Logic programming: Prolog I

- Algorithm = Logic + Control
- Basis for Prolog:
  - Backward chaining with Horn clauses +
  - A few interesting bells & whistles
- Widely used in Europe, Japan (basis of 5th Generation project)
- Compilation techniques $\Rightarrow$ 60 million LIPS (logical inferences per second

Logic programming: Prolog II

- Program = set of clauses = `head :- literal`$_1$`, … literal`$_n$`.`

```
criminal(X) :- american(X), weapon(Y), sells(X,Y,Z),
                          hostile(Z).
```

- Depth-first, left-to-right backward chaining
- Built-in predicates for arithmetic etc., e.g., `X is Y*Z+3`
- Built-in predicates that have side effects (e.g., input and output predicates, assert/retract predicates)
- Closed-world assumption ("negation as failure")
  - e.g., given `alive(X) :- not dead(X).`
  - `alive(joe)` succeeds if `dead(joe)` fails

Logic programming

Remember: knowledge engineering vs. programming...

Sound bite: computation as inference on logical KBs

	Logic programming	Ordinary programming
1.	Identify problem	Identify problem
2.	Assemble information	Assemble information
3.	Tea break	Figure out solution
4.	Encode information in KB	Program solution
5.	Encode problem instance as facts	Encode problem instance as data
6.	Ask queries	Apply program to data
7.	Find false facts	Debug procedural errors

Should be easier to debug $Capital(NewYork, US)$ than $x := x + 2$!

Logic programming systems

e.g., **Prolog:**

- Program = sequence of sentences (implicitly conjoined)
- All variables implicitly universally quantified
- Variables in different sentences considered distinct
- **Horn clause sentences** only (means atomic sentences or sentences with no negated antecedent and atomic consequent)
- Terms: constant symbols, variables or functional terms
- Queries: conjunctions, disjunctions, variables, functional terms
- Instead of negated antecedents, use negation as failure operator: goal NOT P considered proved if system fails to prove P
- Syntactically distinct objects refer to distinct objects
- Many built-in predicates (arithmetic, I/O, etc)

Basic syntax of facts, rules and queries

```
<fact>   ::= <term> .
<rule>   ::= <term> :- <term> .
<query>  ::= <term> .
<term>   ::= <number> | <atom> | <variable>
               | <atom> (<terms>)
<terms>  ::= <term> | <term>, <terms>
```

A PROLOG Program

- A PROLOG program is a set of ***facts*** and ***rules***.
- A simple program with just facts :

```
person(alice).
person(jim).
person(tim).
person(sharon).
person(james).
friend(alice, jim).
friend(jim,   tim).
friend(jim,   dave).
friend(jim,   sharon).
friend(tim,   james).
friend(tim,   thomas).
```

A PROLOG Program

- Similar to a table in a relational database.
- Each line is a ***fact*** (a.k.a. a tuple or a row).
- Each line states that some person `X` is a friend of some (other) person `Y`.
- In GNU PROLOG the program is kept in an ASCII file.

Compiling and Executing

- “gplc” – is a GNU Prolog compiler
- Compile a file “foo.pl” and the result is essentially an executable of name “foo”
- Executing “foo” puts you into a Prolog Interpreter
- Prolog Interpreter: interactively supports query and inputting of additional Prolog statements that you might want to incrementally add

The Prolog Interpreter

- "gprolog" – is a GNU Prolog interpreter
- You can normally just type things into it and get answers.
- To enter your own input interactively, at the prompt enter: *just type it in to the prompt one at a time*
- You can load a pre-existing file "foo.pl" by entering at the prompt: *consult(foo).* ➔ as though you are entering a new rule or fact; this will load foo.pl in

Some Important Implementation Notes

- ON ALUDRA, use version 1.3.0 for both gplc and gprolog
- $ which gplc

/auto/usc/gnu/prolog/1.3.0/bin/gplc

gplc is not working properly (in my experiments using the csci460 class account)

gprolog seems to be working properly

So, use **gprolog** if you want to try things or to do homework # 4

....

A PROLOG Query

- Now we can ask PROLOG questions :

```
| ?- friend(alice, jim).
yes
| ?- friend(jim, herbert).
no
| ?-
```

A PROLOG Query

- Not very exciting yet. But what about this :

```
| ?- friend(alice, Who).
Who = jim
yes
| ?-
```

- “`Who`” is called a ***logical variable***.
  - PROLOG will set a logical variable to any value which makes the query succeed [UNIFICATION]
  - Anything starting with **uppercase** letter is a **variable**

A PROLOG Query II

- Sometimes there is more than one correct answer to a query.
- PROLOG gives the answers one at a time. To get the next answer, the user types "`;`"

```
| ?- friend(jim, Who).
Who = tim ? ;
Who = dave ? ;
Who = sharon ? ;
yes
| ?-
```

NB : The `;` *may* not actually appear on the screen.

- After finding that `jim` was a friend of `sharon` GNU PROLOG detects that there are no more alternatives for `friend` and ends the search.

A Simple PROLOG Rule

- A Prolog **rule** must be written in Horn form:

```
A and B and C => D  is written as:
D :- A, B, C.
, stands for "and"
:- can be read as "when"

acquainted(X, Z) :- friend(X, Y), friend(Y, Z).
States a friend of a friend is an acquaintance.
If X has a friend Y, and Y has a friend Z, then X
  and Z are "acquainted"
```

- Note that this is written in "**reverse implication**" style

A Simple PROLOG Query to Test Condition

- You can now query Prolog in more interesting ways:

```
| ?- acquainted(jim, james).
  acquainted(jim, jam(jim, james).
  true ?
  yes
| ?-
```

- Prolog answers the query "true" or "false" if the query contains no variables.
- Prolog uses backward chaining to decide if the query is true or not

PROLOG Query to Generate Answers

- You can get Prolog to generate answers for you through unification and backward chaining:

```
| ?- acquainted(jim, Who).
  acquainted(jim, Who(jim, Who).
  Who = james ? a
  a
  Who = thomas
  no
| ?-
```

- If the query contains variables (e.g., Who), Prolog generates values one by one
- ; generates queries one by one
- “a” generates all of them
- <return> stops generating values

Another PROLOG Query to Generate Answers

- Similar to the last slide – yet another "generation" query:

```
| ?- acquainted(alice, Who).
  acquainted(alice, W(alice, Who).
  Who = tim ? a
  a
  Who = dave
  Who = sharon
  yes
| ?-
```

- Since the query contains the variable "Who", Prolog generates the values
- Since the user asked for all values (by entering "a") all values are generated

Equality in PROLOG Rules

- Define a popular person:

```
A person is popular if he has 3 (or more) friends:
popular(X) :- friend(X, Y), friend(X, Z),
  friend(X, W), Y \= Z, Y \= W, Z \= W.

In English,
If X has different friends Y, Z, and W, then X is
  popular
In order to ensure that Y, W, and Z unify to
  different values, you have to state the
  “inequality” predicates
```

Querying the “popular” Predicate

- Query the “*popular*” relation:

```
? popular(sharon).
No
? popular(jim).
Yes
```

- You can even ask for popular people (i.e., generate a list of them).

```
 ? popular(Who).
Jim?
Yes
?
```

Negation in PROLOG Rules

- Being in Horn form makes it hard for Prolog to handle negation – recall that Horn Form (in implicative form) to be written in terms of ONLY positive literals
- By "cheating" a little: failure to prove A is considered *not*(A)
- Many Prolog implementations have a built-in negation operation that you can use
- \+ is the equivalent of "not" or negation
- You are allowed to write \+ with clauses in your premise with the understanding that negation of a predicate is simply failure (by Prolog) to show the truth of the predicate

Example PROLOG Rule with Negation

- Define a lonely person:

```
A lonely person has no friends:
lonely(X) :- \+ (person(Y), friend(X, Y)).

| ?- lonely(bill).
  lonely(bill^[[A^[(bill).
  yes
| ?- lonely(jim).
  lonely(jim^[[A^[[(jim).
  no
| ?- lonely(sharon).
  lonely(sharon^[[(sharon).
  yes
| ?- lonely(bil).
  lonely(bil^[[(bil).
  yes
| ?-
```

- Oops – "bil" cannot be lonely because bil is not a person! How did that happen?

Improved PROLOG Rule with Negation

- When Negating, always provide a predicate of some sort to restrict the variables (or you will end up with spurious results).
- Define a lonely person:

```
A lonely person is a person with no friends:
lonely(X) :- person(X), \+ (person(Y), friend(X, Y)).

| ?- lonely(bill).
  lonely(bill^[[A^[(bill).
  yes
| ?- lonely(jim).
  lonely(jim^[[A^[[(jim).
  no
| ?- lonely(sharon).
  lonely(sharon^[[(sharon).
  yes
| ?- lonely(bil).
 lonely(bil^[[A^[[(bil).
 no
| ?-
```

Generating list of “lonely” people

- Query the “*lonely*” relation:

```
| ?- lonely(Who).
  lonely(Who^[[A^[[(Who).
  Who = sharon ? a
  a
  Who = james
  Who = thomas
  Who = bill
  yes
| ?-
```

Tutorial on the Website:

- About locations/geography

```
in_georgia(atlanta).
in_united_states(X) :- in_georgia(X).
in_united_states(georgia).
located_in(usa,north_america).
located_in(X,usa) :- in_united_states(X).
located_in(X,north_america) :- located_in(X,usa).
alsolocated_in(X,Z) :-
  located_in(Y,Z),located_in(X,Y).
```

Prolog List Concept

- [apple, orange, banana, pear] is a Prolog list
- List is a sequence of comma separated items within […]
- [] is the empty list
- [a, [a, b, c], [a, b, c, d], e, f, []] is also a list. It is a List of Lists where the first element is “a”, second is [a, b, c], and so on
- Lists can be nested arbitrarily as you can see

More on Prolog Lists

- Lists are composed from two elements:
  - Head of the list is the first item
  - Tail of the list are the remaining items
  - For the list: [apple, orange, banana, pear] the head is “apple” and tail is [orange, banana, pear]
  - For the list: [[a, b, c], [a, b, c, d], e, f, []] the head is [a, b, c] and the tail is [[a, b, c, d], e, f, []]

The | operator

- Use | operator in a query to pull apart a list. For example:

  [Head | Tail] = [a, b, c, d, e] .

  Prolog will return:

  Head = a

  Tail = [b, c, d, e]

  Yes

Note that Head and Tail are just variables. You can use any other variables that you like

If you do the above on an empty list you will get No

The | operator

- | can be used to provide the first 2 or 3 or n elements if the list is long enough:

[X, Y | Z] = [[], [a, b], c, d, [a, b, c, d]].

X=[]

Y=[a, b]

Z=[c, d, [a, b, c, d]]

So, | can be used to split a list at any point – not just into a head and tail.

Need for an Anonymous variable

- What if you wanted the 4th value only:
- We could do it as below:

```
[X1, X2, X3, X4 | Tail] = [ a, b, c, d, e, g ].
X1=a
X2=b
X3=c
X4=d
Tail=[e, g]
```

You end up introducing variables you are not really interested in – which is not useful

What we need is an anonymous variable

The anonymous variable _

- To get the second and fourth variable do:

[_, X2, _, X4 | _] = [a, b, c, d, e, g].

X2=b

X4=d

You don't care what the values in the other spots are – you care only about the value in the second and fourth spots in the example above

member predicate for Lists

- You might need to know if some element is a member of a list:

```
member(X, [X | Y]).
member(X, [H | T ]) :- member(X, T).
```

This provides a recursive rule formulation.

"member" predicate is built in to Prolog

Solving A Problem using Lists

- Think of it as a “database”
- Supposing you want to represent a list of people, cars and dogs.
- Represent each entry as a list where the first element is a person, 2nd is the car of the person, and the 3rd is the dog
- So, the database is a list of the form:

[[p1, c1, d1], [p2, c2, d2], [p3, c3, d3]] and so on

Example

```
% define predicate "iright" to represent position in the sequence
iright(L, R,  [ L  | [ R  | _ ]]).
iright(L, R, [ _  | Rest]) :- iright( L,  R,  Rest).

% define the "main" predicate using the clues provided
% mary drives a bentley, jane drives a bmw and has a terrier
% bob owns a doberman and john has a shepherd
myprogram(DogsCars) :-
  =(DogsCars, [ [mary, bentley, _ ], [jane, bmw, terrier], [bob, _ , doberman],
                              [john,  _ , shepherd] ]),
/* person to the right of the bmw drives a benz */
iright([ _ , bmw, _ ], [ _ , benz, _ ], DogsCars),
/* person to the right of bob drives a cooper */
iright([bob,  _ , _ ], [ _ , cooper, _ ], DogsCars),
/* dog to the left of the terrier is a spaniel */
iright([ _ , _ , spaniel ], [ _ , _ , terrier], DogsCars).
```

QUERY can be: myprogram(X). -- will list all the entries through unification and backward chaining

Expanding Prolog

- Parallelization:

  OR-parallelism: goal may unify with many different literals and implications in KB

  AND-parallelism: solve each conjunct in body of an implication in parallel

- Compilation: generate built-in theorem prover for different predicates in KB

- Optimization: for example through re-ordering

  e.g., "what is the income of the spouse of the president?"

  Income(s, i) $\wedge$ Married(s, p) $\wedge$ Occupation(p, President)

  faster if re-ordered as:

  Occupation(p, President) $\wedge$ Married(s, p) $\wedge$ Income(s, i)