(syntax(head(cat(np))),args([sign(syntax(head(cat(noun))),args([
edge(3,4,sign(syntax(head(cat(noun))),args([])),semantics(noun(man))),man,[],8))
edge(3,4,sign(syntax(args)),man,[sign(syntax(args([sign(syntax(head(cat(noun))),args([])),structure(dir(before))),semant
edge(2,4,sign(syntax(head(cat(np))),args([]))),[the,man],[],10))
edge(2,4,sign(syntax(args)),[the,man],[sign(syntax(args([sign(syntax(head(cat(np))),args([])),structure(dir(before)))|A

```

Adding

```

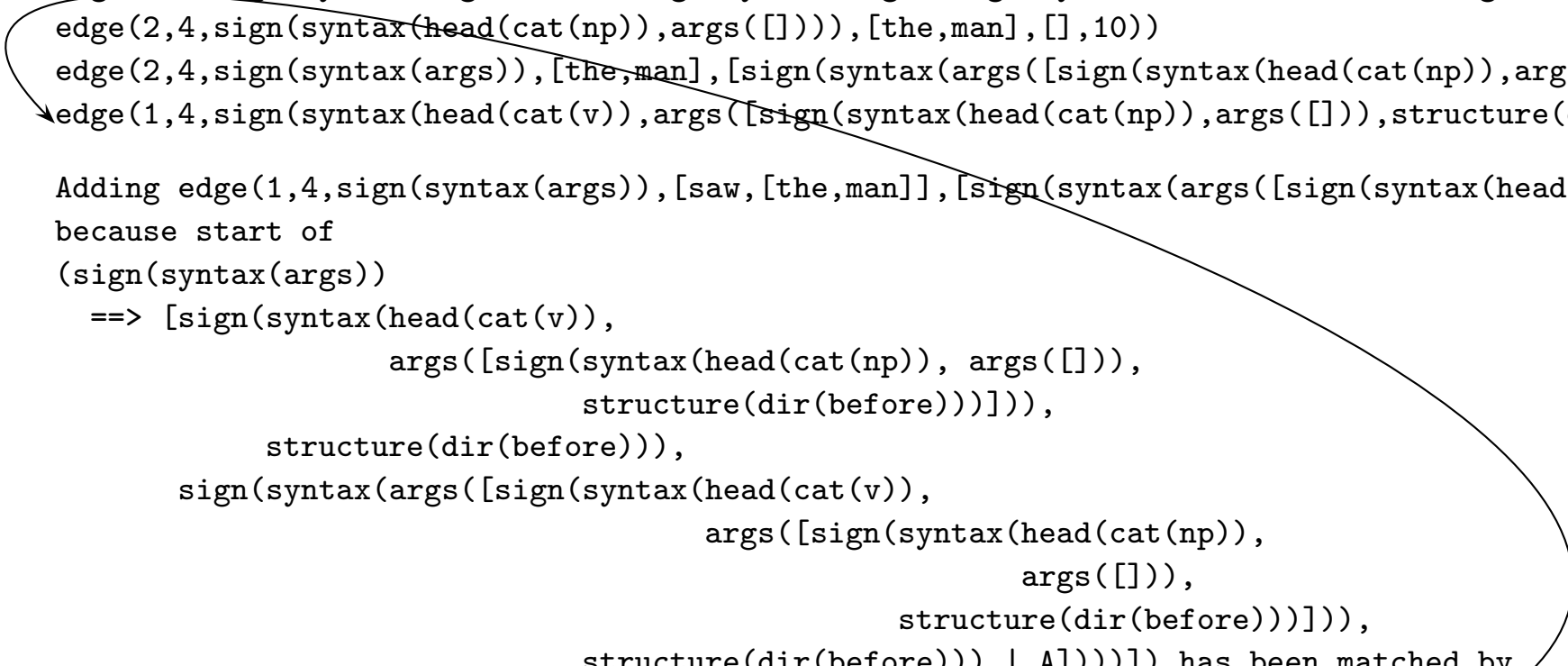
edge(1,4,sign(syntax(head(cat(v))),args([sign(syntax(head(cat(np))),args([])),structure(dir(before)))))),[saw,[t
because
edge(1,2,sign(syntax(head(cat(v))),args([sign(syntax(head(cat(np))),args([])),structure(dir(before)))))),saw,[s
wanted
edge(2,4,sign(syntax(head(cat(np))),args([])),structure(dir(after))),[the,man],[],10)

```

```

edge(0,1,sign(syntax(head(cat(np))),args([]))),i,[],0))
edge(0,1,sign(syntax(args)),i,[sign(syntax(args([sign(syntax(head(cat(np))),args([])),structure(dir(before)))|A)))]),1)
edge(1,2,sign(syntax(head(cat(v))),args([sign(syntax(head(cat(np))),args([])),structure(dir(after))),sign(syntax(head(ca
edge(1,2,sign(syntax(head(cat(v))),args([sign(syntax(head(cat(np))),args([])),structure(dir(before)))))),saw,[sign(synt
edge(1,2,sign(syntax(args)),saw,[sign(syntax(args([sign(syntax(head(cat(v))),args([sign(syntax(head(cat(np))),args([])),
edge(2,3,sign(syntax(head(cat(np))),args([sign(syntax(head(cat(noun))),args([])),structure(dir(after)))))),semantics(def
edge(2,3,sign(syntax(head(cat(np))),args([]))),the,[sign(syntax(head(cat(noun))),args([])),structure(dir(after)))]),6))
edge(2,3,sign(syntax(args)),the,[sign(syntax(args([sign(syntax(head(cat(np))),args([sign(syntax(head(cat(noun))),args([]
edge(3,4,sign(syntax(head(cat(noun))),args([])),semantics(noun(man))),man,[],8))
edge(3,4,sign(syntax(args)),man,[sign(syntax(args([sign(syntax(head(cat(noun))),args([])),structure(dir(before))),semant
edge(2,4,sign(syntax(head(cat(np))),args([]))),[the,man],[],10))
edge(2,4,sign(syntax(args)),[the,man],[sign(syntax(args([sign(syntax(head(cat(np))),args([])),structure(dir(before)))|A
edge(1,4,sign(syntax(head(cat(v))),args([sign(syntax(head(cat(np))),args([])),structure(dir(before)))))),[saw,[the,man]

```



Adding edge(1,4,sign(syntax(args)),[saw,[the,man]],[sign(syntax(args([sign(syntax(head(cat(v))),args([sign(syntax(head(

because start of

(sign(syntax(args))

```

==> [sign(syntax(head(cat(v)),
        args([sign(syntax(head(cat(np))), args([])),
            structure(dir(before))))]),
    structure(dir(before))),
sign(syntax(args([sign(syntax(head(cat(v)),
        args([sign(syntax(head(cat(np))),
            args([])),
                structure(dir(before))))])),
    structure(dir(before))) | A)))] has been matched by

```

```

edge(0,1,sign(syntax(head(cat(np))),args([]))),i,[],0))
edge(0,1,sign(syntax(args)),i,[sign(syntax(args([sign(syntax(head(cat(np))),args([])),structure(dir(before)))|A]))],1)
edge(1,2,sign(syntax(head(cat(v))),args([sign(syntax(head(cat(np))),args([])),structure(dir(after))),sign(syntax(head(ca
edge(1,2,sign(syntax(head(cat(v))),args([sign(syntax(head(cat(np))),args([])),structure(dir(before)))))),saw,[sign(synt
edge(1,2,sign(syntax(args)),saw,[sign(syntax(args([sign(syntax(head(cat(v))),args([sign(syntax(head(cat(np))),args([])),
edge(2,3,sign(syntax(head(cat(np))),args([sign(syntax(head(cat(noun))),args([])),structure(dir(after)))))),semantics(def
edge(2,3,sign(syntax(head(cat(np))),args([]))),the,[sign(syntax(head(cat(noun))),args([])),structure(dir(after))],6))
edge(2,3,sign(syntax(args)),the,[sign(syntax(args([sign(syntax(head(cat(np))),args([sign(syntax(head(cat(noun))),args([
edge(3,4,sign(syntax(head(cat(noun))),args([])),semantics(noun(man))),man,[],8))
edge(3,4,sign(syntax(args)),man,[sign(syntax(args([sign(syntax(head(cat(noun))),args([])),structure(dir(before))),semant
edge(2,4,sign(syntax(head(cat(np))),args([]))),[the,man],[],10))
edge(2,4,sign(syntax(args)),[the,man],[sign(syntax(args([sign(syntax(head(cat(np))),args([])),structure(dir(before)))|A
edge(1,4,sign(syntax(head(cat(v))),args([sign(syntax(head(cat(np))),args([])),structure(dir(before)))))),[saw,[the,man]
edge(1,4,sign(syntax(args)),[saw,[the,man]],[sign(syntax(args([sign(syntax(head(cat(v))),args([sign(syntax(head(cat(np))

```

Adding `edge(0,4,sign(syntax(head(cat(v))),args([]))),[i,[saw,[the,man]]],[],14)`

because `edge(0,1,sign(syntax(head(cat(v))),args([]))),i,[sign(syntax(head(cat(v))),args([sign(syntax(head(cat(np))),args(`  
wanted `edge(1,4,sign(syntax(head(cat(v))),args([sign(syntax(head(cat(np))),args([])),structure(dir(before)))))),[saw,[t`



- Basic top-down and bottom-up parsing algorithms, and places where they make decisions (so places where they may make **wrong** decisions)
- The advantages of chart-parsing: the fundamental rule of chart parsing, **the steps carried out by a chart parser for a simple example**

- What's it all about anyway?
- Where does semantics fit?
- Sources of evidence

# What's it all about?

---

Why are we interested in speech and language anyway?

Because **people** use them to **communicate ideas**.

So if you want a computer system to do anything with language, you must be expecting it do something with ideas.

Semantics = the way that language encodes ideas.

Doing something with ideas = inference.

# How does language work?

---

Structural choices – lexical choices, syntactic organisation, phonology

The 'ideas' that those choices denote

What somebody who accepted those ideas might reasonably believe

Figure 1: The pipeline view of language

The **only** reason we're interested in speech (or text) is because spoken (written) utterances convey ideas.

The **only** reason for doing any of the structural analysis is because the structural choices have semantic consequences.

We may get the structural analysis wrong, or it may not have a unique solution. **The goal of semantics is not to help you get the structural analysis right.**

## Doing something with ideas

---

How can you tell whether a semantic theory is right? If it has the right consequences (this will, at least, tell you if it's not wrong).

- Consequences, contradictions
- Distinctions
- Ambiguities

How can a computer program find out the consequences of something? If it's expressed in a formal language with a notion of inference rule, i.e. in a '**logic**' (note: in a logic, not in logic).

Where can you get the data from?

- In your head?
- In conversation with your mates? With other linguists?
- In corpora?

A formal language which is less expressive than natural language will be inadequate. You can't use propositional logic, for instance, because natural language allows you to say '*All students are lazy*' and '*Some lecturers are clever.*'

But as formal languages get more expressive, they get more intractable.



Attribute:value pairs Database languages	linear
Propositional logic Description logic	NP-complete
Predicate logic Modal logic	Recursively enumerable
Default logic	Undecidable
Property theory Set theory Intensional logic	Incomplete

What shall we do about it?

- Panic: use some language whose complexity results are reassuring, at the cost of missing out some of the content of what's been said. Still, NP-complete is usually taken to be pretty hopeless, so we can't even use propositional logic.
- Don't panic: after all, arithmetic is incomplete, but that doesn't stop us using computers to calculate our tax returns. The **logic** may be intractable, but that doesn't mean that people set each other intractable problems when using it.

I would rather see what happens when I try to get it right, and then compromise when I find I haven't got a suitable inference engine, than give up without even trying.

‘The meaning of the whole is a function of the meaning of the parts and their mode of combination’ (Dowty et al. 1981)

In other words, a particular set of words, configured in a particular way, always makes the same contribution, though what you **do** with that contribution in a given context is up to you.

The theory will be elegant if it is compositional?

The theory will be modular if it is compositional!

The only real problem with wanting a compositional theory of semantics is that you can't have one.

- **'Lexical ambiguity'**: I don't know what the parts are.

(32) a. i. I keep my money tied up at the bank

ii. I keep my boat tied up at the bank.

b. i. There's a wasp in the jam.

ii. I got caught up in a jam on the way in this morning.

- ‘**Structural ambiguity**’: I don’t know what the mode of combination is.

(33) a. i. I saw the man with a big nose.  
          ii. I saw the moon with a telescope.  
      b. i. strawberry jam jar, glass jam jar

- ‘**Scope ambiguity**’: I know the words, I know the structure: and I still don’t know the meaning.

(34) a. John is looking for a unicorn.  
      b. Someone is mugged every five minutes in New York,  
          and he’s getting pretty sick of it.

I must have some idea what I think the representation of a given sentence should be. Who knows where I got it, or whether it's right, but I must at least know what I think or I can't do anything.



So for instance I might think that the correct representation of

(35) John loves Mary

is `love(john, mary)`.

In which case I probably think that the correct representation of

(36) Peter loves Mary

is `love(peter, mary)`

OK, so what do these representations have in common and how do they differ?

They have in common a bit that looks like `love(X, mary)`, and they differ in that one mentions `john` and the other mentions `peter`.

What do the sentences have in common and how do they differ?

(35) has '*John*' as its subject and '*loves Mary*' as its VP, and (36) has '*Peter*' as its subject and the same VP.

So I'm quite tempted by a standard rewrite rule like the following:

```
s(S) ==>
  [np(NP),
   vp(VP)] :-
  S is obtained by using NP to fill in the
  blank in VP
```

What about the VP itself? If '*loves Mary*' means  $\text{love}(X, \text{mary})$  then presumably '*loves Susan*' means  $\text{love}(X, \text{susan})$ . Again what they have in common is the bit about loving and where they differ is in who is being loved, so we could assume that '*loves*' means  $\text{love}(X, Y)$  and that '*Mary*' and '*Susan*' just mean *mary* and *susan*.

So this time I want

$vp(VP) ==>$

$[verb(V),$

$np(NP)] :-$

    VP is obtained by using NP to fill in the  
    blank in V

But V is  $love(X, Y)$ , which has **two** blanks in it. Which one do I want to fill in first?

Write it as `lambda(Y, lambda(X, love(X, Y)))` to fix the order in which they get filled in. Then we have the crucial rule:

$$\text{lambda}(X, P):T = P_{T/X} \quad (\text{TB})$$

(written in mathematical notation as  $\lambda X(P).T$ , and sometimes written in Prolog as `X~P:T`).

TB – the ‘**Tarski biconditional**’ will get us into trouble later, but for now we just accept it. **Dead easy to implement.** Dead hard to do by hand: which you have to do when you’re developing a system.

Example from catgramm

I start by working out what I want it to mean: then I assign parts of that to the words.

Suppose I decide I want '*John sleeps*' to mean  
`exists(A, event(sleep, A, john))`

```
| ?- chart([john, sleeps], P), getMeaning(P, M).
```

Meaning as codes:

```
(iverb(sleep) : name(john))
```

Decoded:

```
(lambda(A, (A : lambda(B,exists(C,event(sleep,C,B))))))  
  : lambda(D, (D : john)))
```

```
(lambda(A,  
      (A  
        : lambda(B, exists(C, event(sleep, C, B))))))  
  : lambda(D,D:john))
```

```
(lambda(D, (D : john))  
  : lambda(B,exists(C,event(sleep,C,B))))
```



```
(lambda(B,  
      exists(C, event(sleep, C, B)))  
  : john)
```

```
exists(C, event(sleep, C, john))
```

Reduced:

```
exists(C, event(sleep, C, john))
```

Hard to work with, but very flexible and each constituent has a well-defined interpretation. Introduced by Montague (see (Dowty et al. 1981)), used, with variations, by lots of people (e.g. (van Eijck and Alshawi 1992; van Genabith and Crouch 1997; Gazdar et al. 1985; Groenendijk and Stokhof 1984; Kamp and Re 1993; Keller 1987; Krifka 1993))

The  $\lambda$ -calculus allows us to pick out the common parts of a range of propositions: what do '*John loves Mary*', '*Peter loved Mary*', '*Arthur loves Mary*' have in common? They say that someone loves Mary, or that someone **satisfies the property of loving Mary**.

So  $\lambda X(X \text{ loves Mary})$  is a property. Some things satisfy it, some don't.  $\lambda X(\text{Mary loves } X)$  is the property of being loved by Mary.

And so on and so on.  $\lambda$ -abstraction lets us define abstractions, properties, partial-states-of-affairs, situation types, ...

There's a very tight relation between a property and the set of things that satisfy it, and people sometimes get sloppy and say they're the same thing. OK so long as you remember that you're being sloppy.

Or is it?

$\text{lambda}(X, \text{rational}(X) \ \& \ \text{biped}(X)) = \text{lambda}(Y, \text{human}(Y))?$

$\text{lambda}(X, \text{unicorn}(X)) = \text{lambda}(Y, \text{gryphon}(Y))?$

$\text{lambda}(X, \text{prime}(X) \ \& \ \text{even}(X)) = \text{lambda}(Y, Y=2)?$

# Quantification and reference

---

Utterances introduce things into the context; they make claims about all or most of the things that you know about; and they refer to things.

(37) A man died.

(38) All professors are clever.

(39) The old man lives in London.

If you can't cope with this, you're not going to get anywhere: what things do we know of, what things are we talking about?

Ordinary logic is OK for dealing with (37), (38).

(37) A man died.

`exists(X, man(X)`  
    `& exists(Y, event(Y, die)`  
        `& exp(Y, X)))`

or (restricted quantifier version)

`exists(X :: {man(X)},`  
    `exists(Y :: {event(Y, die)}, exp(Y, X)))`

or

$\exists X (man(X) \wedge \exists Y (event(Y, die) \wedge exp(Y, X)))$

(38) All professors are clever.

`forall(X :: {professor(X)}, clever(X))`

Examples from catgramm.



```
| ?- chart([john, loves, mary], P), getMeaning(P, M).
```

Meaning as codes:

```
(tverb(love):name(mary) : name(john))
```

Decoded:

```
((lambda(A,  
          lambda(B,  
                (B  
                  : lambda(C,  
                        (A  
                          : lambda(D,  
                                exists(E, event(love, E, C, D))))))))  
  : lambda(F, (F : mary)))  
 : lambda(G, (G : john)))
```

```

((lambda(A,
      lambda(B,
        (B
          : lambda(C,
            (A
              : lambda(D,
                exists(E, event(love, E, C, D))))))))
  : lambda(F,F:mary))
: lambda(G, (G : john)))

```

```

(lambda(B,
  (B
    : lambda(C,
      (lambda(F, (F : mary))
        : lambda(D, exists(E, event(love, E, C, D)))))))
  : lambda(G,G:john))

```

```

(lambda(G, (G : john))
  : lambda(C,lambda(F,F:mary):lambda(D,exists(E,event(love,E,C,D)))))

```

```
(lambda(C,  
      (lambda(F, (F : mary))  
        : lambda(D,  
          exists(E,  
                event(love, E, C, D))))))  
: john)
```

```
(lambda(F, (F : mary))  
: lambda(D,exists(E,event(love,E,john,D))))
```

```
(lambda(D,  
      exists(E, event(love, E, john, D)))  
: mary)
```

```
exists(E, event(love, E, john, mary))
```

Reduced:

```
exists(E, event(love, E, john, mary))
```

```
| ?- chart([every, man, loves, mary], P), getMeaning(P, M).
```

Meaning as codes:

```
(tverb(love):name(mary) : forall:noun(man))
```

Decoded:

```
((lambda(A,  
      lambda(B,  
        (B  
          : lambda(C,  
            (A  
              : lambda(D,  
                exists(E, event(love, E, C, D)))))))))  
 : lambda(F, (F : mary)))  
: (lambda(G,  
    lambda(H,  
      forall((I :: {G:I}), (H : I))))  
 : lambda(J, man(J))))
```

```

((lambda(A,
      lambda(B,
        (B
          : lambda(C,
            (A
              : lambda(D,
                exists(E, event(love, E, C, D))))))))
: lambda(F,F:mary))
: (lambda(G,
      lambda(H,
        forall((I :: {G:I}), (H : I))))
: lambda(J, man(J)))

```

```

(lambda(B,
  (B
    : lambda(C,
      (lambda(F, (F : mary))
        : lambda(D, exists(E, event(love, E, C, D)))))))
: lambda(G,lambda(H,forall(I::{G:I},H:I)):lambda(J,man(J)))

```

```

((lambda(G,
      lambda(H,
        forall((I :: {G:I}),
          (H : I))))
  : lambda(J,man(J)))
: lambda(C,
  (lambda(F, (F : mary))
    : lambda(D, exists(E, event(love, E, C, D))))))

```

```

(lambda(H,
  forall((I :: {lambda(J,man(J)):I}),
    (H : I)))
: lambda(C,lambda(F,F:mary):lambda(D,exists(E,event(love,E,C,D)))))

```

```
forall((I
      :: {(lambda(J,
                    man(J))
           : I)}),
      (lambda(C,
              (lambda(F, (F : mary))
                : lambda(D, exists(E, event(love, E, C, D))))))
      : I))
```

```
forall((I :: {man(I)}),
      (lambda(C,
              (lambda(F, (F : mary))
                : lambda(D,
                        exists(E,
                              event(love,
                                    E,
                                    C,
                                    D))))))
      : I))
```

```
forall((I :: {man(I)}),  
      (lambda(F,  
              (F : mary))  
        : lambda(D,exists(E,event(love,E,I,D))))))
```

```
forall((I :: {man(I)}),  
      (lambda(D,  
              exists(E, event(love, E, I, D)))  
        : mary)))
```

```
forall((I :: {man(I)}),  
      exists(E, event(love, E, I, mary)))
```

Reduced:

```
forall((I :: {man(I)}),  
      exists(E, event(love, E, I, mary)))
```



```
| ?- chart([every, man, loves, some, woman], P), getMeaning(P, M).
```

Meaning as codes:

```
((tverb(love) : some:noun(woman))  
 : (forall : noun(man)))
```

Decoded:

```
((lambda(A,  
  lambda(B,  
    (B  
      : lambda(C,  
        (A  
          : lambda(D,  
            exists(E, event(love, E, C, D)))))))))  
 : (lambda(F,  
   lambda(G,  
     exists((H :: {F:H}), (G : H))))  
 : lambda(I, woman(I))))  
 : (lambda(J,  
   lambda(K,  
     forall((L :: {J:L}), (K : L))))  
 : lambda(M, man(M))))
```

```

((lambda(A,
      lambda(B,
        (B
          : lambda(C,
            (A
              : lambda(D,
                exists(E, event(love, E, C, D))))))))
: lambda(F,lambda(G,exists(H:: {F:H},G:H)):lambda(I,woman(I)))
: (lambda(J,
      lambda(K,
        forall((L :: {J:L}), (K : L))))
: lambda(M, man(M)))

```

```

(lambda(B,
  (B
    : lambda(C,
      ((lambda(F,
        lambda(G,
          exists((H :: {F:H}),
                (G : H))))
        : lambda(I, woman(I)))
      : lambda(D, exists(E, event(love, E, C, D)))))))
: lambda(J,lambda(K,forall(L::{J:L},K:L)):lambda(M,man(M)))

```

```

((lambda(J,
      lambda(K,
        forall((L :: {J:L}),
              (K : L))))
  : lambda(M,man(M)))
: lambda(C,
  ((lambda(F,
    lambda(G,
      exists((H :: {F:H}),
            (G : H))))
    : lambda(I, woman(I)))
  : lambda(D, exists(E, event(love, E, C, D))))))

```

```

(lambda(K,
  forall((L :: {lambda(M,man(M)):L}),
    (K : L)))
  : lambda(C,lambda(F,lambda(G,exists(H::{F:H},G:H))):lambda(I,woman(I)):lambda(D,

```

```
forall((L
      :: {(lambda(M,
                    man(M))
              : L)}),
(lambda(C,
      ((lambda(F,
            lambda(G,
                  exists((H :: {F:H}),
                        (G : H))))
              : lambda(I, woman(I)))
      : lambda(D, exists(E, event(love, E, C, D))))))
: L))
```

```

forall((L :: {man(L)}),
  (lambda(C,
    ((lambda(F,
      lambda(G,
        exists((H :: {F:H}),
          (G : H))))
      : lambda(I, woman(I)))
    : lambda(D,
      exists(E,
        event(love,
          E,
            C,
            D))))))
  : L))

```

```
forall((L :: {man(L)}),
      ((lambda(F,
              lambda(G,
                    exists((H
                          :: {F:H}),
                          (G : H))))
       : lambda(I,woman(I)))
      : lambda(D, exists(E, event(love, E, L, D))))))
```

```
forall((L :: {man(L)}),
      (lambda(G,
              exists((H :: {lambda(I,woman(I)):H}),
                    (G : H)))
       : lambda(D,exists(E,event(love,E,L,D))))))
```

```
forall((L :: {man(L)}),
  exists((H
    :: {(lambda(I,
      woman(I))
      : H)}),
    (lambda(D, exists(E, event(love, E, L, D)))
      : H)))
```

```
forall((L :: {man(L)}),
  exists((H
    :: {woman(H)}),
    (lambda(D,
      exists(E,
        event(love, E, L, D)))
      : H)))
```

```
forall((L :: {man(L)}),
  exists((H :: {woman(H)}),
    exists(E, event(love, E, L, H))))
```



Reduced:

```
forall((L :: {man(L)}),  
      exists((H :: {woman(H)}),  
            exists(E, event(love, E, L, H))))
```

Doing reductions is entirely mechanical. Tedious, easy to make mistakes, but completely mechanical. Requires no thought.

Working out what the meanings of the constituents should be is creative.

- Compositionality: what is it, why do I want it, why can't I have it, why the  $\lambda$ -calculus may help
- $\lambda$ -calculus:
  - as a notion: abstracting the common elements of a set of propositions, defining a set
  - as a mechanism:  $\beta$ -reduction (why  $\beta$ : no idea)
- Montague's treatment of determiners. This is fairly hard (!), but **crucial**

The formal paraphrases above preserve the logical organisation of the input sentence.

But we also need to know what the terms that appear in them **mean**.

!!!! HELP !!!!

Is there some basic set of concepts that you can reduce everything else to? (Schank 1972; Jackendoff 1983).

Would it do you any good if there were?

Can you define words in terms of sets of attributes? (Katz and Fodor 1963)

Can you define words in terms of each other?

Garbage collect a dictionary:

1. Collect all the words that appear in the right hand side of some definition
2. Go through the dictionary and delete anything that wasn't in this collection
3. If anything got deleted, go to (1)

More than half the dictionary will remain. Which suggests to me that there is no core set (only suggests: it might be possible but awkward to do all the definitions in terms of some core set).

A fairly feeble, but probably tenable, position: you can specify relationships between words so long as you use a default logic (Cruse 1986).

Let's see some cases.

## Events & thematic roles

---

In the examples given earlier, some of the terms came directly from words in the text, so we can be fairly sure that nothing has been introduced or lost by using them.

But some of them didn't: in particular, roles like agent, object, recipient are not manifest in the text itself.

Presumably they mean something: otherwise why introduce them?

But if they do mean something, we'd better get them right.



Why should we care?

### **‘Transformations’**

(40) He ate it/it was eaten.

(41) I gave my mother a book/I gave a book to my mother.

Zeugma (Fillmore 1968)

(42) John is cooking

(43) Mary is cooking.

(44) Dinner is cooking.

(45) John and Mary are cooking.

(46) John and dinner are cooking.

They seem to carry independent information

(47) The mome raths outgrabe.

## Thematic roles: weak version

---

The transitive verb 'eat' involves two entities, an eater and an eatee; the transitive verb 'kill' involves two entities, a killer and a killee; the intransitive verb 'sleep' involves one entity, a sleeper; ...

### Advantages

- Can't be wrong.
- Supports the active/passive contrast, gives you a handle on zeugma.

### Disadvantages

- Doesn't tell you anything.

See (**Pollard and Sag 1988**).

Lexical items look like

```
word(fancies, X) :-  
    weakThematicRoles,    % Flag to see which version we want to use  
    verb(X, transitive(A1, A2)),  
    glue@A1 -- theta(fancier),  
    glue@A2 -- theta(fanciee),  
    content@X -- lambda(E, fancy(E)).
```

```
word(love, X) :-  
    weakThematicRoles,  
    verb(X, transitive(_A1, _A2)),  
    glue@A1 -- theta(lover),  
    glue@A2 -- theta(lovee),  
    content@X -- lambda(E, love(E)).
```

```
word(likes, X) :-  
    weakThematicRoles,  
    verb(X, transitive(A1, A2)),  
    glue@A1 -- theta(lover),  
    glue@A2 -- theta(lovee),  
    content@X -- lambda(E, love(E)).
```

```
?- chart([john, loves, mary, '.'], P), showMeaning(P).
```

```
claim(exists(A,  
            like(A)&theta(A,likee,mary)  
            &theta(A, liker, john)))
```

```
| ?- chart([john, loves, mary, '.'], P), showMeaning(P).
```

```
claim(exists(A,  
            love(A)&theta(A,lovee,mary)  
            &theta(A, lover, john)))
```

love is a special case of like: fair enough. But then I also have to say that lover is a special case of liker, ...

## Thematic roles: strong version

---

There's a finite set of thematic roles. Each of these has a surface marker, though in some cases this marker is omitted.

In English, for instance, there is a set of prepositions which appear to mark manifestations of thematic roles in specified surface forms:

(48) I gave my mother a book.

(49) I gave a book to my mother.

*'my mother'* is fulfilling the role denoted by *'to'* in each of these sentences – she is the *to* of (48) and (49). You can call this role the *recipient* if you want: it doesn't actually buy you anything, but it doesn't cost much either.



(50) She killed him.

(no obvious marker of either role)

(51) He was killed by her.

(the person who did it is marked by '*by*')  
 \_\_\_\_\_

(52) The Romans destroyed the city.

(53) The city was destroyed by Romans.

(54) The city's destruction by the Romans . . .

(55) The Romans' destruction of the city . . .

(the thing it was done to is marked by 'of')

(56) I saw him fall.

(57) I watched him fall.

(58) I heard a bird singing.

(59) I listened to a bird singing.

## Advantages

- You can attach quite a lot of semantic information to the role *per se*, rather than repeating it for every verb with a *to*.
- Relates syntax and semantics nicely.

## Disadvantages

- Doesn't work.

See (Dowty 1989) (but not necessarily other work by Dowty: see (Dowty 1991) for a detailed, and inconclusive, investigation).

Once you've settled on your set of roles, how would you attach any significance to them?

By saying what they entail. What else could there be?

```
forall(E,  
  forall(X :: {agent(E, X)},  
    animate(X) & intended(X, E)))
```

```
forall(E,  
  forall(X :: {instrument(E, X)},  
    forall(Y :: {agent(E, Y)}, used(Y, X, E))))
```

...

How do you know what roles a particular verb takes? What do you do if the involvement of a given player depends on what it is?

```
forall(E,  
  forall(X :: {agent(E, X) & animate(X)},  
    intended(X, E)))
```

```
forall(E,  
  forall(X :: {agent(E, X)}, caused(X, E)))
```

And what do animate, caused, intended, ... mean?

- Thematic roles say something about the way that something or someone is involved in an event
- They're not clearly marked in the surface form, so there's a lot of room for debate about how many there are and which verbs take which ones. There is some connection between prepositions (in English: case-marking in other languages) and thematic roles, but it's not transparent.
- Once you've chosen them you have to axiomatise them. Which is pointless unless you're then going to reason with your axioms. HOLD THAT THOUGHT.

```
| ?- chart([the, old, man, bought, a, nice, book, '.'], P), showMeaning(P),  
claim(exists(A,  
    buy(A)  
    &exists(B::{book(B)&nice(B)},theta(A, object, B))  
    &theta(A,  
        agent,  
        ref(lambda(C, man(C)&old(C))))))
```

```
| ?- chart([the, nice, man, bought, an, old, book, '.'], P), showMean  
claim(exists(A,  
    buy(A)  
    &exists(B::{book(B)&old(B)}, theta(A, object, B))  
    &theta(A,  
        agent,  
        ref(lambda(C, man(C)&nice(C))))))
```



What do '*nice*' and '*old*' contribute here?

- '*old man*' – man in his seventies  
'*old book*' – book printed in the 19th century
- '*nice man*' – man who is kind to children, doesn't tell lies, looks after his infirm elderly relatives  
'*nice book*' – book with a happy ending, one which makes you feel happy when you read it

Can we produce meaning postulates which are flexible enough to cope with this?

```
forall(X :: {old(X)},  
      man(X) -> age(X) > 70  
      & book(X) -> age(X) > 90  
      & house(X) -> age(X) > 60  
      & star(X) -> age(X) > 6000000000  
      & ...).
```

```
forall(X :: {nice(X)},  
      man(X) -> kind_to_children(X) & ...  
      & book(X) -> happy_ending(X) & ...  
      & ...)
```

Hard work, but if you want to get any mileage out of saying that something is a nice book you will have to say what nice books are like.

But it somehow misses the point: I can coherently talk about an old star without actually knowing anything about how old a typical star is.

I'd like something more like

```
forall(X :: {old(X)},  
      most(Y :: {type(X)=type(Y)},  
           age(X) > age(Y)))
```

I've introduced a default here: that doesn't scare me – that was always going to have to happen.

More worrying is the introduction of X's type. The problem is that the type is not an intrinsic property of X; it's part of the way it was described.

(60) – That's a very small dog.

– But it's quite a large chihuahua.

It gets worse.

(61) He was waving an imitation gun.

(62) They sold him a fake Picasso.

(63) JFK's alleged murderer was himself assassinated.

I don't think that what he was waving was a gun; I'm sure that what they sold him wasn't a Picasso; the speaker doubts that the person in question assassinated JFK.

So we can't have an analysis of (61) like

```
| ?- chart([he, was, waving, an, imitation, gun, '.'], P), showMeaning  
claim(exists(A,  
    wave(A)  
    &exists(B::{gun(B)&imitation(B)},  
        theta(A, object, B))  
    &theta(A,  
        agent,  
        ref(lambda(C, male(C)&salient(C))))))
```

It's not something which is a gun and is an imitation: it's something which is an imitation of a gun – something which lacks some essential property of guns.

```
| ?- chart([he, was, waving, an, imitation, gun, '.'], P), showMeaning(P).
```

```
claim(exists(A,  
    wave(A)  
    &exists(B  
        ::{imitation(B,lambda(C,gun(C)))},  
        theta(A, object, B))  
    &theta(A,  
        agent,  
        ref(lambda(D, male(D)&salient(D))))))
```

The thing that he was waving was something which was a fake when considered as a gun.



## 1. Non-factive adjectives:

```
forall(X,  
  forall(P :: {fake(X, P)},  
    not(P:X)  
    & artefact(X)  
    & intended(creator(X), like(X, P))))
```

## 2. Non-intersective adjectives:

```
forall(X,  
  forall(P :: {old(X, P)},  
    (P.X)  
    & most(Y :: {P:Y},  
      age(X) > age(Y))))))
```

### 3. Intersective adjectives (fairly rare):

```
forall(X,  
  forall(P :: {stolen(X, P)},  
    (P.X)  
    & exists(Y, stole(Y, X))))
```

Hard work. Involves a move to intensional logic, with all its potential for complexity.

But worth it: if I said to you '*That's a fake Picasso*' I would be shocked if you spend £25000000 on it.

- Adjectives generally give you a relationship between an entity and a property.
- Some adjectives comment on whether the property holds of the item. Trying to axiomatise things like this requires you to use a rather expressive logic.
- Once you've thought about what adjectives mean intuitively, you have to axiomatise your intuitions. Which is pointless unless you're then going to reason with your axioms.

I've started writing down relationships between terms. These are **not** definitions:

- they are not necessary and sufficient conditions for using the term: some are necessary, some are sufficient, some are neither (most are neither!).
- what we have here is more like an encyclopædia than a dictionary. These things are explanations, not definitions.

I may need some very fancy logic for expressing the meaning of a word like '*want*'. That doesn't mean that everything I want needs a very fancy logic.

We often need to assign entities to '**natural kinds**': you can deal with natural kinds (and other predefined hierarchies and type lattices) very efficiently, and we should take advantage of this when we can.

We now have two levels of analysis:

- the formal paraphrase, obtained (so far) compositionally (and hence almost for free) when we have parsed the sentence.
- the meaning postulates, which describe the relationships between concepts. These will be exploited when we want to know what something entails. How much this costs will depend on how much you want to know.



This has some nice consequences in terms of modularity: the construction of a formal paraphrase is separated from reasoning about its consequences. So you can use different sets of meaning postulates, or different inference engines, with the same '**linguistic meaning**'.

What can we do now that we've got an inference engine?

Take part in dialogues. Answer questions.

But dialogues are extended, and involve talking about things that we've already talked about.

Ordinary logic cannot cope with the '**situated nature of natural language**'.

(64) The man died.

(65) She had hated him.

(66) The president of the USA is the most powerful person on earth.

(67) John believes he will marry the richest debutante in Dubuque.

Intuitively it's quite easy to see what's going on? Dealing with it formally (and computationally) is tricky.

First attempt was by Russell. You can use a '**referring expression**' (name, pronoun, definite NP, various other things) if there is exactly one thing that satisfies the description.

Introduce a new quantifier into our logic: there is exactly one such item.

(68) a. A man slept.

```
exists(A :: {man(A)},  
      exists(B, event(sleep,B) & agent(B,A)))
```

b. The man slept.

```
exists1(A :: {man(A)},  
       exists(B, event(sleep,B) & agent(B,A)))
```

<http://plato.stanford.edu/entries/descriptions/#RusTheDes>

## Interaction with *'not'*

---

(69) | ?- chart([john, did, not, see, a, cat], P), decode( semantics@P

```
    not(exists(A :: {cat(A)},
           exists(B,
                  event(see, B)
                  & agent(B,john) & object(B,A))))
```

(70) | ?- chart([a, dog, did, not, chase, a, cat], P), decode( semanti

```
    not(exists(A :: {B:A},
           exists(C :: {cat(C)},
                  exists(D,
                         event(chase, D)
                         & agent(D,A) & object(D,C))))))
```

```
(71) | ?- chart([john, saw, a, cat, or, john, did, not, see, a, cat],  
exists(A :: {cat(A)},  
    exists(B,  
        event(see, B)  
        & agent(B,john) & object(B,A)))  
or not(exists(C :: {cat(C)},  
    exists(D,  
        event(see, D)  
        & agent(D,john) & object(D,C))))
```

(72) | ?- chart([john, saw, the, cat, or, john, did, not, see, the, c

```
exists1(A :: {cat(A)},
        exists(B,
                event(see, B)
                    & agent(B,john) & object(B,A)))
or not(exists1(C :: {cat(C)},
          exists(D,
                  event(see, D)
                      & agent(D,john) & object(D,C))))
```

Maybe this is just a scope problem (like '*Someone is mugged in New York every five minutes*', '*I am looking for a unicorn*'), so if we could get the scope right it would be OK.

```
(73) exists1(A :: {cat(A)},  
      exists(B,  
            event(see, B)  
            & agent(B, john) & object(B, A)))  
or exists1(C :: {cat(C)},  
          not(exists(D,  
                  event(see, D)  
                  & agent(D, john) & object(D, C))))
```

We're missing the fact that it's the **same** cat in both case:  
but worse ...



- (74) a. – The Prince of Wales is bald.  
– The Prince of Wales is not bald.  
b. – The King of France is bald.  
– The King of France is not bald.

The context in which the item is unique has to be local: almost no referring expressions are unique with respect to the entire universe, they're unique with respect to the situation.

(75) A man and a woman came into the room. **The man** had a hat.

You wouldn't want to say that there is only one man in the world. There is only one man **in this conversation**

Numerous solutions: situation semantics (Barwise and Perry 1983), dynamic predicate logic (Groenendijk and Stokhof 1991), discourse representation theory (Kamp 1984; Kamp and Reyle 1993). Two basic moves + a lot of detail.

- Introduce something which looks just like a referring expression into your formal paraphrase.
- Treat it as a name whose denotation can only be fixed by inspecting the current discourse state (whatever **that** is). Do that **before** you add it to your belief set.

Plenty of problems:

- very few descriptions are actually **uniquely** identifying: '*the man*'
- there may not be anything suitable in the context **at the moment**: '*the first snowdrops of Spring*'
- One or other of us may fail to believe in the existence of the referred to item:

(76) A: My husband bought me a Ferrari for Christmas.

B: I don't believe you.

A: He did, he really did.

B: Go on then, what colour is it?

- Funny things happen when reference occurs inside belief contexts (and maybe other intensional contexts as well).

(77) Hob believes that a witch killed his pig, and Nob believes that she killed his cow.

- Funny things happen when reference occurs inside logical operators.

(78) Each man kills the thing he loves.

(79) If a farmer owns a donkey he beats it.

(80) If a nurse works hard he will be promoted.

- It may be new to you.

(81) My eldest son's train set is in bits all over his bedroom floor.

You may not have known I had an eldest son. You certainly didn't know he had a train set. But it's not that surprising, so you '**accommodate**' it.

- Referential success is a good cue for disambiguation (Hirst 1987)

(82) I saw the man in the park with a telescope.

How many men are there? How many parks? Is there a park with a telescope?

If there are no parks with a telescope then any interpretation that attaches '*with a telescope*' to '*park*' will have to accommodate the existence of such a thing.

If there are two parks, one of which has a telescope, then any interpretation that doesn't attach '*with a telescope*' to '*park*' will have an ambiguity.

and likewise for '*man*' and '*in the park*'.

- The ‘**slate**’ (‘**common ground**’, ‘**minutes**’): what someone who has been in this conversation should be committed to.
- My (your) view of the slate contains objects I (you) know about and relations between them.
- My view of it is not visible to you: in particular, the objects it contains are not.



- I can tell you about the **existence** of objects, and I can make **universal** statements.
- I can tell you about the existence of proofs!
- Referential terms: claims that I have certain kinds of proof

$\Delta[S], \Delta[H]$  are the Speaker and Hearer's view of what the common ground **ought** to be like.

$$\vdash \Delta[S] \rightarrow \exists X(\text{man}(X) \\ \wedge \forall Y((\Delta[S] \rightarrow \text{man}(Y)) \\ \rightarrow Y = X))$$

In practice:

- Check that  $\Delta[H] \vdash \exists X(\text{man}(X))$
- Replace the term by a new constant  $s$  and note that  $\forall Y((\Delta[S] \rightarrow \text{man}(Y)) \rightarrow Y = s)$

The examples with misleading gender markers suggest that we actually fill in some stuff that isn't explicitly present in the utterance as soon as we hear it.

The examples with misleading gender markers suggest that we actually fill in some stuff that isn't explicitly present in the utterance as soon as we hear it.

(83) If a farmer owns a donkey she will look after it nicely.

It also seems likely that we do a sanity check as soon as we hear someone say something:

It also seems likely that we do a sanity check as soon as we hear someone say something:

(84) My father was born at the bottom of the sea.

So it looks as though we do a bit of reasoning forwards, at least to create a basic model. Don't know how much to do. Just do enough to be fairly confident you've got the general gist? Enough to anchor the referring expressions? Enough to fix the defaults? Everything you can without recursion? As much as you can be bothered to do?

```

| ?- chart([a, farmer, owns, a, donkey, '.'], P), update(P, Q).

claim(exists(A,
    own(A)
    & exists(B :: {donkey(B)},
        theta(A, object, B))
    & exists(C :: {farmer(C)},
        theta(A, agent, C))))

% Skolemised, anchored and Prolog-ised
newFact(own(#(6)))
, newFact(donkey(#(7)))
, newFact(theta(#(6), object, #(7)))
, newFact(farmer(#(8)))
, newFact(theta(#(6), agent, #(8)))

```



(Use `newFact` rather than just `assert` to store a record of what we have asserted so we can clean up the database for a new dialogue)

The 'model' that this produces:

```
| ?- listing(temp).  
temp(own(#(6))).  
temp(donkey(#(7))).  
temp(theta(#(6),object,#(7))).  
temp(farmer(#(8))).  
temp(theta(#(6),agent,#(8))).
```

```
| ?- chart([he, beats, it, '.'], P), update(P, Q).
```

```
claim(exists(A,  
    beat(A)  
    & theta(A,  
        object,  
        ref(lambda(B, neuter(B) , salient(B))))  
    & theta(A,  
        agent,  
        ref(lambda(C, male(C) , salient(C))))))  
  
% anchored  
newFact(beat(#(9)))  
    , newFact(theta(#(9), object, #(7)))  
    , newFact(theta(#(9), agent, #(8)))
```

```
| ?- listing(temp).  
temp(own(#(6))).  
temp(donkey(#(7))).  
temp(theta(#(6),object,#(7))).  
temp(farmer(#(8))).  
temp(theta(#(6),agent,#(8))).  
temp(beat(#(9))).  
temp(theta(#(9),object,#(7))).  
temp(theta(#(9),agent,#(8))).
```

Anchoring involves reasoning **using background knowledge**

```
woman(X) :- named(X, susan). % should be female(X) :- named(X, susan)
man(X) :- named(X, john).
```

```
man(X) :- farmer(X). % should be a default rule
male(X) :- man(X).
```

```
neuter(X) :- donkey(X). % but donkeys aren't neuter!
animal(X) :- donkey(X).
```

```
salient(_X). % Don't have a proper theory of salience yet
```

What happens if there isn't anything that fits the description?

Just pretend there was?

```
| ?- chart([he, hates, susan, '.'], P), update(P, Q).
```

```
claim(exists(A,  
            hate(A  
              & theta(A,  
                    object,  
                      ref(lambda(B, named(B, susan))))  
              & theta(A,  
                    agent,  
                      ref(lambda(C, male(C) , salient(C))))))
```

```
newFact(hate(#(10)))  
  , newFact(theta(#(10), object, accomodated(#(11), named(#(11), susan))))  
  , newFact(theta(#(10), agent, #(8)))
```

I've flagged the fact that we'd never heard of Susan before by instantiating the new fact with `accomodated(#(9), named(#(9), susan))`. Is this the right thing to do? Should I just introduce her as `#(9)` and record the fact this is an accommodation separately?

I certainly should record the fact that she is named Susan: `named(#(9), susan)`.

```
| ?- listing(temp).  
temp(own(#(6))).  
temp(donkey(#(7))).  
temp(theta(#(6),object,#(7))).  
temp(farmer(#(8))).  
temp(theta(#(6),agent,#(8))).  
temp(beat(#(9))).  
temp(theta(#(9),object,#(7))).  
temp(theta(#(9),agent,#(8))).  
temp(named(#(11),susan)).  
temp(hate(#(10))).  
temp(theta(#(10),object,accomodated(#(11),named(#(11),susan)))).  
temp(theta(#(10),agent,#(8))).
```



# CHECKPOINT

---

- Successful reference depends on both parties having identical versions of the minutes. There is no way of checking this (except by referential failure)! The more general the description in a referential term, the more effective it as a check that the minutes are in step.
- Reference is about **proofs**. The uniqueness element of reference is about there being nothing else which can be **proved** to satisfy the description from mutually available information.
- Referring expressions are supposed to pick out known entities. Sometimes they are used in situations where it is clear that the entities in question are not known; this **accommodation** marks them as being not very interesting.

## Index

accommodate, 454  
accusative case, 239  
add, 168  
ambiguity, 235  
arguments, 108  
attachment sites, 235  
backtracking stack, 29  
choice stack, 29  
closed world assumption, 166  
common ground, 456  
complete, 115  
Conjunction, 110  
constants, 106  
constructive implication, 138  
consulting, 24  
context-free, 223, 282  
cut, 60  
database, 68  
delete, 168  
Disjunction, 110  
Existential quantification, 110  
facts, 100  
features, 252  
frame problem, 178  
goal stack, 28  
head, 37  
holds of, 108

Horn-clause logic, 135

hypothetical, 209

Implication, 110

inference, 102

interpretation, 119

label, 150

Lexical ambiguity, 366

linguistic meaning, 441

logic, 358

Long-distance dependencies, 277

marked orders, 282

minutes, 456

model, 119

model theory, 114

name, 167

natural kinds, 439

Negation, 110

negation as failure, 75, 137

negation-as-failure, 178

non-canonical orders, 282

non-monotonic, 75

object case, 239

out of order items, 282

preconditions, 167

predicate names, 22

predicates, 107

proof theory, 114

ramification problem, 208

referring expression, 444

rules, 101

satisfies, 108

satisfy, 108

Scope ambiguity, 367  
singleton variables, 46  
situated nature of natural language, 443  
Skolem constants, 130  
Skolem functions, 130  
slate, 456  
sound, 115  
Structural ambiguity, 367  
subject, 239  
subject case, 239  
tail, 37  
Tarski biconditional, 374  
Transformations, 409  
unifiable, 34  
unification, 34, 255  
Universal quantification, 110  
WH-marked, 280

## References

- Barwise, J., Perry, J., 1983. *Situations and Attitudes*. Bradford Books, Cambridge, MA. [451](#)
- Cruse, D. A., 1986. *Lexical Semantics*. Cambridge University Press, Cambridge. [407](#)
- Dowty, D. R., 1989. On the semantic content of the notion of 'thematic role'. In: Chierchia, G., Partee, B. H., Turner, R. (Eds.), *Properties, Types and Meaning II: Semantic Issues*. Kluwer Academic Press, Dordrecht, pp. 69–130. [419](#)
- Dowty, D. R., 1991. Thematic proto-roles and argument selection. *Language* 67, 547–619. [419](#)
- Dowty, D. R., Wall, R. E., Peters, S., 1981. *Introduction to Montague Semantics*. D Reidel, Dordrecht. [364](#), [378](#)
- Field, D. G., Ramsay, A. M., 2004. How to build towers of arbitrary heights. In: *The 23rd Annual Workshop of the UK Planning and Scheduling Special Interest Group (PlanSIG 2004)*. University College Cork, p. 11 pages. [208](#)

- Fikes, R. E., Nilsson, N. J., 1971. Strips: a new approach to the application of theorem proving to problem solving. *Artificial Intelligence* 3(4), 251–288. [160](#)
- Fillmore, C., 1968. The case for case. In: Bach, E., Harms, R. (Eds.), *Universals in Linguistic Theory*. Holt, Rinehart and Winston, Chicago, pp. 1–90. [410](#)
- Gabbay, D. M., 1996. *Labelled Deductive Systems*. Oxford University Press, Oxford. [150](#)
- Gazdar, G., Klein, E., Pullum, G. K., Sag, I., 1985. *Generalised Phrase Structure Grammar*. Basil Blackwell, Oxford. [378](#)
- Groenendijk, J., Stokhof, M., 1984. Studies on the semantics of questions and the pragmatics of answers. Ph.D. thesis, Department of Philosophy, University of Amsterdam. [378](#)
- Groenendijk, J., Stokhof, M., 1991. Dynamic predicate logic. *Linguistics and Philosophy* 14, 39–100. [451](#)
- Hirst, G., 1987. Semantic interpretation and the resolution of ambiguity. *Studies in natural language processing*. Cambridge University Press, Cambridge. [455](#)

- Jackendoff, R. S., 1983. *Semantics and Cognition*. MIT Press, Cambridge, MA. [405](#)
- Kamp, H., 1984. A theory of truth and semantic representation. In: Groenendijk, J. A. G., Janssen, T. M. V., Stokhof, M. B. J. (Eds.), *Formal Methods in the Study of Language*. Foris Publications, Dordrecht, pp. 277–322. [451](#)
- Kamp, H., Reyle, U., 1993. *From discourse to logic: introduction to model theoretic semantics of natural language*. Kluwer Academic Press, Dordrecht. [378](#), [451](#)
- Katz, J. J., Fodor, J. A., 1963. The structure of a semantic theory. *Language* 39, 170–210. [405](#)
- Keller, W. R., 1987. Nested Cooper storage: the proper treatment of quantification in ordinary noun phrases. Tech. Rep. CSRP. 73, University of Sussex. [378](#)
- Krifka, M., 1993. Focus, presupposition and dynamic interpretation. *Journal of Semantics* 10. [378](#)
- Manthey, R., Bry, F., 1988. Satchmo: a theorem prover in Prolog. In: Lusk, R., Overbeek, R. (Eds.), *Proceedings of the 9th International Con-*

- ference on Automated Deduction (CADE-9). Vol. 310 of Lecture Notes in Artificial Intelligence. Springer-Verlag, Berlin, pp. 415–434. [141](#)
- Pollard, C. J., Sag, I. A., 1988. An Information Based Approach to Syntax and Semantics: Vol 1 Fundamentals. CSLI lecture notes 13, Chicago University Press, Chicago. [411](#)
- Reiter, R., 1980. A logic for default reasoning. Artificial Intelligence 13(1), 81–132. [153](#)
- Sacerdoti, E. D., 1974. Planning in a hierarchy of abstraction spaces. Artificial Intelligence 5(2). [195](#)
- Schank, R. C., 1972. Conceptual dependency: a theory of natural language understanding. Cognitive Psychology 3(4), 552–631. [405](#)
- van Eijck, J., Alshawi, H., 1992. Logical forms. In: Alshawi, H. (Ed.), The Core Language Engine. Bradford Books/MIT Press, Cambridge, Mass., pp. 11–40. [378](#)
- van Genabith, J., Crouch, R., 1997. How to glue a donkey to an f-structure. In: Bunt, H. C., Kievit, L., Muskens, R., Verlinden, M. (Eds.), 2nd International Workshop on Computational Semantics. University of Tilburg, pp. 52–65. [378](#)