Prolog - introductory comments

[source: J. R. Fisher's tutorial, www.csupomona.edu/~jrfisher/www/prolog_tutorial]

- a *logical* and *declarative* programming language
- short for PROgramming in LOGic
- its heritage: 1960's and 1970's theorem-prover and automated-deduction research
- its inference mechanism is based upon Robinson's resolution principle (1965) together with mechanisms for extracting answers proposed by Green (1968).
- the "first" Prolog was "Marseille Prolog" based on work by Colmerauer (1970).

what is declarative programming?

[source: Wikipedia, "Declarative Programming", http://en.wikipedia.org/wiki/Declarative_programming]

- in declarative programming, you express "the *logic* of a computation without describing its control flow";
- ...that is, you describe "*what* the program should accomplish, rather than describing *how* to go about accomplishing it";
- ("this is in contrast with imperative programming, which requires an explicitly provided algorithm")

what is logic programming?

[source: Wikipedia, "Logic Programming", http://en.wikipedia.org/wiki/Logic_programming]

- "in its **broadest** sense ... [it is] the use of mathematical logic for computer programming."
- "in the **narrower** sense in which it is more **commonly** understood, [it] is the use of logic as both a declarative and procedural representation language."
- "it is based upon the fact that a **backwards reasoning theorem-prover** applied to declarative sentences in the form of implications [can treat] the implications as goal-reduction procedures"
- ...as we'll see in Prolog;

uses of Prolog

[source: Wikipedia, http://en.wikipedia.org/wiki/Prolog]

- designed for natural language processing
- has been used in a variety of other areas as well, including:
  - theorem proving
  - expert systems
  - games
  - automated answering systems
  - ontologies
  - sophisticated control systems
- "...modern Prolog environments support creation of grgraphical user interfaces, as well as administrative and networked applications."

SWI-Prolog

- the version of Prolog we will be using in this course
- available for free from http://www.swi-prolog.org/
- has versions for Windows, Mac, Linux
- interesting buzzwords from its installation window:
  - "...an open source ISO/Edinburgh-style Prolog compiler including modules, ... libraries, garbage-collector,...C/C++-interface, multiple threads, GNU-readline interface, coroutining, constraint programming, global variables, very fast compiler. Including packages clib (Unix process control, sockets...), cpp (C++ interface), sgml (reading XML...), ...ODBC interface & XPCE (Graphics UI toolkit, integrated editor (Emacs-clone) and graphical debugger)."

SWI-Prolog - starting and stopping

- command-line interface
- (installed in `/opt/local/bin` when I installed on Mac OS X in Spring 2010)
- ...since that's in my path, then typing: `swipl`

  ...in a Terminal window starts it up;
- According to the SWI-Prolog manual, for Windows:
  - "Opening a `.pl` file will cause `swipl-win.exe` to start, change directory to the directory in which the file-to-open resides and load this file."
- to quit: type `halt.` at the prompt...

SWI-Prolog - example 1 (2 pgs)

```
Macintosh-194:~ smtuttle$ swipl

% library(swi_hooks) compiled into
pce_swi_hooks 0.00 sec, 3,688 bytes

Welcome to SWI-Prolog (Multi-threaded, 64
bits, Version 5.8.3)

Copyright (c) 1990-2009 University of
Amsterdam.

SWI-Prolog comes with ABSOLUTELY NO
WARRANTY. This is free software,

and you are welcome to redistribute it
under certain conditions.
```

```
Please visit http://www.swi-prolog.org for
details.

For help, use ?- help(Topic). or ?-
apropos(Word).

?- halt.
Macintosh-194:~ smtuttle$
```

Logic Programming Concepts - part 1

[source: Scott, "Programming Language Pragmatics III", Ch. 11, p. 546]

- "Logic programming systems allow the programmer to state a collection of **axioms** from which theorems can be proven."
- "The user of a logic program states a theorem, or **goal**, and the language implementation attempts to find a collection of axioms and inference steps (including choices of values for variables) that together imply that goal."

Logic Programming Concepts - part 2

[source: Scott, Ch. 11, p. 546]

- "In almost all logic languages [including Prolog], axioms are written in a standard form known as a **Horn clause**.
  - A Horn clause consists of a **head**, or **consequent** term *H*, and a **body** consisting of terms B_i:

    H <-- $B_1, B_2, \ldots, B_n$

  - The semantics of this statement are that when the B_i are all true, we can deduce that *H* is true as well.
  - When reading aloud, we say, "*H*, if B_1, B_2, ..., and B_n."
- Horn clauses can be used to capture most, but not all, logical statements."

Resolution

[source: Scott, Ch. 11, p. 546]

- "...to derive new statements, a logic programming system combines existing statements, canceling like terms, through a process known as **resolution**."
- EXAMPLE:
  - If we know that A and B imply C,
  - and that C implies D,
  - we can deduce that A and B imply D:

```
C <-- A, B

D <-- C

-----------------

D <-- A, B
```

Unification

[source: Scott, Ch. 11, p. 546]

- To add power to this, "In general, terms like A, B, C, and D may consist not only of constants ("Arcata is rainy"), but also of predicates applied to atoms or to variables:

```
rainy(Rochester),
rainy(Arcata), rainy(X)
```

- During resolution, free variables may acquire values through unification with expressions in matching terms

```
flowery(X) <-- rainy(X)

rainy(Arcata)

----------------------------

flowery(Arcata)
```

Prolog specifics, part 1

[source: Scott, Ch. 11, pp. 547-548]

- "...a Prolog interpreter runs in the context of a **database** of **clauses (Horn clauses**) that are assumed to be true."
- "Each clause is composed of **terms**, which may be **constants**, **variables**, or **structures**."
  - "A **constant** is either an **atom** or a **number**."
  - A **structure** can be thought of as either a **logical predicate** or a **data structure**."

Prolog specifics: Atoms

[source: Scott, Ch. 11, pp. 547-548]

- "**Atoms** in Prolog are similar to symbols in Lisp.
- "lexically, an atom looks like:
  - an identifier beginning with a lowercase letter,
  - a sequence of punctuation characters,
  - or a quoted character string
- Examples:

```
foo

my_Const

+

'Hi, Mom'
```

Prolog specifics: Numbers

[source: Scott, Ch. 11, pp. 547-548]

- "Numbers resemble the integers and floating point constants of other programming languages"
- Examples:

```
13

28.007
```

Prolog specifics: Variables

[source: Scott, Ch. 11, pp. 547-548]

- "A **variable** looks like an identifier beginning with an *UPPERCASE* letter:

```
 Foo        My_var        X
```

  - Variables can take be instantiated to (i.e., can take on) arbitrary values at run time as a result of unification.
  - The **scope** of every variable is **limited** to the **clause** in which it appears.
  - There are **no** declarations.
  - As in Lisp, type checking occurs only when a program attempts to use a variable in a particular way at run time.

Prolog specifics: Structures

[source: Scott, Ch. 11, pp. 547-548]

- "**Structures** consist of an atom called the **functor** and a list of arguments:

```
 rainy(arcata)

 teaches(tuttle, cs335)

 bin_tree(foo, bin_tree(bar, arc))
```

  - Prolog requires the opening parenthesis to come IMMEDIATELY after the functor, with NO intervening space;
  - Arguments can be arbitrary terms: constants, variables, or (nested) structures."

Prolog specifics: Structures (cont'd)

[source: Scott, Ch. 11, pp. 547-548]

- "*Internally*, a Prolog implementation can represent a structure using Lisp-like cons-cells;
- *CONCEPTUALLY*, the programmer may prefer to think of certain structures (e.g., `rainy`) as **logical predicates**.
  - We use the term "predicate" to refer to the combination of a functor and an "arity" (number of arguments.
  - The predicate `rainy` has arity 1.
  - The predicate `teaches` has arity 2."

Clauses in a Prolog database

[source: Scott, Ch. 11, pp. 547-548]

- The clauses in a Prolog database can be classified as facts or rules, each of which ends with a PERIOD.
- A **fact** is a Horn clause without a right-hand side.
  - It looks like a single term (the implication symbol is implicit):

```
rainy(arcata).
```

- A **rule** has a RHS:

```
snowy(X) :- rainy(X), cold(X).
```

  - The token `:-` is the implication symbol;
  - The comma indicates "and"
-

Clauses, continued

- Variables that appear in the head of a Horn clause are **universally** quantified:
  - **for all X**, X is snowy if X is rainy and X is cold.
- can also "...write a clause with an empty LEFT-hand-side. Such a clause is called a **query**, or a **goal**.
  - Queries do NOT appear in Prolog programs.
  - Rather, one builds a **database** of facts and rules,
  - and then initiates execution by giving the Prolog interpreter (or the compiled Prolog program) a query to be answered (i.e., a goal to be proven)