Lecture 02

Declarative Programming

An Introduction to Prolog

Jeremy Pitt
Department of Electrical & Electronic Engineering
Imperial College London, UK
Email: j.pitt _at_ imperial _dot_ ac _dot_ uk
Phone: (int) 46318
Office: 1010 EEE

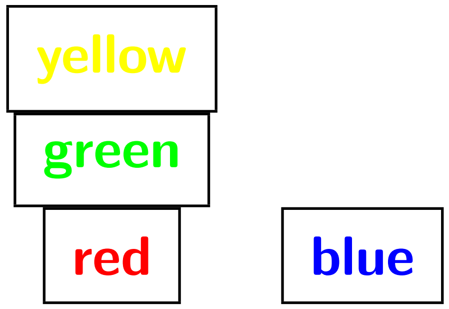
Aims and Objectives

- Aims
  - An introduction to the declarative programming style, and the programming language Prolog
  - An overview of knowledge representation in logical form
  - The specification, implementation and execution of algorithms for ‘machine intelligence’
- Objectives
  - Able to ‘read’ declarative specifications
  - Able to write declarative programs

Contents

Sample Domain

- The Blocks World



- What 'sort of thing' are we interested in (declaratively) speaking?
  - That which is 'true', or which can be shown to be true. This includes
    - Facts concerning 'on'-ness
    - Relations concerning 'tower'-ness, or 'above'-ness
    - (And an ontological commitment to the objects in the domain)

- So simple, yet it exhibits many of the problems of 'commonsense reasoning' we take for granted

Procedural vs. Declarative

- Procedural Programming
  - Specify the steps in a process (how)
  - **run** the program
- Declarative Programming
  - Specify the facts/relations that hold (what)
  - The relation might be **recursive** – defined in terms of itself
  - **query** the program
    - the query is also called a **goal**

Example

- Towers of a certain height
  - Procedural approach – build a tower of height $n (n \geq 1)$ **!**

```
∗ pick a free block $b$ on the table; $h = 1; top = b$;
∗ while $(h < n)$ {
      pick a free block $b'$ such that $b' \mathrel{!=} top$;
      stack $b'$ on $top$;
      $h = h + 1; top = b'$
  }
```

  - Declarative approach – is there a tower of height n on some base b **?**
    - ∗ there is a tower of height 1, with b as the base, if

      Some a is on b, and a is clear;
    - ∗ there is a tower of height n, with b as the base, if

      Some a is on b, and

      There is a tower of height $n - 1$, with a as the base

Declarative Specification

- Specification in a declarative language
  - In pseudo-English
    - There is a table, and blocks red, green, blue and yellow
    - X is on Y, if

      X is green and Y is red, OR

      X is red and Y is the-table
    - X is above Y, if

      X is on Y, OR

      X is on Z, and Z is above Y
  - In Predicate Logic
    - Objects (constants, terms)
      - things with individual identities and properties, for example, tables, or blocks
    - Properties and relations
      - things that ‘hold’ (are true) of and between objects
    - Formulas: sentences on properties/relations about objects
      - atomic sentences: facts
      - complex sentences: using connectives and quantifiers

Declarative Specification, in Prolog

- Objects are still terms
  - Those things we commit to our exist in our micro-world
  - Don't have to declare them; don't even have to give them a type
  - Simple terms: constants and variables; complex terms: lists, tuples, compound terms

- Sentences (atomic and complex) are clauses (facts and rules)
  - Clauses specify relations (boolean-valued functions) about the objects

```
• on( green, red ).
• on( yellow, green ).
• on( red, table ).
• on( blue, table ).
```

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

```
• tower( 1, B ) :-
      on( A, B ),
      \+ on( _, A ).
• tower( N1, B ) :-
      on( A, B ),
      N is N1 - 1,
      tower( N, A ).
```

Example (1)

- The specification is its own implementation – i.e. it is a program
- But we *don't run* the program, we *query* the program
- on
  - ?- on(green, red).
  - ?- on(green, yellow).
  - ?- on(green, Y).
  - ?- on(X, red).
  - ?- on(X, table).
  - ?- on(X, Y).
- When "computer says 'yes' ", it gives which values of variables make the query true (and the inference engine "remembers" how far it got in the computation of proving that it was true)

Example (2)

- above
  - ?- above(yellow, red).
  - ?- on(yellow, red). – fails, so try the second clause
  - ?- on(yellow, Z), above(Z, red).
  - ?- yellow is on green, so on(yellow, Z) succeeds, with Z = green
  - ?- above(green, red).
  - ?- on(green red).
  - ?- yes.

- tower
  - ?- tower(3, table).
  - 3 does not match with 1, so try the second clause (N1=3, B=table)
  - ?- on(A, table), N is 3 - 1, tower(N, Y).
  - matches with on(red, table), so succeeds with A=red
  - ?- N is 3 - 1, so succeeds with N=2
  - ?- tower(2, red)
  - and? . . .

Simple Terms: Constants

- Atoms
  - Constants (notational convention: these start with a lower case letter, followed by zero or more alphanumeric characters)
    - red, blue, green, table, bart, lisa, homer, marge
  - Symbolic: sequence of symbolic characters
    - $+$, $=>$, $>=$, $:-$

- Numbers
  - Integers
  - Floating point

Simple Terms: Variables

- Start with an upper case letter, followed by by alphanumeric sequence of characters, including underscore (_)
  - X, X1, X2, Y, Jeremy, Magic_Dragon, Jolly_Green_Giant
- Variables can take the value of (**be instantiated to**) any term, simple or complex
  - But get this: once a variable takes a value, it can never change (it can only be 'undone')
  - There is **no assignment**
  - There is **no assignment**
- Prolog 'variables' are **not** variable: they are just 'things' we don't the value of . . . yet

- Scope of a variable
  - Within one sentence, all occurrences of the same variable name refer to the same term
    - So once one occurrence of a variable takes a value, they all take the same value
  - Between (any) two sentences, the same variable names do not refer to the same thing

- The anonymous variable
  - Written as an underscore on its own: _
  - Used when you don't care what value of a variable makes a statement true
  - Multiple occurrence of the anonymous variable within the same statements *can* have different values
    - middleblock(B) :- on(_, B), on(B, _).

Complex Terms (1): Compound Terms

- Complex terms are used to represent structured/related information

- A compound term is made up of:
  - an atom, called the **functor**
  - followed by $k \geq 1$ terms
    - enclosed in brackets – ‘(’ ‘)
    - separated by commas
  - k denotes the **arity** of the term (the functor has k arguments)
  - Examples
    - course(ai, fall, e316, pitt)
    - lecturer(pitt, ee1010, 46318, ’j.pitt@imperial.ac.uk’)
    - on(X, Y), above(X, Y), tower(2, red)

Clauses and Programs

- Strict syntax: a **clause** consists of
  - a **head**
    - either an atom or a compound term
  - a possibly empty **body**
    - if non-empty, it consists of **:-** followed by $n >= 1$ comma-separated terms
    - each term in the body can be an atom or a compound term
  - and terminated with a full-stop '**.**'
- A clause without a body is a fact
- The variables in a clause are implicitly universally quantified and restricted in scope.
- A set of clauses is a **program**

Execution

- A query (goal) is also has the same form as a compound term

- To answer a query, we have to **match** two terms: the goal with the head of a clause
  - Two atoms match if they are identical
  - A variable matches with any term, even another variable
  - Two compound terms match if they are identical, or can be made identical by instantiating the variables
    - ∗ The functor is the same
    - ∗ The arity is the same
    - ∗ Each pairwise terms match
  - The same variable has to be instantiated with the same value throughout a match (not the anonymous variable)
    - ∗ on(X, X) will not match with on(red, green)
  - This matching process is called **unification** (see later)

- If a goal matches with the head of a clause
  - The variable instantiations are passed to the body of the clause
  - The body of the clause becomes the new goal . . . and repeat . . .
- What this means (logically)
  - The head of a clause is provably true, if . . .
  - . . . the conjunction of all the terms in the body is provably true
- Notes
  - Prolog will try to prove a goal, as a conjunction of sub-goals, starting from the leftmost conjunct
  - Prolog will try to match the goal with the first clause head it can find in the program
  - If Prolog fails to prove a sub-goal, it will **backtrack** and try to find alternative way of proving the previous sub-goal

! (Cut)

- For controlling execution
- Sometimes, you get so far in a proof, you don't want Prolog to look for any more alternatives
- Use '**!**' (pronounced 'cut')
- This freezes the proof of the current goal at this point
  - if the proof ever backtracks to here, Prolog will not re-try any previous subgoals in the body or any later clauses in the program, and fails this goal immediately
- Often needed when dealing with lists

Complex Terms: Lists

- List notation
  - sequence of $k \geq 0$ terms, separated by commas, enclosed in square brackets – ‘**[**’ ‘**]**’
  - square brackets with nothing in them – [ ] – denotes the empty list

- List matching
  - the notation [H | T] matches with a list such that
    - $*$ H is matched with the first *element* of the list
    - $*$ T is matched with the rest of the list
    - $*$ so H is an element and T is a list
  - The first n elements of a list can be explicitly listed using comma separated terms: [H1, H2, . . ., Hn | T]
    - $*$ Each of the Hi $(1 \leq i \leq n)$ is matched with ith *element* of the list
    - $*$ T is still matched with the rest of the list (the $n + 1$th elements to the end of the list)

Lists: An Example

- What does this represent?
  - [[blue], [yellow, green red]]
  - An alternative representation of the blocks world's towers using lists rather than predicates
    - Note: the table is not explicitly represented
    - The last element of each list is assumed to be on the table

- What does this do?

  -
    ```
    guess( [ ], [ ] ).
    guess( [Tower | T1], [B | T2] ) :-
        last( Tower, B ),
        guess( T1, T2 ).
    ```

- This pattern (base case, process head, recurse on tail) is very common in list processing

Complex Terms: Tuples

- Tuple notation
  - sequence of $k \geq 1$ terms, separated by commas, enclosed in round brackets – ‘**(**’ ‘**)**’
- A collection of related information
- E.g. complicating the blocks world: representing blocks by colour and size
  - (green, 3), (red, 2), (blue, 7)

DIY

- Making it up as you go along
- Use own defined **operators**
  - Operators have a precedence and associativity
  - Define boolean connectives (and, or, etc.) as follows
  - `op( 700, xfy, <-> ), op( 650, xfy, -> ),`
    `op( 600, xfy, + ), op( 500, xfy, & ), op( 40, fy, - ).`
  - Terms will then be constructed properly, i.e.:

$\neg p \wedge q \vee r \rightarrow s \leftrightarrow t$ is parsed as $((((\neg p) \wedge q) \vee r) \rightarrow) s \leftrightarrow t$

$\neg p \leftrightarrow q \rightarrow r \vee s \wedge t$ is parsed as $(\neg p) \leftrightarrow (q \rightarrow (r \vee (s \wedge t)))$

Meta-statements: Negation as Failure

- Prolog answering "yes" to a query means it is provably true
- Prolog answering "no" therefore only means it is not provably true, at least not with information provided in the knowledge base using the clauses in the program
- The **closed world assumption**: anything not provably true, is actually false
- The '\+' operator
  - ?- \+ Goal. succeeds, if Goal fails
  - clear(X) :-

    \+ on(_, X).
- You can't apply \+ to facts, or to the heads of rules (they are not goals)

Meta-Statements: findall

- Sometimes, want all the answers that make a goal succeed in one go, rather than generating them one by one by backtracking

- findall
  - ?- findall(X, Goal, S).
    - X is a variable
    - Goal explicitly mentions X
    - S is instantiated to a list of values for X that make Goal succeed

- Examples
  - ?- findall(X, on(X, table), S).
    S = [blue, red]
  - ?- findall(C, father(homer, C), S).
    S = [bart, lisa, maggie]

Other Fun Stuff

- Arithmetic operations
- Input and output
- $=$, and $==$
- Debugging

Summary

- Introduction to the basics of declarative programming in Prolog
- The logical foundations will be presented later in the course
- This should be sufficient to understand the specification of algorithms in subsequent lectures and to write your own programs in labs