COMP24412 - Symbolic AI

November 26, 2017

1 Course Introduction

Aristotle attempted to catalogue all the valid argument forms as *syllogisms*. In the 19th century, De Morgan, Boole and Jevons attempted to broaden the scope of this logic. The most decisive advance was made at the end of the 19th century by Frege and Bertrand Russell: they developed *formal languages* to represent information. In the 1980s , serious attempts were made to use logical methods in AI.

- The agent's knowledge is represented as a collection formulas in some logic.
- Reasoning tasks (planning, deduction, hypothesis formation, learning) is realised by the manipulation of these formulas.

2 Prolog

2.1 Prolog I

The `?-` is the Prolog interpreter's prompt.
The `[<program>]` means "reconsult". Once the program has been "consulted", it can be queried.

Prolog is a form of **declarative programming**: the program is a list of facts and it is run by typing goals. The meaning of the program is separated from the goals that are set. Goals may contain variables, which are signalled by capital letters.

The semicolon is typed by the user to find further solutions to a given goal.
You can have more than one variable in a query, but shared variables have to take unitary values.

Unification is when Prolog has a goal, it tries to match that goal against something in the program. The *matching process* involves finding values for any variables so that the goal and the fact with which it is matched become identical. Unification is a **completely deterministic process**. There are no choices available. Choices only arise when several facts and/or rules unify with a given goal. Matching of goals and facts are called *rule–heads.*

Clauses in programs can themselves contain variables. The comma is interpreted as *and* and :- means *if.*

Prolog processing:

1. When a goal is typed in, Prolog puts it on a **goal stack**.
2. Prolog then tries to find (i) a factor or (ii) a rule–head in the program which unifies with the top goal on the goal stack.

3. If (i), then variable bindings are carried through and the goal is popped off the stack.
4. If (ii), then variable bindings are carried through, the goal is popped off the stack and the clauses in the rule tail are pushed onto the top of the goal stack.
5. The process is repeated from step 2 until the goal stack is empty.

A given goal may unify with several goals or rule–heads, these are called *choice points*. If Prolog does not find a rule that unifies with the top goal on the stack, the goal is said to **fail**. Whenever Prolog encounters failure, it **backtracks** to the most recent choice point, takes the next available choice, and proceeds as before. It helps thinking of the program as implicitly defining an execution tree, which the Prolog interpreter searches. The nodes represent states of the goal stack and then links possible unifications of the top goal with some fact or rule in the program. The tree is searched branch-by-branch, so it is a **depth–first search**, or chronological backtracking.

Backtracking is the process by which all possibilities for matching facts and rules are tried.
It is also a good idea to think of programs as stating facts, then Prolog is a **theorem prover**.

A Prolog program is frequently recursive since recursion may be used on all sorts of structures (including numbers).

2.2 Prolog II

Arithmetic is performed with the usual arithmetic operators and the predicate `is`.
Arithmetic operators cannot be used backwards, like `6 + 2 is X`.
The underscored variable `_R` means we do not care about its value.
Some operators are written in *infix* form.

In a compound term such as `couple(costi, fil)`, `couple` is the **functor** while `costi, fil` are the **arguments**.

Predicates are used to make statements, while **functors** are used to refer to complex objects.

A **list** in Prolog is denoted by a sequence of its elements, separated by commas, and enclosed in brackets `[ ]`. Prolog syntax allows | to separate the head and the tail of a list. They are operated recursively, i.e. `[a,b,c,d]` is `'.'(a, '.'(b, '.'(c, '.'(d, []))))`.

Test whether an element is in a list.

```
member(X, [X|_L]).
member(X, [_Y|L]) :-
    member(X,L).
```

Number of goal calls linear in the first argument.

Append `L1` and `L2` to make `L3`.

```
append([], L, L).
append([X|L1], L2, [X|L3]) :-
    append(L1, L2, L3).
```

Get the length of a list.

```
length([], 0).
length([X|L], N) :-
    length(L, N1),
    N is N + 1.
```

Reverse a list.

```
rev1([], []).
rev1([X|L], L_ans):-
    rev1(L, L_ans1),
    append(L_ans1, [X], L_ans).
```

Suppose we are reversing a list with N elements. For the kth item in the list, we perform an append on the end of a $N-k$ element list. This append takes $N-k+1$ reductions. $(N+1)+N+\cdots+1 = \frac{1}{2}(N+1)(N+2)$

Pluralise a word.

```
pluralizer(Ws, Wp) :-
    name(Ws, WsChars),
    reverse(WsChars, WsCharsRev),
    reverse([115 | WsCharsRev], WpChars),
    name(Wp, WpChars).
```

You can use `maplist` to apply a function to each member of the list.

`=..` converts complex terms into lists, which is called *univ*.

Strings in Prolog are lists of ASCIIs.

Useful functions:

- `name/2` converts between atoms and their strings.
- `var/1` checks if it is a variable.
- `atom/1` checks if it is an atom.
- `atomic/1` check if it is an atom, including numbers.
- `compound/1` checks if it is a compound term.
- `is_list/1` checks if it is a list.
- `proper_list/1` checks it it is a list whose tail is non-empty.

It is important to test for equality without using unification (`=`) and using `==` instead.

The predefined predicate `setof` sorts the results to get a sorted list of alternatives without duplicates. You can make your own infix operator using `op`.

3 Prolog III

The goal `not(goal)` succeeds when goal fails , it sometimes means "not as far as I know". `not` is not part of logic programming and has no clear declarative meaning.

A **set** is like a list except that the order of elements is unimportant and there are no repeated elements.

Union of two sets.

```
union([], S, S).
union([X|S], S1, S2) :-
    member(X, S1),
    union(S, S1, S2).
union([X|S], S1, [X|S2]) :-
    not(member(X, S1)),
    union(S, S1, S2).
```

`X \= Y` is the same as `not(X = Y)`. For examples, `not(parent(X, Noel))` will fail. `!` will always succeeds. Once an instance of `!` has succeeded, the **flow of control** of Prolog is committed to all choices made between the matching of the clause containing that instanceof `!` and the instance of `!` itself.

Find the max between two arguments.

```
max2(X, Y, Y).
max2(X, Y, Z) :-
    X >= Y,
    !,
    X = Z.
```

`call(term)` calls term as if it were a goal in its own right.

We can define `not` in terms of `!`.

```
not(Goal) :-
    call(Goal),
    !,
    fail.
not(Goal).
```

Linear in the length of the list. `L_acc` is an **accumulator**, the program interpreter can forget about the program state on the stack before making each recursive call, which is more space efficient.

Better reverse list program.

```
rev2(L, L1) :-
    rev_acc(L, [], L1).
rev_acc([X|L], L_acc, L_ans) :-
    rev_acc(L, [X|L_acc], L_ans).
rev_acc([], L_acc, L_acc).
```

List-reverse program using difference list.

```
rev3(L, L1) :-
    rev_dfl(L, L1/[]).
rev_dfl([X|L], L_ans/L_ans_tail) :-
    rev_dfl(L, L_ans/[X|L_ans_tail]).
rev_dfl([], l_ans_total/L_and_tail).
```

3.1 Tail Recursion Optimisation

Space reclamation handled by the Prolog compiler. It is a technique consisting on economising on the use of memory during recursion. Its name refers to the position of the recursive sub-goal at the tail-end of the recursive rule.

Without a tail-recursive optimisation

```
factorial(0, 1).
factorial(N, F) :-
    N > 0,
    N1 is N - 1,
    factorial(N1, F1),
    F is N * F1.
```

The Prolog interpreter has to store the program state on the stack before making each recursive call.

With a tail-recursive optimisation

```
factorial2(N, F) :-
    factorial2_acc(N, 1, F).
factorial2_acc(N, Acc, F) :-
    Acc1 is Acc * N,
    N1 is N - 1,
    factorial2_acc(N1, Acc1, F).
factorial2_acc(0, Acc, Acc).
```

The Prolog interpreter can forget about the program state on the stack before making each recursive call.

Pattern for a tail-recursive optimisation.

```
foo(List, Result) :-
    foo(List, InitialValue, Result).
foo([], CurrentResult, CurrentResult).
foo([Head|Tail], CurrentResult, Result) :-
    Current2 is somefunction(CurrentResult, Head),
    foo(Tail, Current2, Result).
```

4 Search

Sometimes it helps to represent the problem in terms of a tree of possible states. The nodes represent possible situations and the links represent actions. Paths with repeated states are not considered.

4.1 Depth–First Search

Search the tree branch by branch — referred to as *chronological backtracking*.

```
WHILE Queue is not empty (set Queue to be the list [StartNode])
    Let FirstNode to be firs element of Queue (push)
    IF FirstNode is a goal node
        RETURN FirstNode
    ELSE
        Delete FirstNode from start of Queue (pop)
        Add children of FirstNode to front of Queue (push)
RETURN FAILURE
```

And in Prolog.

Executed with dPlan([InitNode], Result).

```
dPlan([Item|RestOfQueue], Item) :-
    success_node(Item).
dPlan([Item|RestOfQueue], Result) :-
    possible_moves(Item, Children),
    append(Children, RestOfQueue, NewQueue),
    dPlan(NewQueue, Result).
```

All the work is involved in decisions:

- How to represent the states and actions.
- How to compute the possible actions and their effects.
- Eliminating already–encountered situations.
- Recovering the plan once a goal state has been found.

The goal here are considered together. The aim is to work on goals independently, and combine the results into a single plan. Planning for one goal typically undoes previously satisfied goals.

4.2 Breadth–First Search

Search the tree row-by-row.

```
Set Queue to be the list [StartNode]
WHILE Queue is non-empty
    Let FirstNode be the first element in Queue
    IF FirstNode is a goal
        RETURN FirstNode
    ELSE
        Delete FirstNode from start of Queue
        Add children of FirstNode to end of Queue
RETURN FAILURE
```

4.3 Hill-Climbing Search

We consider the children in order of nearness to goals.

```
Set Queue to be the list [StartNode]
WHILE Queue is non-empty
    Let FirstNode be the first element in Queue
    IF FirstNode is a goal
        RETURN FirstNode
    ELSE
        Delete FirstNode from start of Queue
        Order children of FirstNode with most promising first
        Add ordered children to start of Queue
RETURN FAILURE
```

The measure of nearness to goals will be problem-dependent.

The strategy of **Best-First Search** is like hill-climbing, except that we order *all* nodes in the queue by nearness to goals.

4.4 A* Search

Examines nodes in increasing order of *cost-so-far* and *underestimate-of-remaining-cost.*

```
Set Queue to be the list [StartNode]
WHILE Queue is non-empty
    Let FirstNode be the first element in Queue
    IF FirstNode is a goal
        RETURN FirstNode
    ELSE
        Delete FirstNode from start of Queue
        Add children of FirstNode to Queue
        Re-order Queue by cost-so-far and underestimate-of-remaining-cost
RETURN FAILURE
```

The algorithm is guaranteed to produce an optimal solution if a solution exists, so it is **finite**.

The **travelling salesman problem**: a salesman has to tour each one of n cities, with given distance between them. Find a complete tour which will minimise the total distance travelled. The time taken to find a solution to this problem increases exponentially with n. This happens because it is a *combinatorially hard optimisation problem*, which is **NP-Complete**, complete for non-deterministic polynomial time. The A* method can be used for the travelling salesman problem, but tends to be breadth–first like in its performance.

For large n, exhaustive search is impractical. Instead, we could start with an old tour and improve it. We then do hill-climbing on this problem space until we reach a local optimum.

In the **towers of Hanoi** problem, every state is reachable from every other state. The length of the shortest path from the start state to a goal state is $2^n - 1$. The length of the longest path from the start state to a goal state *without repetitions* is $3^n - 1$.

All these search algorithms are:

- Easy to implement.
- Typically noncompetitive on large problem instances.
- Use no special information about the structure of the problem.
- Sometimes there is no solution.

4.5 Minimax Procedure

The minimax procedure determines the best move to make, assuming optimal playing by the adversary.

```
minimax(node)
    IF node is at limit of search three
        Let utility be the utility of node
        RETURN [node, utility]
    ELSE IF node is a white node
        Apply minimax to children of node
        RETURN result which has maximum utility
    ELSE IF node is a black node
        Apply minimax to children of node
        RETURN result which has minimum utility
```

Implementation of α-β minimax.

- At maximising nodes:
  - Get value of β from parent.
  - Keep a parameter α set to α to maximum so far returned by children.
  - Return final value of α.
- At minimising nodes:
  - Get value of α from parent.
  - Keep a parameter β set to α to maximum so far returned by children.
  - Return final value of β.
- When processing any node, stop if $\alpha > \beta$.

5 First-Order Logic

A **logic** consist of three components:

1. **Syntax**: which expressions are formulas.
2. **Semantics**: what each formula means.
3. **Proof Theory**: which ones are valid argument?

The **syntax** rules define a set of formulas for a given vocabulary of non–logical symbols, a *signature.* A **signature** Σ consists of a finite set of *individual constants* and a finite sets of *predicates* of various arities. In addition to it, we employ an infinite set of **variables** $(x, y, z, z_1, y_1, x_1, \dots)$ nad a fixed set of **logical constants**: the *universal quantifier* $\forall$, the *existential quantifier* $\exists$, the *boolean connectives* $\land, \lor, \neg, \implies$.

A **term** is a variable or an individual constant or a n-ary function on symbols.

An **atomic formula** is an expression of the form $r(T_1, \ldots, T_n)$ where r is a n-ary predicate and T_k is a term.

A **formula** is defined as:

- All atomic formulas are formulas.
- If φ and ψ are formulas and C is a variable then $(\forall C \varphi)$, $(\exists C \varphi)$, $(\varphi \wedge \psi)$, $(\varphi \vee \psi)$, $(\neg \varphi)$ and $(\varphi \implies \psi)$ are formulas.

Given signature Σ, the set of formula over this signature, defined by the recursive rules above, is $\mathcal{L}(\Sigma)$.
The **semantics** specifies the truth–conditions of formulas in $\mathcal{L}(\Sigma)$.

An **interpretation** over a given signature consists of:

- The **domain** (a set of A elements).
- A function mapping each individual constant c in Σ to an element in A.
- A function mapping each n-ary predicate r in Σ to a set of n-tuples of A.

An **assignment** is a function mapping each variable to an element of A, it is similar to a "temporary" interpretation of variables.

The symbol $\models$ (**models)** is defined recursively.

A formula φ is a **sentence** if it has no free variables.

Identity is expressed by the binary predicate $=$, which is always interpreted $\{< a, a > \mid \in A\}$ for every domain in A.

Equality enables us to count.

A signature can also contain **function symbols**, used to build complex terms.

An **argument** consists of a finite set of **premises** and a c**conclusion**.

6 First-Order Logic and AI

We use **situation variables** to relativise the predicate calculus description to the situations in which they hold. The **situation calculus** is the predicate calculus used with situation variables to reason about changing environment. We name situations by means of the actions that produce them.

We can use theorem–proving techniques to verify that a plan will achieve a given set of goals.

The **SATCHMO theorem prover** (Manthey and Bry, 1989) works on clauses of the form $a_1 \wedge \cdots \wedge a_m \implies b_1 \vee \cdots \vee b_n$ or $\top \implies b_1 \vee \cdots \vee b_n$ or $a_1 \wedge \cdots \wedge a_m \implies \bot$, where a_i and b_i are positive literals.

Any clause can be put in this form by putting the negative literals on the left of $\implies$ and the positive ones on the right.

Limit clauses to those which are **range-restricted**: every variable in the consequent of a clause occurs in the antecedent as well. "Every set of clauses can be transformed into an equisatisfiable set of range-restricted clauses."

Transformation process:

- Any $\top \implies C$ containing $x_1, \ldots, x_n$ becomes $\mathrm{dom}(x_1) \wedge \mathrm{dom}(x_2) \wedge \cdots \wedge \mathrm{dom}(x_n) \implies C$.
- Any $A \implies C$ with C containing $x_1, \ldots, x_n$ but not in A becomes $A \wedge \mathrm{dom}(x_1) \wedge \cdots \wedge \mathrm{dom}(x_n) \implies C$.

- For any constant c in any clause, the clause $\top \implies \text{dom}(C)$ is added.
- For every n-ary function-symbol f in any clause, $\text{dom}(x_1) \wedge \cdots \wedge \text{dom}(x_n) \implies \text{dom}(f(x_1, \ldots, x_n))$ is added.

We would like to prove facts using SATCHMO, by translating the axioms into SATCHMO clauses: we face two problems though.

1. The treatment of equality. SATCHMO will attempt to satisfy it by asserting it to the database Such attempt will result in an error-message. We solve this by preventing the assertion of false identity statements.
2. Returning the plan. We can save important results.

SATCHMO is a first-order theorem prover implemented in a page of Prolog. It has nice termination properties, but it is not practically very useful. You first have to convert to clause form.

7 First order logic and Prolog

All Prolog assertions correspond to formulas in the predicate calculus. Prolog maps systematically onto logic as follows:

- Facts map to atomic formulas.
- Variables are treated as universally quantified.
- Rules map to conditional statements.

A **literal** is an atomic proposition or an atomic proposition letter prefixed by $\neg$.

A **clause** is a (possibly empty) collection of literals joined by $\vee$.

Putting the formula in clause form is the process of transforming any quantifier–free formula into a logically equivalent conjunction of clauses.

Prenex form: a formula in the predicate calculus in which all the quantifiers are at the front. Any formula is logically equivalent to a formula with all the quantifiers at the front. Given a prenex formula, any existential quantifier can be eliminated.

The formulas that are expressible in Prolog are those whose normal forms are conjunctions of clause-form formulas having at most one positive literal, these are called **Horn clauses**.

Skolemisation is the process of replacing existentially quantified variables with **Skolem constants** and **functions**. This process does not produce logically equivalent formulas but it does produce *equisatisfiable* formulas.

Since we can equisatisfiably convert to clausal forms, we need only inference procedure which work on clausal forms.

8 Natural Language Grammar I

Pied-piping: phenomenon of syntax whereby a given focused expression takes an entire encompassing phrase with it when it is "moved".

Italian is known as a **pro-drop** language (so that you can omit the subject from a sentence) while English is a **non-pro-drop** language.

The **head** (the main word often comes at the end in German and at the beginning in English: German is predominantly **head-final** and English predominantly **head-initial**.

Syntax is the study of sentence structure. Certain basic principles of syntax are universal; languages vary in the "settings" of certain parameters. The task of theoretical linguistics is to elicit these principles and parameters.

There are different formalisms for describing a grammar:

- Transformational grammar.
- head driven phrase-structure grammar.
- Tree–adjoining grammar.
- Type–logical grammar.
- Model–theoretic grammar.

Languages recognised by **Finite State Automata** are called **regular languages**.
Languages recognised by **Context–Free Grammars** are called **context–free languages**.
Languages recognisable by **Context–Sensitive Grammars** are called **context–sensitive languages**.

Over time, the dominant formalism became **transformational grammar**, which has very high expressive power.

Supposition: every grammatical sentence has a **phrase structure**. The sentence is divided into **phrases**, corresponding to nodes of the tree. Each phrase has a **category**.

Some common **terminal** categories (or part of speech):

- N (noun).
- V (verb).
- Adj (adjective).
- Adv (adverb).
- P (preposition).
- Det (determiner).

Some common **non–terminal** or **phrasal** categories:

- S (sentence).
- NP (noun phrase).
- VP (verb phrase).
- PP (preposition phrase).

We are interested in a **descriptive** theory of language. Phrase structure is assigned on the basis of *evidence*.

Sentences arise through **transformations**, in which material is moved around.

Pumping lemma: you have to repeat categories in derivation.

9 Natural Language Parsing I

A grammar can be written as a Prolog **definite clause grammar (dcg)**.
Examples.

```
s -> np, vp.      det -> [some].     n -> [girl].     v -> [loves].
np -> det, n.     det -> [every].    n -> [boy].
vp -> v, np.
```

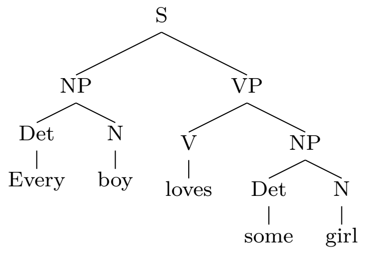
When this file is loaded and consulted, it is translated into ordinary Prolog clauses. Prolog dcg can include variables, which can pass agreement information around the sentence.

Curly-bracketed material is not pre-processed by the Prolog compiler, it turns off pre-processing.

Strings in Prolog are lists of ASCIIs.

Grammars for natural language parsing can be written directly in Prolog. Grammar rules are identified as rules with –> as the main functor. The Prolog compiler pre-processes these grammar rules.

The words to be processed are represented as a difference list. Square brackets recover words from the difference list.



10 Natural Language Grammars II

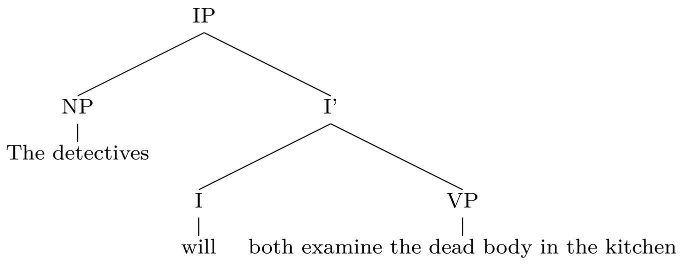
The standard analysis takes the auxiliaries to be **inflections (I)**. Sentences containing them are said to be **inflection phrases (IPs)**.

According to the **X–bar theory** of natural language syntax, there are 3 levels of phrase:

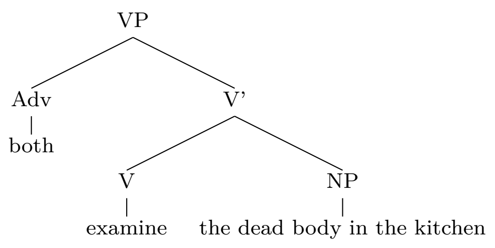
1. The **head (I)**, combined with its **complement** forms.
2. An **intermediate projection (I')**, combined with a **specifier**.
3. A **maximal projection (IP)**.

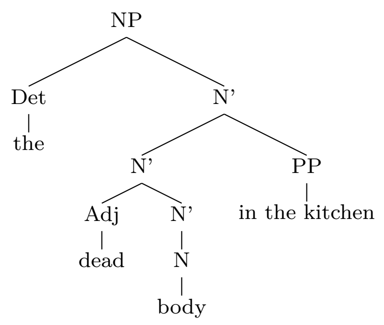
Intermediate projections combined with **adjuncts** to produce further intermediate projections. Complements are obligatory while adjuncts are not.

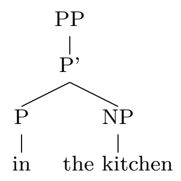
"The detectives will both examine the dead body in the kitchen". Here "The detectives" is the **specifier**, "will" is the **head** and "both examine the dead body in the kitchen" is the **complement**.

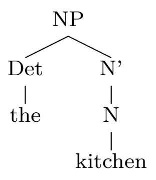


We can construct trees for intermediate projections.

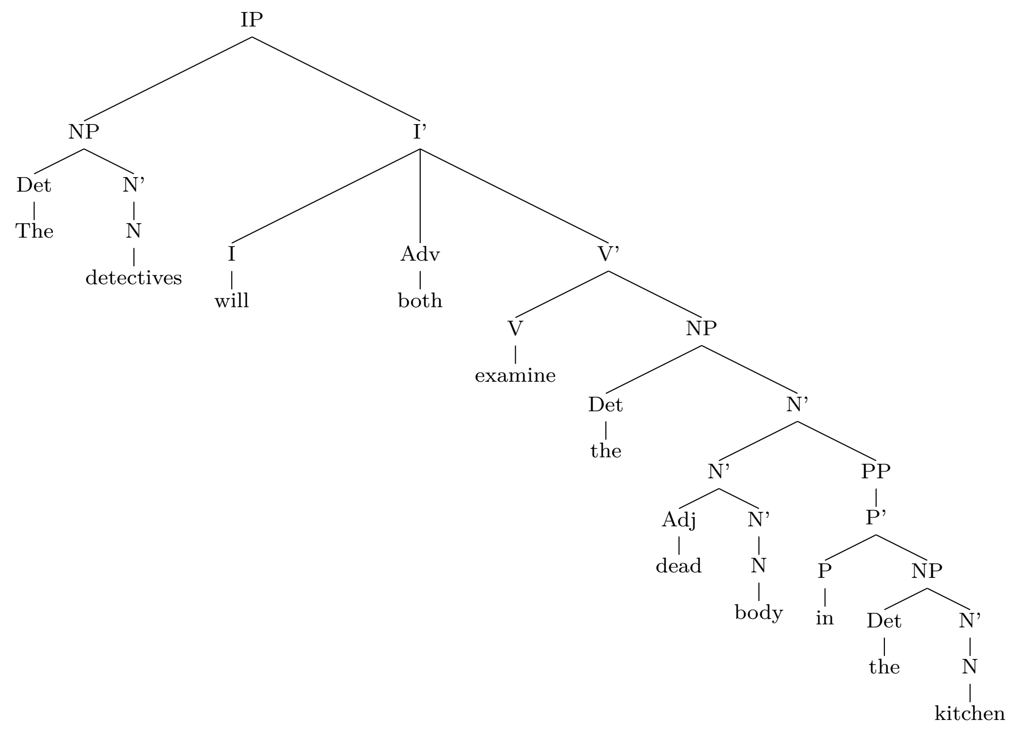








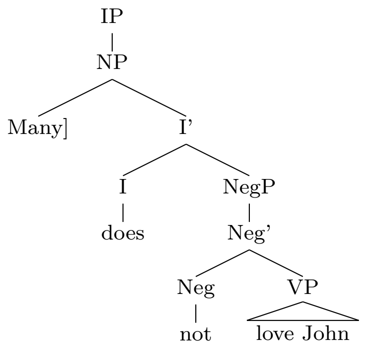
The final tree.



When there is an auxiliary, the auxiliary is **finite** and the verb is **infinite**. When there is *no* auxiliary, the verb is **finite** — it has **sense**, the verb migrates upwards to join the inflection (**head movement**).

The **deep structure** is the initial structure generated by the underlying grammar. The **surface structure** is the structure that results after movement. This is true in the transformational grammar described by Noam Chomsky.

Negation is not available for sentences without auxiliaries, a **do-support** is required. Not heads a **negation phrase (NegP)**. The requirement for the do-support can be imposed by decreeing that a verb cannot move out of a NegP to join the I.



11 Natural language Parsing II

The grammar postulates a two-stage process of sentence formation:

1. **Generation**: deep structure.
2. **Movement:** surface structure.

```
ip(Sentence) :-
    setof(IPSem,
          ip(IPSem, Sentence, []),
          IPSems),
    treeP(IPSems).
ip(ip(NPSyn, VPSyn)) ->
    np(NPSyn), ibar(VPSyn).
ibar(ibar(ISyn, VPSyn)) ->
    i(ISyn), vp(VPSyn).
i(i(Aux)) ->
    [Aux],
    {isAux(Aux)}.

det(det(Det)) ->
    [Det],
    {isDet(Det)}.
```

```
np(np(DetSyn, NbarSyn)) ->
    det(DetSyn),
    nbar(NbarSyn).
nbar(nbar(NSyn))->
    n(NSyn).
n(n(Noun)) ->
    [Noun],
    {isNoun(Noun)}.

vp(vp(VbarSyn)) ->
    vbar(VbarSyn).
vbar(vbar(VSyn, NPSyn)) ->
    v(VSyn),
    np(NPSyn).
v(v(Verb)) ->
    [Verb],
    {isVerb(Verb)}.
```

We include new rules for movement.

```
ip(ip(NPSyn, VPSyn)) ->
    np(NPSyn),
    ibar(VPSyn).
ibar(ibar(ISyn, VPSyn)) ->
    i(ISyn, MvdVbL),
    vp(VPSyn, MvdVbL).
i(i(Aux),[]) ->
    [Aux],
    {isAux(Aux)}.
i(i(pastInfl), [Verb]) ->
    [InflVerb],
    {pastInfl(Verb, InflVerb),
     isVerb(Verb)}.

vp(vp(VbarSyn), MvdVbL) ->
    vbar(VbarSyn, MvdVbL).
vbar(vbar(VSyn, NPSyn), MvdVbL) ->
    v(VSyn, MvdVbL),
    np(NPSyn).
v(v(Verb), []) ->
    [Verb],
    {isVerb(Verb)}.
v(v(MvdVb), [MvdVb]) ->
    [].
```

12 Natural Language Semantics I

Higher-order functions take functions as arguments.

Lambda-calculus

- α**-conversion**: renaming of bound variables, $\lambda x[\varphi(x)] \implies \lambda y[\varphi(y)]$ (y variable not occurring free in y).
- β**-reduction**: applying lambda functions to their argument, $(\lambda x[\varphi(x)]\ \psi) \implies \varphi(\psi)$. Here, $(\lambda x[\varphi(x)]\ \psi)$ is the **redex** and $\varphi(\psi)$ is the **reduct**.

Semantic information.
S/$(\varphi\ \psi) \rightarrow$ NP/φ, VP/ψ
VP/$\psi \rightarrow$ Vi/ψ
VP/$(\varphi\ \psi) \rightarrow$ V/φ, NP/ψ

An example could be $\lambda p[\lambda q[\forall x((p\ x) \rightarrow (p\ q))]]$ which translates to Prolog as `lbd(p, lbd(q, forall(x, px -> qx)))`, using the syntax defined in the labs.

Substitution predicate: expression B is the result of substitution an occurrence of expression φ for each occurrence of expression X in expression A.

S/$\varphi(\psi) \rightarrow$ NP/φ, VP/ψ. is translated in Prolog as:

```
s(SSem) -->
    np(NPSem),
    vp(VPSem),
    {var_replace(NPSem, NPSem1),
     beta(NPSem1@VPSem, SSem)}.
```

```
ip(Sentence) :-
   ip(SSem, Sentence, []),
   printF(SSem).

ip(SSem) -->
   np(NPSem),
   ibar(IbarSem),
   {var_replace(NPSem, NPSem1),
    beta(NPSem1@IbarSem, SSem)}.
ibar(VPSem) -->
   i(MvdVbL),
   vp(VPSem, MvdVbL).
i([]) -->
    [Aux],
    {isAux(Aux)}.
i([Verb]) -->
   [InflVerb],
   {pastInfl(Verb, InflVerb)}.
```

```
vp(VbarSem, MvdVbL) -->
   vbar(VbarSem, MvdVbL).
vbar(VbarSem, MvdVbL) -->
   tv(VSem, MvdVbL),
   np(NPSem),
   {var_replace(VSem, VSem1),
    beta(VSem1@NPSem, VbarSem)}.

tv(lbd(s, lbd(x, s@lbd(y, Fla))), []) -->
   [Verb],
   {isTransVerb(Verb), Fla=..[Verb, x, y]}.
tv(lbd(s, lbd(x, s@lbd(y, Fla))), [MvdVb])
    -->
   [],
   {isTransVerb(MvdVb), Fla=..[MvdVb, x, y
     ]}.
```

13 Natural Language Semantics II

For **adjectives**:

- The augmented syntax rule is:
  - N'/$\varphi(\psi) \rightarrow$ Adj/φ, N'/ψ.
- The dcg rule is:

```
nbar(NbarSem) ->
    adj(AdjSem, nbar(NbarSem1),
    {var_replace(AdjSem, AdjSem1),
     beta(AdjSem1@NbarSem1, NbarSem)}.
```

- The meaning assignment is $[\![Adj_{TALL}]\!] = \lambda p[\lambda x[p(x) \wedge \texttt{tall}(x)]]$

For **not**:

- The augmented syntax rule is:
  - I'/$\varphi \rightarrow$ I, NegP/φ
  - NegP/$\varphi \rightarrow$ Neg'/φ
  - Neg'/$\varphi(\psi) \rightarrow$ Neg/φ, VP/ψ
- The meaning assignment is $[\![Neg_{NOT}]\!] = \lambda p[\lambda x[\neg p(x)]]$

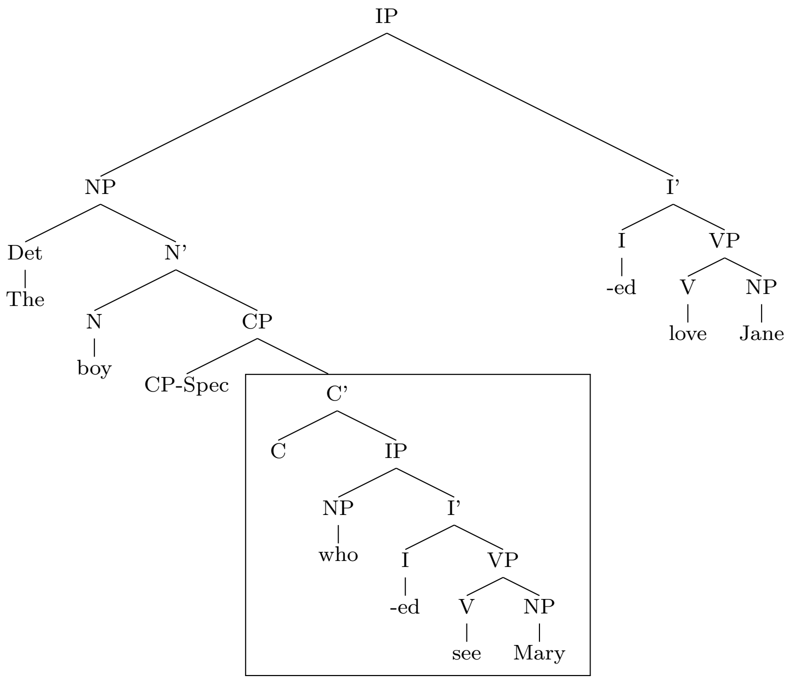
For **noun-attached prepositions**:

- The augmented syntax rule is:
  - I'/$\varphi(\psi) \rightarrow$ N'/ψ, PP/φ
  - PP/$\varphi \rightarrow$ P'/φ
  - P'/$\varphi(\psi) \rightarrow$ P/φ, NP/ψ
- The meaning assignment is $[\![P_{WITH}]\!] = \lambda s[\lambda p[\lambda x[p(x) \wedge s(\lambda y[\texttt{with}(x,y)])]]]$

For example "Every boy saw Mary with a telescope" is:
$\forall x(\texttt{boy}(x) \rightarrow$
  $\exists e(\texttt{seeing}(e, x, mary) \wedge$
    $\exists y(\texttt{telescope}(y) \wedge \texttt{with}(e, y))))$

14 Natural Language Parsing III

"The boy who saw Mary loved Jane" is a **relative clause**: syntactically, they are *adjuncts*, so **complementiser phrases (CPs)**. CPs contains little sentences with references to the subject missing: "The boy who [(The boy) saw mary] loved Jane" includes a **wh-movement** inside the CPs.



15 Natural Language Semantics III

"Every boy who kissed Mary kissed Jane" is translated as $\forall x(\texttt{boy}(x) \land \texttt{kissed}(x, Mary) \rightarrow \texttt{kissed}(x, Jane))$.

A relative pronoun is extracted from the C' where it lives in deep structure. Thus, the remaining C' is dependent on the extracted relative pronoun, an **index** needs to express this dependency.

For **relative clauses**:

- The augmented grammar rule is:
  - N'/$\varphi(\psi) \rightarrow$ N'/ψ, CP/φ
  - CP/$\lambda p[\lambda x[p(x) \land \varphi]] \rightarrow$ CP-Spec$_x$, C'/φ
  - C'/$\varphi \rightarrow$ C, IP/φ
- The meaning assignment is $[\![NP_{WHO}]\!] = \lambda p[p(x)]$, where x is the index of the relative pronoun and it is an unquantified free variable.

16 Resolution Theorem–Proving

First we need to transform the formula to Clause Form. For example $\forall x(b(x) \rightarrow \exists y(g(y) \land love(x, y)))$.

1. Bring quantifiers to the front to have **prenex form**.
   $\forall x \exists y(b(x) \rightarrow (g(y) \land love(x, y)))$
2. Skolemise.
   $\forall x(b(x) \rightarrow (g(f(x)) \land love(x, f(x))))$

3. Remove universal quantifiers, they convey no information anymore.
   $b(x) \to (g(f(x)) \land love(x, f(x)))$

4. Put it into clausal form (in clauses).
   Based on $A \to (B \land C) \equiv (\neg A \lor B) \land (\neg A \lor C)$, we can rewrite the formula as $(\neg b(x) \lor g(f(x))) \land (\neg b(x) \lor love(x, f(x)))$.

Resolution is the basis of many widely–used theorem–provers today. With SATCHMO, we first convert to **clause form**, so **literals** or collection of literals adjoined by $\lor$, where each literal is either an atomic formula or an atomic formula prefixed by $\neg$.

Any quantifier free formula is logically equivalent to the conjunction of some collection of clauses. A formula in the predicate calculus in which all quantifiers are at the front is said to be in **prenex form**. Any formula is logically equivalent to a formula with all the quantifiers at the front.

Given a prenex formula, existential quantifiers can be eliminated:

- $\exists x(man(x) \land philosopher(x))$ is equisatisfiable with $man(a) \land philosopher(a)$ provided that a is a new variable, not occurring in any other formula.
- $\exists y \forall x(loves(x, y))$ can be rewritten as $\forall x(loves(x, b))$, where b is a **Skolem constant**. In this case every x loves the same person b.
- $\forall x \exists y(loves(x, y))$ can be rewritten as $\forall x(loves(x, f(x)))$, where $f(x)$ is a **Skolem function**. In this case everyone loves someone but not necessarily the same person.

We need a Skolem function whenever we want to eliminate an existential quantifier which is to the right of some universal quantifier.

Skolemisation: process of replacing existentially quantified variables. It *does not produce logically equivalent* formulas, but *equisatisfiable* ones.

Ground case: gives us a way to make inferences.

- Modus Ponens: $\dfrac{p \to q \qquad p}{q}$
- Re-writing: $\dfrac{\neg p \lor q \qquad p}{q}$
- Generalizing to longer clauses: $\dfrac{\neg p \lor C \qquad p \lor D}{C \lor D}$

To test **validity**, we convert the initial formula into clausal form and try to use resolution and factoring to obtain a contradiction. If we succeed, the sequent is valid.

For example, $boy(john) \to (girl(mary) \to loves(john, mary))$ in clausal form is $\neg boy(john) \lor \neg girl(mary) \lor loves(john, mary)$. If we apply resolution repeatedly:

$$\dfrac{\neg loves(john, mary) \qquad \dfrac{boy(john) \qquad \dfrac{girl(mary) \qquad loves(john, mary) \lor \neg boy(john) \lor \neg girl(mary)}{loves(john, mary) \lor \neg boy(john)}}{loves(john, mary)}}{\perp}$$

The formula we have started with are not satisfiable because the **empty clause** has been derived.

To apply resolution to *non–ground* clauses, we need the concept of **unification**: if A and A' are atoms, A and A' are **unifiable** if there is a substitution σ of terms for variables such that $A\sigma = A'\sigma$, for example $man(x) = man(socrates) \quad \sigma : x \mapsto socrates$, or $q(x, f(x)) = q(u, v) \quad \sigma : u \mapsto x, f(x) \mapsto v$.

If A and A' are unifiable, then there is a substitution θ such that $A\theta = A'\theta$ and, for any substitution σ such that $A\sigma = A'\sigma$, we have $\sigma = \theta p$ for some substitution p. In this case θ is the **most general unifier (m.g.u.)**, which is unique up to renaming of variables.

Non–ground case: there are free variables. Unification must be performed.

- Modus Ponens: $\dfrac{\neg p(x) \vee q(x) \qquad p(a)}{q(a)}$
- Generalizing: $\dfrac{\neg A \vee C \qquad A' \vee D}{(C \vee D)\theta}$

The last example is the **resolution rule**, where A and A' are unifiable atoms. We can remove literals which are repeated in the clause. Generalising the non–ground case gives the **factoring rule** $\frac{C}{(C\theta)^o}$, where o denotes the deletion of repeated literals.

If a clause C is derivable from a set $\mathscr{C}$ of clauses by means of resolution and factoring, $\mathscr{C} \vdash C$.

Theorem. Let $\mathscr{C}$ be a set of clauses. Then $\mathscr{C} \vdash \bot$ if and only if the universal closure of $\mathscr{C}$ is unsatisfiable. Resolution and factoring are all the rules we need to determine entailments. The proof system in question is **sound** and **complete**.

Let A be a collection of all atoms over some signature. Let $\prec$ be a partial order on A. Thus, $\prec$ is an **A–ordering** if, for all substitution θ, $A \prec A'$ implies $A\theta \prec A'\theta$. We extend any A–ordering $\prec$ to literals by ignoring negations. For example, $A \prec^2 A'$, if and only if A involves a unary predicate and A' involves a binary predicate. Given an A–ordering $\prec$, **ordered resolution** is the same as resolution, see examples above, subject to added restriction that, for every literal L in C, $A \not\prec L$, and for every literal L' in D, $A' \not\prec L$.

Theorem. Let $\mathscr{C}$ be a set of clauses and $\prec$ an A–ordering. Then $\mathscr{C} \vdash_{\prec} \bot$ if and only if $\mathscr{C}$ is unsatisfiable.

The **1-variable fragment** is the set of clauses C satisfying either of the following conditions:

1. Every literal of C is ground.
2. There is variable x such that, for every literal L of C, $Vars(L) = x$.

For example:

- These are in the 1-variable fragment: $p(a) \vee q(b, f(a))$ and $p(x) \vee \neg r(h(x, g(x)), x)$.
- While these are not: $p(a) \vee q(x, f(a))$ and $p(x) \vee \neg r(y, x)$

Resolution and factoring preserve membership in the 1-variable fragment. Suppose we take a set $\mathscr{C}$ of clauses in the 1-variable fragment, and keep applying resolution and factoring them. Obviously, only clauses featuring the signature of $\mathscr{C}$ will be generated because the functional depth is not increased. the total number of clauses that can be generated from $\mathscr{C}$ by $\prec$d-ordered resolution and factoring is bounded by an exponential function of the size of the signature. Hence, the satisfiability of clauses in the 1-variable fragment with a fixed–depth bound can be algorithmically decided in exponential time.

17 Grand Finale

Resolving or factoring clauses with at most two literals produces only clauses with at most two literals. $\dfrac{\neg p(x) \vee q(x) \qquad \neg q(x') \vee \neg(x')}{\neg p(x) \vee \neg r(x)}$

Resolving, or factoring, clauses with no function-symbols introduces no new terms or predicates.

Suppose we have a set E of sentences in our syllogistic fragment, composing n words. The translation to first–order logic and hence to clause form will yield at most n clauses over a signature of at most n predicates and at most n individual constants, with each clause being function–free and having at most two literals. There are at most $(2n(n+1))^2$ such clauses, so a resolution theorem prover working on these clauses will stop in time bounded by a polynomial function of n.

Fragment	Computational Complexity
cop	PTIME
cop + rel	NP–Complete
cop + TV + DTV	PTIME
cop + rel + TV	EXPTIME–Complete
cop + rel + TV + DTV	NEXPTIME–Complete
cop + rel + TV + RA	NEXPTIME–Complete
cop + rel + TV + GA	Undecidable
cop + rel + TV + DTV + RA	NEXPTIME–Complete
cop + num	NP–Complete
cop + TV + num	NEXPTIME–Complete

Table 1: Complexities of different fragments

A First-Order Logic

Vocabularies tell us which first–order *languages* and *models* (situations) belong together. They tell us *what* they we are going to be talking about and *how* we are going to talk bout these things.

A **model** is a **semantic** entity: it contains the kinds of things we want to talk about. It tells us which collection of entities we are talking about (the **domain** of the model) and it gives an appropriate semantic value to each symbol in the vocabulary through **interpretation functions**. Each constant should be interpreted as an element of the domain and each n–place relation should be interpreted as an n–place relation in the domain. The domain is made up of **stars**, who are names, and **extras**, who are not. One entity may have several names.

A.1 First-Order Language

1. All the symbols in the vocabulary, **non–logical** symbols of the language (choice of vocabulary).
2. A countably infinite collection of variables $x, y, z, \ldots$.
3. The boolean connectives $\neg$, $\wedge$, $\vee$ and $\implies$.
4. The quantifiers $\forall$ **universal quantifiers** and $\exists$ **existential quantifier**.
5. The round brackets and the comma are used to group symbols.

A first–order **term** τ is any constant or any variable, like noun phrases of first–order logic.

Atomic formulas are the combination of noun phrases and predicates. if R is a relation symbol of arity n, and $t_1, \ldots, t_n$ are terms, then $R(t_1, \ldots, t_n)$ is an atomic/basic formula.

Well formed formulas:

- All atomic formulas are well formed formulas.
- If ϕ and ψ are well formed formulas, then so are $\neg\phi$, $(\phi \vee \psi)$, $(\phi \wedge \psi)$ and $(\phi \implies \psi)$.
- If ϕ is a well formed formulas, and x is a variable, then both $\exists x\phi$ and $\forall x\phi$ are well formed formulas.

First–order formulas of the form $\exists x\phi$ and $\forall x\phi$ are called **quantified formulas**.

The **subformulas** of a formula ϕ are ϕ itself and all the formulas used to build ϕ.

A.2 Free vs. Bound

1. Any occurrence of any variable is free in any atomic formula.
2. If an occurrence of any variable is free in ϕ or in φ, then that same occurrence is free in $\not\phi$, $(\phi \vee \psi)$, $(\phi \wedge \psi)$ and $(\phi \implies \psi)$.
3. If an occurrence of a variable x is free in ϕ, then that occurrence is free in $\forall y\phi$ and $\exists y\phi$. However, no occurrence of x is free in $\forall x\phi$ and $\exists x\phi$.
4. The only free variables in a formula ϕ are those whose freeness follows from the preceding clauses. Any variable in a formula that is not free is said to be bound.

A **sentence** is a formula containing no occurrences of free variables.

Propositional logic is the quantifier free part of first–order logic.

Satisfaction is a three place relation which holds between a formula, a model and an assignment of values to variables. **Assignments** tell us what the free variable stands for.

If we want to use first–order logic to model natural language semantic, it is sensible to think in terms of 3 components: first–order formulas (descriptions), first–order models (situations) and variable assignments (contexts).

Variant assignments allows us to try out new values.

Let $\mathcal{M} = (D, F)$ be a model, let g be an assignments in $\mathcal{M}$ and τ be a term. Then, the **interpretation** of τ with respect to $\mathcal{M}$ and g is $F(\tau)$ if τ is a constant, and $g(\tau)$ is τ is a variable.

A sentence ϕ is **true** in a model $\mathcal{M}$ if and only if for any assignments g of values to variables in $\mathcal{M}$, we have that $\mathcal{M}, g \models \phi$. Start with whatever assignment you like, the result will be the same.

A term is said to be **closed** if it contains no variables.

The atomic formula $\tau_1 = \tau_2$ is satisfied if τ_1 and τ_2 have exactly the same interpretation.

Formal semantics: the business of giving model-theoretic interpretations to fragments of natural language.

Computational semantics: business of using a computer to build such representation.

Inference is the process of making *implicit* information *explicit*. It has three main tasks:

1. **Querying task**: given a model $\mathcal{M}$ and a first–order formula ϕ, is ϕ satisfied in $\mathcal{M}$ or not? First–order logic plays the role of the database query language, and the model plays the role of the database. It is possible to translate some kinds of natural language questions into first–order logic and see if their translations are satisfied in the model. **Model checker**: program that performs querying tasks.

2. **Consistency checking task**: given a first–order formula ϕ, is ϕ consistent or inconsistent? A formula is consistent if it makes sense, is intelligible, describe something reliable. It is natural to identify the pre–theoretic concept of consistency with the model–theoretic concept of satisfiability and to identify inconsistency with unsatisfiability. Satisfiable formulas are those which describe conceivable/realisable solutions. **Computationally undecidable task**: three is no algorithm capable of solving consistently checking for all possible input formulas. Given a formula, we have to determine if somewhere out there in the mathematical universe of models, a satisfying model exists.

3. **Informativity checking task**: given a first–order formula ϕ, is ϕ informative (invalid) or uninformative (valid)? A **valid** formula is a formula that is satisfied in all model given any variable assignments. Suppose $\phi_1, \ldots, \phi_n$ and ψ are a finite collection of first–order formulas. Then we say that the argument with *premises* $\phi_1, \ldots, \phi_n$ and *conclusion* ψ is a valid argument if and only if, whenever all the premises are satisfied in some model, using some variable assignment, the conclusion is satisfied in the same model using the same variable assignments. Valid formulas are uninformative: they do not tell us anything at all about any particular model, the do not rule out possibilities. Invalid formulas are informative. Uninformativity can be a sign that something is going wrong with the communication process. Lack of informality is not such a reliable indicator of communicative problems as inconsistency. The informativity checking task is **undecidable**: validity means satisfiable in all models, and there are an awful lot of awfully big models.

A.3 Relating

1. ϕ is *consistent* if and only if $\neg\phi$ is *informative.*
2. ϕ is *inconsistent* if and only if $\neg\phi$ is *uninformative.*
3. ϕ is *informative* if and only if $\neg\phi$ is *consistent.*
4. ϕ is *uninformative* if and only if $\neg\phi$ is *inconsistent.*

Why? Suppose ϕ is consistent, then ϕ is satisfiable in at least one model, so there is at least one model where $\neg\phi$ is not satisfied. Hence, $\neg\phi$ is informative.

A.4 The satisfaction definition in Prolog

The four arguments that `satisfy/4` takes are:

1. The formula to be tested.
2. The model.
3. A list of assignments.
4. A **polarity feature** (is a “flag” that records whether we are trying to see if a particular subformula is true or false) which tells you whether a formula should be positively or negatively evaluated.

B First–Order Inference

Unification is the process of carrying out substitutions on two terms so that they become identical.

Propositional Resolution is a two step process:

1. **Clausal normal form**: the input formula is converted to conjunctive normal form and all the quantifiers are eliminated along the way.
    (a) Put the formula into Negation Normal Form.
    (b) Skolemise away any existential quantifiers. We can always convert a formula ϕ to a formula ϕ^s that contains no existential quantifier without doing semantic damage. When we skolemise away an existential quantifier that is under the scope of universal quantifiers $\forall x_1, \ldots, \forall x_n$, then instead of using simple Skolem constants we should use a Skolem term of the form $\mathcal{S}(x_1, \ldots, x_n)$.
    (c) Discard any universal quantifiers.
    (d) Put the resulting quantifier–free formula into (set) conjunctive normal form.
2. **Resolution phase**: resolution rule and unification. Sometimes we need to relabel variables before we unify. A **non-redundant factor** $\mathcal{F}$ of a clause C is a clause obtained by applying unification to literals within C as often as possible. A clause my have more than one non-redundant factor. If none of the literals within the clause C can be unified, then C itself is its own unique non-redundant factor.

C Learn Prolog Now - Lists

A **list** is a finite sequence of elements. We can specify lists in Prolog by enclosing the elements of the list in square brackets, elements being separated by commas, The *length* of a list is the number of elements it has. All sorts of Prolog objects can be in a list. An item may occur more than once in a list. The **empty list** contains no elements, and its length is 0. Lists can contain other lists as elements.

Any non-empty list can be thought of as consisting of two parts: a **head**, the first item in the list, and a **tail**, the remaining list when we take the first element away. The empty list has no internal structure, neither a head nor a tail.

Prolog has a special built-in operator `|` which can be used to decompose a list into its head and tail. Its most obvious use is to extract information from lists, and we do this by using `|` together with unification. For example:

```
[X|Y] = [mia, vincent, jules]     X = mia, Y = [vincent, james]
[X|Y] = []                        no. % It fails
[X, Y|Z]                          X = first element, Y = second element,
                                  Z = rest of the list
```

It can be used to split a list at any point, to the left of `|` we simply indicate how many elements we want to take away from the front of the list and then to the right of `|` we will get what remains.

To obtain only the second and fourth element we can do `[_A, B, _C, D|_T]`. Here `_` is the **anonymous variable**. it is used when we need to use a variable but we are not interested in what Prolog instantiates the variable to. Each occurrence of `_` is independent, each it is bound to something different. For example `[_, _, [_|X]|_]` extract the tail of the internal list as the first element of the current one.

Is the element in the list?

```
% An object X is in the list if it is the head of that list
member(X, [X|_T]).
% An object X is a member of a list if it is a member of the tail of that list
member(X, [_H|T]) :-
    member(X, T).
```

The reason that this recursion is safe is that end of the line Prolog has to ask a question about the empty list. The empty list cannot be broken down into smaller parts and this allows a way out of recursion.

D Learn Prolog Now - Arithmetic

$6+2=8$ means "Query: 8 is $6+2$? Yes.".
$6+2=?$ means "Query: X is $6+2$, so $X=8$". We can use arithmetic operations when we define predicates.

Prolog understand the usual conventions we use for disambiguate arithmetical expression.

The sign $=$ performs **unification**, so `X = 3 + 2` returns `X = 3 + 2` because Prolog simply unifies the variable `X` to the complex term `3 + 2`.

To force Prolog to evaluate arithmetic expression we have to use `is`. For example `X is 6 + 2` gives back `X = 8`, or `X is +(6, 2)` or `is(X, +(6, 2))`. On the other hand, `6 + 2 is X` gives an error. Altough we are free to use variables on the right hand side of `is`, when we carry out evaluation the variables mus already have been instantiated to a variable–free arithmetic expression. If the variable is uninstantiated, or if it is instantiated to something other than an integer, we will get an instantiation error message. This is because arithmetic is not performed using Prolog's usual unification and knowledge base search mechanisms.

How long is a list?

```
len([], 0).
len([H|T], N) :-
    len(T, X),
    N is X + 1.
```

Some Prolog arithmetic predicates actually do carry out arithmetic all by themselves (without `is`), like the operators that compare integers. They force both their right hand and left hand arguments to be evaluated.

`=` tries to unify its arguments, while `=:=` forces arithmetic evaluation.

Find the maximum in a list using an accumulator.

```
accMax([], A, A).

% Is head larger than current max?
accMax([H|T], A, Max) :-
    H > A,
    accMax(T, H, Max).

% Is head smaller than current max?
accMax([H|T], A, Max) :-
    H =< A,
    accMax(T, A, Max).
```

E Learn Prolog Now - More Lists

`append(L1, L2, L3)` will hold when the list `L3` is the result of *concatenating* lists `L1` and `L2` together. It can also be used to split up a list like in `append(X, Y, [a, b, c, d])`.

```
% Appending an empty list to any list yelds the same list
append([], L, L).

% When we concatenate a non empty list [H|T] with L2, we end up with the list
    whose head is H and whose tail is the result of concatenating T with L2
append([H|T], L2, [H|L3]) :-
    append(T, L2, L3).
```

The recursive calls to `append/3` build up a nested pattern of variables which code up the required answer.

```
% A list P is the prefix of list L when there is some list such that L is the result of
    concatenating P with that list
prefix(P, L) :-
    append(P, _, L).

suffix(S, L) :-
    appen(_, S, L).

% SubL is a sublist of L if there is some suffix S of L of which SubL is a prefix
sublist(SubL, L) :-
    suffix(S, L),
    prefix(SubL, S).
```

`append/3` is *inefficient*. Prolog works down the length of the first list.

```
% Take the head of the list and add it as the head of the accumulator which contains the
    reversed list we want
inefficient_reverse([], []).
inefficient_reverse([H|T], R) :-
    inefficient_reverse(T, RevT),
    append(RevT, [H], R).

% Efficient because we do not have to waste time carrying out concatenation or other
    irrelevant work, must be initialised with rev(L,R) :- accRev(L, [], R).
accRev([], A, A).
accRev([H|T], A, R) :-
    accRev(T, [H, A], R).
```

To reverse an eight element list, the inefficient one takes 90 steps while the other one takes only 20.

F Definite Clause Grammars

Notation for writing grammars that hides the underlying difference list variables.

A **context–free grammar** is a finite collection of rules which tell us that certain sentences are grammatical and what their grammatical structure actually is. A **context–free rule** consists of a simple non-terminal symbol, followed by `->`, followed by a finite sequence made up of terminal and/or non-terminal symbols. A **context–free language** is a language that can be generated by a context–free grammar.

Terminal symbols only decorate the nodes right at the bottom of the tree while non-terminal symbols only decorate nodes that are higher up in the tree.

Parse tree gives us information about the strings and the structure.

The key idea underlying **difference lists** to represent the information about grammatical categories not as a single list but as the difference between two lists. I I consume all the symbols on the left, and leave behind the symbols on the right, then I have the sentence I am interested in: the difference between the content of the two lists.

It is easy to write context–free grammars that generate infinitely many sentences, using recursive rules.

The order of the goals determine the order of words in a sentence.

We would like our programs not only to tell us which sentences are grammatical but also to give us analysis of their structure.

Extra arguments can be used to build semantic representations.

Any program whatsoever can be written in DCG notation.

DCGs have the tendency to loop when the goal ordering is wrong. They are best viewed as a convenient tool for testing new grammatical ideas, or for implementing reasonably complex grammars for particular applications.

G Learn Prolog Now - Cut and Negation

Cut `!` is a goal that always succeeds. It commits Prolog to any choices that were made since the parent goal was unified with the left hand side of the rule.

```
% Potentially inefficiency because the two clauses are mutually exclusive, if the first
    succeeds, the second must fail and viceversa
max(X, Y, Z) :-
    X =< Y,
    Y = Z.

% Attempting to resatisfy this clause is a waste of time
max(X, Y, X) :-
    X > Y.

% The correct version
max(X, Y, Z) :-
    X =< Y,
    !,
    Y = Z.
```

Red cuts are potentially dangerous, if we take them out we do not get an equivalent program. They are indispensable, programs containing them are not fully declarative.

`fail/0` will immediately fail when Prolog encounters it as a goal. It is an instruction to force backtracking.

The cut–fail combination let us define a form of negation called **negation as failure** (to express exceptions).

```
% neg(Goal) succeeds only when Goal does not succeeds
neg(Goal) :-
    Goal,
    !,
    fail.
neg(Goal).
```

The symbol `\+` express negation as failure in Prolog. It can be used only after the variable X has been instantiated.

```
% Vincent enjoys X if X is a burger and X is not a Big Kahuna Burger
enjoys(Vincent, X) :-
    burger(X),
    \+ big_kahuna_burger(X).
```

Negation as failure does not work as logical negation.

For example, to express "*p* holds if *a* and *b* hold, or if *a* does not hold and *c* holds too" we can write either of these.

```
P :- A,
    B.
P :- \+ A,
    C.
```

```
P :- A,
    !,
    B.
P :- C.
```