Logic Programming course

- 3pm Mondays: lectures on Prolog (James Cheney)

- 3pm Thursdays: lectures on theory
Logic Programming course

- 3pm Mondays: lectures on Prolog (James Cheney)
  How to use logic as a programming language.

- 3pm Thursdays: lectures on theory
Logic Programming course

- 3pm Mondays: lectures on Prolog (James Cheney)
  How to use logic as a programming language.

- 3pm Thursdays: lectures on theory
  Why logic can be used as a programming language, and which fragments of logic are suitable for this.
Logic Programming course

- 3pm Mondays: lectures on Prolog (James Cheney)
  How to use logic as a programming language.

- 3pm Thursdays: lectures on theory
  Why logic can be used as a programming language, and which fragments of logic are suitable for this.

The theory lectures provide (mathematical) *explanation* for logic programming, complementary to the *practical experience* provided by the Prolog lectures.
Declarative programming

- Logic programming is a *declarative* style of programming.
Logic programming is a *declarative* style of programming. The programmer says *what* they want to compute, but does not explicitly specify *how* to compute it. It is up to the interpreter (compiler/implementation) to figure out how to perform the computation requested.
Declarative programming

▶ Logic programming is a *declarative* style of programming. The programmer says *what* they want to compute, but does not explicitly specify *how* to compute it. It is up to the interpreter (compiler/implementation) to figure out how to perform the computation requested.

▶ In contrast, in a *procedural* style of programming, the program explicitly describes the individual steps of computation.
Declarative programming

- Logic programming is a *declarative* style of programming. The programmer says *what* they want to compute, but does not explicitly specify *how* to compute it.

  It is up to the interpreter (compiler/implementation) to figure out how to perform the computation requested.

  **Examples:** Logic programming (Prolog), database query languages (SQL), functional programming (Haskell)

- In contrast, in a *procedural* style of programming, the program explicitly describes the individual steps of computation.

  **Examples:** Imperative programming (C), object-oriented programming (Java)
Logic programming — idealistically

- A *logic program* is given as a collection of assumed properties (stated as logical formulas) about the world (or rather about the world of the program)
Logic programming — idealistically

- A *logic program* is given as a collection of assumed properties (stated as logical formulas) about the world (or rather about the world of the program)
- The user supplies a logical formula stating a property that might or might not hold in the world as a *query*
A logic program is given as a collection of assumed properties (stated as logical formulas) about the world (or rather about the world of the program).

The user supplies a logical formula stating a property that might or might not hold in the world as a query.

The system determines whether the queried property is a consequence of the assumed properties in the program.
Logic programming — idealistically

► A *logic program* is given as a collection of assumed properties (stated as logical formulas) about the world (or rather about the world of the program)

► The user supplies a logical formula stating a property that might or might not hold in the world as a *query*

► The system determines whether the queried property is a consequence of the assumed properties in the program.

One declarative aspect is that the user does not specify the method by which the system determines whether or not the query is a consequence of the program.

Another is that whether or not the query is indeed a consequence is independent of the method chosen by the system.
Example logic program

    chicago  →  windy
    edinburgh  →  windy
    edinburgh  →  scotland
    scotland  →  rainy
    windy ∧ rainy  →  insideOutUmbrella
    edinburgh
Example query

Does insideOutUmbrella hold?
Let’s try this in Sicstus Prolog

Program:

windy :- chicago.
windy :- edinburgh.
scotland :- edinburgh.
rainy :- scotland.
insideOutUmbrella :- windy, rainy.
edinburgh.

Query:

| ?- insideOutUmbrella.
Let’s try this in Sicstus Prolog

Program:

```
windy :- chicago.
windy :- edinburgh.
scotland :- edinburgh.
rainy :- scotland.
insideOutUmbrella :- windy, rainy.
edinburgh.
```

Query:

```
| ?- insideOutUmbrella.
! Existence error in user:chicago/0
! procedure user:chicago/0 does not exist
! goal: user:chicago/0
```
Slightly modified Prolog code

Program:

windy :- chicago.
windy :- edinburgh.
scotland :- edinburgh.
rainy :- scotland.
insideOutUmbrella :- windy, rainy.
edinburgh.
chicago :- false.

Query:

| ?- insideOutUmbrella.
yes
Remainder of today’s lecture

- Why is this the correct answer?

- How Prolog computes the answer
Remainder of today’s lecture

- Why is this the correct answer? *(Logical consequence)*
- How Prolog computes the answer
Remainder of today’s lecture

- Why is this the correct answer?  
  *Logical consequence*

- How Prolog computes the answer  
  *Proof search*
Why is this the correct answer?
(Logical consequence)

How Prolog computes the answer
(Proof search)

Today, we restrict attention to propositional logic
Propositional logic (recap)

(Recall notes on logic from Inf 1 - Computation and Logic. Alternatively use on-line references (e.g., Wikipedia).)

Grammar of formulas:

\[ \text{form} ::= \text{atom} \mid \neg \text{form} \mid \text{form} \land \text{form} \mid \text{form} \lor \text{form} \mid \text{form} \rightarrow \text{form} \]

The formulas in our example logic program all have very simple structure. (We shall see later this is no coincidence.)

An example of a more complex propositional formula:

\[ \text{scotland} \land \neg \text{windy} \rightarrow (\text{glasgow} \lor \text{perth}) \]
Logical consequence

An *interpretation* is a function assigning truth values to atoms.

A formula \( F \) is *true* under an interpretation \( \mathcal{I} \), notation

\[
\mathcal{I} \models F,
\]

iff the truth value of the formula comes out as true using the standard truth tables.

A formula \( G \) is said to be a *logical consequence* of formulas \( F_1, F_2, \ldots, F_n \), notation

\[
F_1, \ldots, F_n \models G,
\]

iff, for all interpretations \( \mathcal{I} \),

\[
\text{if } \mathcal{I} \models F_1 \text{ and } \ldots \text{ and } \mathcal{I} \models F_n \text{ then } \mathcal{I} \models G.
\]
Examples

A logical consequence:

\[ \text{poor} \rightarrow \text{happy}, \neg \text{happy} \models \neg \text{poor} \]

We look at all (four!) interpretations. Every time both formulas on the left are true, so is the formula on the right.

A non consequence:

\[ \text{poor} \rightarrow \text{happy}, \neg \text{poor} \not\models \neg \text{happy} \]

The interpretation that assigns ‘false’ to poor and ‘true’ to happy makes both formulas on the left true, but \( \neg \text{happy} \) is false.
Idea of a (propositional) logic program

A logic program is given by a list of (propositional) formulas

\[ F_1, F_2, \ldots, F_n \]

A goal is given by another formula

\[ G \]

The task of the system is to determine whether

\[ F_1, \ldots, F_n \models G \]

If so, the system returns ‘yes’. Otherwise the system returns ‘no’.
Checking logical consequence semantically

In principle, the system can check the logical consequence by constructing a truth table, with one row for every possible interpretation.

When the program and query contain $n$ different atoms, the resulting truth table will have $2^n$ rows.

This method is so computationally expensive as to be infeasible. (Exponential time.)
In principle, the system can check the logical consequence by constructing a truth table, with one row for every possible interpretation.

When the program and query contain \( n \) different atoms, the resulting truth table will have \( 2^n \) rows.

This method is so computationally expensive as to be infeasible. (Exponential time.)

**Million dollar open question.**

Is it possible to find a better method of deciding propositional logical consequence that works in time polynomially bounded in the size of the program and query?
Checking logical consequence semantically

In principle, the system can check the logical consequence by constructing a truth table, with one row for every possible interpretation.

When the program and query contain $n$ different atoms, the resulting truth table will have $2^n$ rows.

This method is so computationally expensive as to be infeasible. (Exponential time.)

 Million dollar open question.
 Is it possible to find a better method of deciding propositional logical consequence that works in time polynomially bounded in the size of the program and query?

Equivalently, does $P = NP$ hold?
(Clay Mathematics Institute — Millenium Prize Problems)
Route to feasibility

- Restrict formulas in logic programs to *definite clauses*
- Use *proof search* to determine logical consequence.
  (That is, use syntactic methods rather than the semantic method of checking truth tables.)

This methodology is effective and flexible.

In later lectures we shall see that it extends to predicate (i.e., first-order) logic (and beyond!)

(Indeed Prolog implements predicate logic, not only propositional logic.)
Propositional definite clauses

A *definite clause* is a formula of one of the two shapes below

\[ q \] (a Prolog *fact* \( q \).

\[ p_1 \land \cdots \land p_k \rightarrow q \] (a Prolog *rule* \( q :- p_1, \ldots, p_k \).

where \( p_1, \ldots, p_k, q \) are all *atoms* (that is, atomic statements).

A *logic program* is a list \( F_1, \ldots, F_n \) of definite clauses

A *goal* is a list \( g_1, \ldots, g_m \) of atoms.

The job of the system is to ascertain whether the logical consequence below holds.

\[ F_1, \ldots, F_n \models g_1 \land \cdots \land g_m. \]
An inference system (or proof system)

The definite clauses $F_1, \ldots, F_n$ in the program are taken as axioms.

We use a single inference rule to derive new facts from facts already established

\[
\frac{p_1 \quad p_2 \quad \ldots \quad p_k \quad p_1 \land p_2 \land \cdots \land p_k \rightarrow q}{q} \quad \text{(Backchain)}
\]

A derivation (or proof) of an individual goal, $g_i$, is a tree of applications of the Backchain rule, in which every leaf is one of the definite clauses $F_1, \ldots, F_n$ comprising the program, and the root of the tree is $g_i$.

A derivation of the compound goal $g_1, \ldots, g_m$, is provided by a list of $m$ derivations, one for each individual goal $g_1, \ldots, g_m$. 
Example derivation

\[
\begin{align*}
\text{edin} & \quad \text{edin} \rightarrow \text{scot} \\
\text{edin} & \quad \text{edin} \rightarrow \text{wind} \\
\text{scot} & \quad \text{scot} \rightarrow \text{rain} \\
\text{wind} & \quad \text{rain} \\
\text{wind} \land \text{rain} & \rightarrow \text{iOU} \\
\text{insideOutUmbrella} & 
\end{align*}
\]
We can also write derivations in a linearized form

1. edinburgh
2. edinburgh $\rightarrow$ windy
3. windy
4. edinburgh $\rightarrow$ scotland
5. scotland
6. scotland $\rightarrow$ rainy
7. rainy
8. windy $\wedge$ rainy $\rightarrow$ insideOutUmbrella
9. insideOutUmbrella
Constructing a derivation

Derivations can be constructed in two directions.

- **Bottom up**: To prove a goal $g$, start from the axioms and apply the inference rules until $g$ is reached.

- **Top down**: To prove $g$, apply the inference rules backwards, starting from $g$, until a set of axioms is found.

Counterintuitively, “bottom up” starts at the top of the paper and proceeds downwards, whereas “top down” starts at the bottom and builds upwards. The terminology “top down” and “bottom up” is motivated by the approach to problem solving rather than the direction the process takes on paper.

It makes no difference to the derivation how it is put together. However, the “top down” approach is (usually) the one best suited to searching for a derivation.
Constructing a derivation

Derivations can be constructed in two directions.

▶ **Bottom up:** To prove a goal \( g \), start from the axioms and apply the inference rules until \( g \) is reached.

▶ **Top down:** To prove \( g \), apply the inference rules backwards, starting from \( g \), until a set of axioms is found.

▶ Counterintuitively, “bottom up” starts at the top of the paper and proceeds downwards, whereas “top down” starts at the bottom and builds upwards. The terminology “top down” and “bottom up” is motivated by the approach to problem solving rather than the direction the process takes on paper.
Constructing a derivation

Derivations can be constructed in two directions.

- **Bottom up**: To prove a goal \( g \), start from the axioms and apply the inference rules until \( g \) is reached.

- **Top down**: To prove \( g \), apply the inference rules backwards, starting from \( g \), until a set of axioms is found.

Counterintuitively, “bottom up” starts at the top of the paper and proceeds downwards, whereas “top down” starts at the bottom and builds upwards. The terminology “top down” and “bottom up” is motivated by the approach to problem solving rather than the direction the process takes on paper.

- It makes no difference to the derivation how it is put together. However the “top down” approach is (usually) the one best suited to searching for a derivation.
A proof search (or inference) procedure is an algorithm that searches for a derivation of a goal from a given list of axioms.

We describe (propositional) Prolog proof search, which is a top-down search procedure.

At every stage in the search, we have an associated list of atoms: the current goal.

The search starts with the goal \([g_1, \ldots, g_m]\), given by the user to the system, as the initial (current) goal.

We illustrate the procedure, using our running example. The initial goal is thus [insideOutUmbrella].
Current goal: $\text{insideOutUmbrella}$.

The first atom in the goal, $\text{insideOutUmbrella}$, is not an axiom.

The only rule in the program that can be used to derive $\text{insideOutUmbrella}$ is

$$\text{windy} \land \text{rainy} \rightarrow \text{insideOutUmbrella}$$

Replace the current goal with

$$[\text{windy}, \text{rainy}]$$
Current goal: [windy, rainy]

The first atom in the goal, \textit{windy}, is not an axiom. The rules in the program that might be used to derive \textit{windy} are:

\begin{align*}
\text{chicago} \rightarrow & \text{windy} \\
\text{edinburgh} \rightarrow & \text{windy}
\end{align*}

Try the first rule first, so replace the current goal with

[\text{chicago, rainy}]
Current goal: [chicago, rainy]

The first atom in the goal, chicago, is not an axiom. There are no rules in the program that can be used to derive chicago.

(This is the point at which Sicstus Prolog complains with the original Prolog formulation of the example. The error message states that no rule is found with chicago as conclusion.)

We thus backtrack to the last point at which there remains an unexplored search option.

We thus restore the current goal to 

[windy, rainy]
Current goal: [windy, rainy]

We have already seen that the first atom in the goal, windy, is not an axiom, and that the first matching rule

\[ \text{chicago} \rightarrow \text{windy} \]

does not produce a derivation of windy.

We thus try the next rule in the program that matches windy:

\[ \text{edinburgh} \rightarrow \text{windy} \]

So replace the current goal with

[edinburgh, rainy]
Current goal: [edinburgh, rainy]

The first atom in the goal, edinburgh, is an axiom.

We thus remove edinburgh from the goal list.

So replace the current goal with

[rainy]
Continuing the same way, we obtain successively:

Current goal: [rainy]

Current goal: [scotland]

Current goal: [edinburgh]

Current goal: []

Now the search has produced an empty list for the current goal. This means the search for a derivation has been successful.

The Prolog search thus returns the answer yes to the query insideOutUmbrella
Main points today

propositional logic
logical consequence
definite clauses
inference systems
propositional Backchain rule
propositional Prolog proof search procedure