Declarative programming

- Logic programming is a *declarative* style of programming.

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.

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*

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.

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.

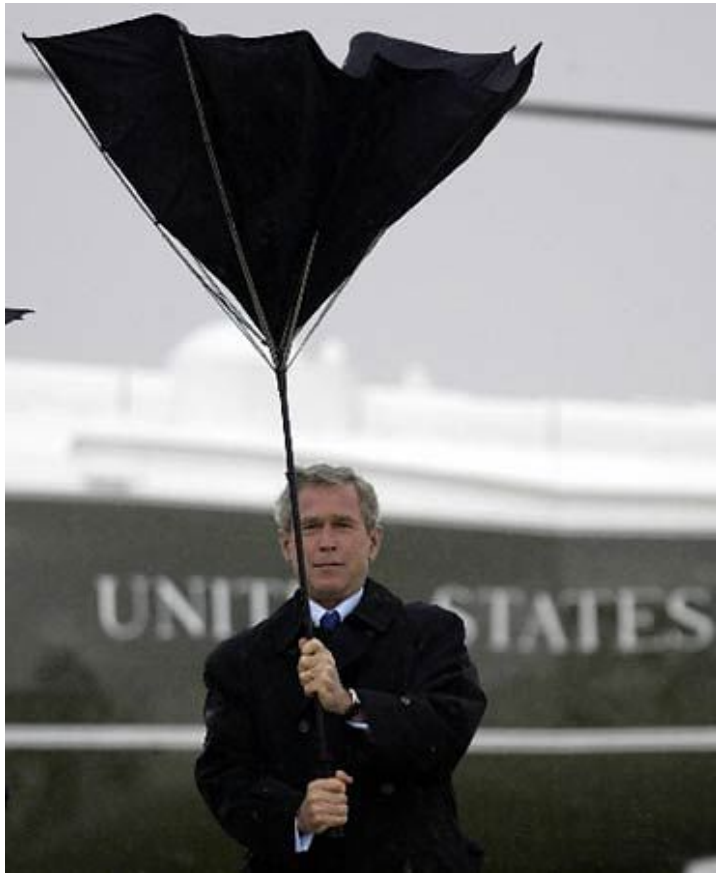
  Another is that whether or not the query is indeed a consequence is independent of the method chosen by the system.

Example logic program

$$
\begin{aligned}
\texttt{chicago} &\rightarrow \texttt{windy} \\
\texttt{edinburgh} &\rightarrow \texttt{windy} \\
\texttt{edinburgh} &\rightarrow \texttt{scotland} \\
\texttt{scotland} &\rightarrow \texttt{rainy} \\
\texttt{windy} \wedge \texttt{rainy} &\rightarrow \texttt{insideOutUmbrella} \\
&\quad\ \texttt{edinburgh}
\end{aligned}
$$

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 Sicstus Prolog code

Program:

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

Query:

```
| ?- insideOutUmbrella.
yes
```

We avoid this quirk, by working with idealized Prolog

Program:

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

Query:

```
| ?- insideOutUmbrella.
yes
```

Key points to address

- Why is this the correct answer?
- How Prolog computes the answer

Key points to address

- Why is this the correct answer?
  (Logical consequence) — today's lecture
- How Prolog computes the answer

Key points to address

- Why is this the correct answer?
  (Logical consequence) — today's lecture
- How Prolog computes the answer
  (Proof search) — Lecture 2

Key points to address

- Why is this the correct answer?
  (Logical consequence) — today's lecture
- How Prolog computes the answer
  (Proof search) — Lecture 2
- But Prolog does not always find the correct answer
  (Incompleteness of Prolog's search procedure) — Lecture 2

Key points to address

- Why is this the correct answer?
  (Logical consequence) — today's lecture
- How Prolog computes the answer
  (Proof search) — Lecture 2
- But Prolog does not always find the correct answer
  (Incompleteness of Prolog's search procedure) — Lecture 2

  In Lectures 1–2 we restrict attention to *propositional logic*.

Key points to address

- Why is this the correct answer?
  (Logical consequence) — today's lecture
- How Prolog computes the answer
  (Proof search) — Lecture 2
- But Prolog does not always find the correct answer
  (Incompleteness of Prolog's search procedure) — Lecture 2

  In Lectures 1–2 we restrict attention to *propositional logic*.
  From Lecture 3 we shall look at *predicate 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:

$$\mathit{form} \ ::= \ \mathit{atom} \mid \neg \mathit{form} \mid \mathit{form} \wedge \mathit{form} \mid \mathit{form} \vee \mathit{form} \mid \mathit{form} \rightarrow \mathit{form}$$

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:

$$\texttt{scotland} \wedge \neg \texttt{windy} \ \rightarrow \ \texttt{glasgow} \vee \texttt{perth}$$

Interpretations

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

For example, if the atoms are $\texttt{poor}, \texttt{happy}$ then

$$\{\texttt{poor} \mapsto \textbf{false}, \texttt{happy} \mapsto \textbf{true}\}$$

is an interpretation.

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.

For example,

$$\{\texttt{poor} \mapsto \textbf{false}, \texttt{happy} \mapsto \textbf{true}\} \models \texttt{poor} \rightarrow \texttt{happy}$$

Logical consequence

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 $\mathcal{I}$,

$$\text{if } \mathcal{I} \models F_1 \text{ and } \ldots \text{ and } \mathcal{I} \models F_n \text{ then } \mathcal{I} \models G \ .$$

Don't get confused! The symbol $\models$ is used in two different ways:

$$\mathcal{I} \models F$$
$$F_1, \ldots, F_n \models G$$

In the first the left-hand-side is an interpretation, in the second it is a sequence (or set) of formulas.

Examples

A logical consequence:

$$\texttt{poor} \rightarrow \texttt{happy}, \neg\texttt{happy} \models \neg\texttt{poor}$$

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:

$$\texttt{poor} \rightarrow \texttt{happy}, \neg\texttt{poor} \not\models \neg\texttt{happy}$$

The interpretation that assigns 'false' to `poor` and 'true' to `happy` makes both formulas on the left true, but $\neg$`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 rows.

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.)

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?

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

$$q \qquad \text{(a Prolog } \textit{fact}\ q\,.\ \text{)}$$

$$p_1 \wedge \cdots \wedge p_k \ \rightarrow q \qquad \text{(a Prolog } \textit{rule}\ q\ \texttt{:-}\ p_1, \ldots, p_k.\text{)}$$

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.

$$F_1, \ldots, F_n \models g_1 \wedge \cdots \wedge g_m \ .$$

Main points today

propositional logic

logical consequence

definite clauses

Next time

inference systems

propositional Prolog proof search procedure

properties of Prolog proof search