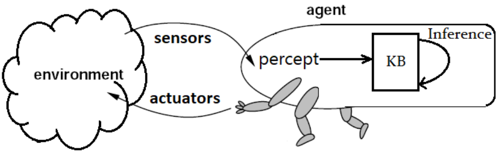
CS761 Artificial Intelligence

First-order Logic Inference: Logic Programming

Recall: Logical agents

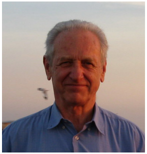
A knowledge-based agent implements the following:

- Knowledge base: Expressing knowledge (i.e., constraints) in a logical language
- Inference engine: Performing reasoning to solve problems.



The "declarative ideal" of Robert Kowalski (1979):

$$\text{Algorithm} = \text{Logic} + \text{Control}$$



Imperative v.s. Declarative Programming

Imperative programs: describe the steps of computation, i.e., how to produce an output.

Example. The procedure to sum a list of integers.

```
int sum(int[] list){
    int result = 0;
    for (int i=0; i<list.length;++i){
        result += list[i];
    }
    return result;
}
```

Declarative programs: describe what output you want, rather than how to produce the output.

Example. The definition of the sum of a list of integers.

```
sum([],0).       % sum of an empty list is 0
sum([FirstItem | Rest], Sum) :-
    sum(Rest,SumRest), Sum is FirstItem + SumRest.
                 % sum of a non-empty list is the first item +
                       sum of the rest of the list.
```

Logic Programming

Logic programming is a declarative programming language paradigm that aims to separate a program into its logic component and its control component:

- the logic component applies logic as a knowledge description language to describe what to compute;
- the control component applies inference to solve the problem.

Prolog is one of the most important logic programming languages:

- Developed in Marseille, France in 1972.
- First-order logic as its knowledge representation language.
- Widely used in theorem proving, expert systems, automated planning, and natural language processing.
- Download and install the client (Windows, Linux, MacOS (not recommended)):

  `https://www.swi-prolog.org/download/stable`

  Online version (recommended):

  `https://swish.swi-prolog.org/`

SWI Prolog

Recall example. *"The law says that it is a crime for a New Zealander to sell alcohol to minors. The girl Lucy is 15 years old and has some beers. All of the beers were sold to her by David, who is a New Zealander."* Prove that David is guilty.

- *"it is a crime for a New Zealander to sell alcohol to minors"*

$$(1)\ Crime(x) \leftarrow NZ(x) \wedge Alcohol(y) \wedge Sells(x, y, z) \wedge Minor(z)$$

- *"Lucy is 15 years old and has some beers"*

$$(2)\ Owns(Lucy, B), (3)\ Beers(B), (4)\ Under17(Lucy)$$

- *"All of the beers were sold to her by David"*

$$(5)\ Sells(David, x, Lucy) \leftarrow Beers(x) \wedge Owns(Lucy, x)$$

- Beers are alcohol

$$(6)\ Alcohol(x) \leftarrow Beers(x)$$

- Anyone younger than 17 years old is a minor.

$$(7)\ Minor(x) \leftarrow Under17(x)$$

- "David, who is a New Zealander"

$$(8)\ NZ(David)$$

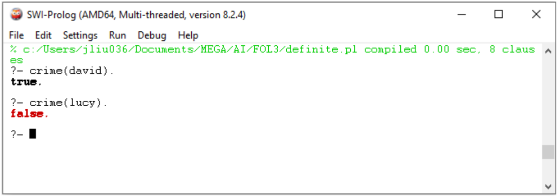
- Query: $Ask(\mathrm{KB}, Crime(David))$

Prolog program:

```
crime(X) :- nz(X), alcohol(Y), sells(X,Y,Z), minor(Z).
owns(lucy,b).
beers(b).
under17(lucy).
sells(david,X,lucy) :- beers(X), owns(lucy,X).
alcohol(X) :- beers(X).
minor(X) :- under17(X).
nz(david).
```

Query:

```
?- crime(david).
true.
?- crime(lucy).
false.
```



Prolog Program

Basic Syntax. A Prolog program implements a first-order knowledge base:

- Facts: Atoms

  E.g. `rainy(dunedin).` `cold(dunedin).` `play.`

- Rules: Head :- $\text{Body}_1, \text{Body}_2, \ldots, \text{Body}_k$.

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

Note.

- Constant, function, relation names all start with small case letters. **E.g.** `dunedin`, `david`, `sum`, `sells`, `''ET''`.
- Variable names start with capital letters or "_". **E.g.** `X, Lucy, _x`
- Left implication: "`:-`"
- Conjunction: "`,`"

Rules. A rule in Prolog

$$\text{Head :- Body}_1, \text{Body}_2, \ldots, \text{Body}_k.$$

represents a logical implication.

- Universal quantification: Variables in the head are universally quantified.

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

  The last statement can be viewed as $\forall x\colon (\mathit{rainy}(x) \wedge \mathit{cold}(x)) \rightarrow \mathit{snowy}(x)$.

- Existential quantification: Variables that only appear in the body are existentially quantified.

  ```
  in(auckland, northIs).
  in(wellington, northIs).
  sameIsland(X, Z) :- in(X, Y), in(Z, Y).
  ```

  The last statement can be viewed as

  $$\forall x, z\colon\ [\exists y\colon \mathit{in}(x, y) \wedge \mathit{in}(z, y)] \rightarrow \mathit{sameIsland}(x, z).$$

Queries. Queries start with "?-"

```
rainy(dunedin).
cold(dunedin).
rainy(wellington).
cold(wellington).
rainy(auckland).
snowy(X) :- rainy(X), cold(X).
```

- True/false queries:

  E.g. `?- snowy(dunedin).`

  ```
  true.
  ```

- Unification in queries:

  E.g.

  ```
  ?- snowy(C).
  C=dunedin ;
  C=wellington.
  ```

Recursive rules. Prolog allows recursive definitions of predicates:

Example.

```
in(hamilton,waikato).
in(waikato,northIs).
belong(X,Y) :- in(X,Y).
belong(X,Y) :- in(X,Z), belong(Z,Y).

?- belong(hamilton,northIs).
true.
```

Negation as failure. \+ denotes the negation symbol in Prolog.
E.g.

```
man(jim).
man(fred).
woman(X) :- \+(man(X)).
?- woman(jim).
false.
?- woman(X).
false.
```

Note.

- `\+(`*literal*`(x))` is true whenever it is not possible to prove *literal*`(x)` to be true, i.e.,

  $$\backslash\texttt{+(}\textit{literal}\texttt{(x))} \text{ returns true } \Leftrightarrow \textit{literal}\texttt{(x)} \text{ does not exist in KB.}$$

- `\+(`*literal*`(X))` is true whenever it is not possible to unify `X` with a constant that makes *literal*`(X)` true, i.e.,

  $$\backslash\texttt{+(}\textit{literal}\texttt{(X))} \text{ returns true } \Leftrightarrow \neg\exists x : \textit{literal}(\texttt{x}).$$

Example. [a simple KB in Prolog]

```
parent(ann,bob).
parent(abe,bob).
parent(bob,dan).
parent(cat,dan).
parent(ann,ema).
parent(dan,fay).
male(abe).
male(bob).
male(dan).
female(X) :- \+(male(X)).

father(X,Y) :- parent(X,Y), male(X).
mother(X,Y) :- parent(X,Y), female(X).
son(X,Y) :- parent(Y,X), male(X).
sister(X,Z) :- parent(Y,X), parent(Y,Z), female(X).
aunt(X,Z) :- sister(X,Y), parent(Y,Z).
grandfather(X,Z) :- father(X,Y), parent(Y,Z).

ancestor(X,Y) :- parent(X,Y).
ancestor(X,Z) :- parent(X,Y), ancestor(Y,Z).
```

Prolog Data Structure: Lists

Prolog defines a list using the [] notation:

- [a, b, c]
- [a | [b, c]], [a, b | [c]], [a, b, c | []]

Checking list membership:

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

Checking if a list is sorted:

```
sorted([]).                % empty list is sorted
sorted([_]).               % singleton is sorted
sorted([A, B | T]) :- A =< B, sorted([B | T]). % =< is a built-in predicate
```

Appending elements to a list:

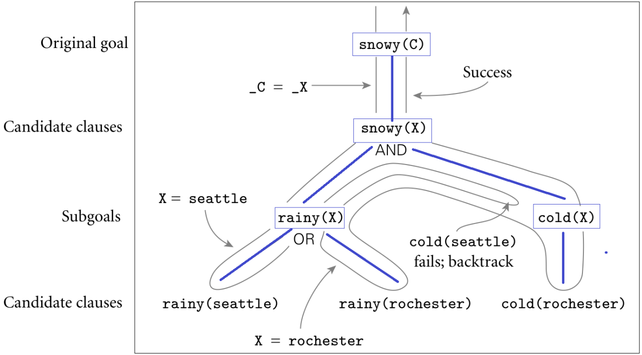
```
append([], A, A).
append([H | T], A, [H | L]) :- append(T, A, L).
?- append([a, b, c], [d, e], L).
L = [a, b, c, d, e].
?- append(X, [d, e], [a, b, c, d, e]).
X = [a, b, c].
```

Inference in Prolog

Backward chaining. Start with query and work backward, attempting to "unresolve" it into a set of pre-existing clauses.

- The backward chaining search can be described by a tree of subgoals.
- Prolog explores this tree using backtracking: depth-first, from top line to bottom line.

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



However, Prolog is not a fully logical programming language.
We are going to discuss three important “imperative features” of Prolog:

- Ordering of literals
- Cut
- Arithmetic

Imperative feature 1: Ordering

- During search, Prolog considers clauses in the fixed order from first to last.
- Prolog programs do not have the same semantics as first-order logic.
- Programmers must be careful with the order of clauses to ensure recursive programs will terminate.

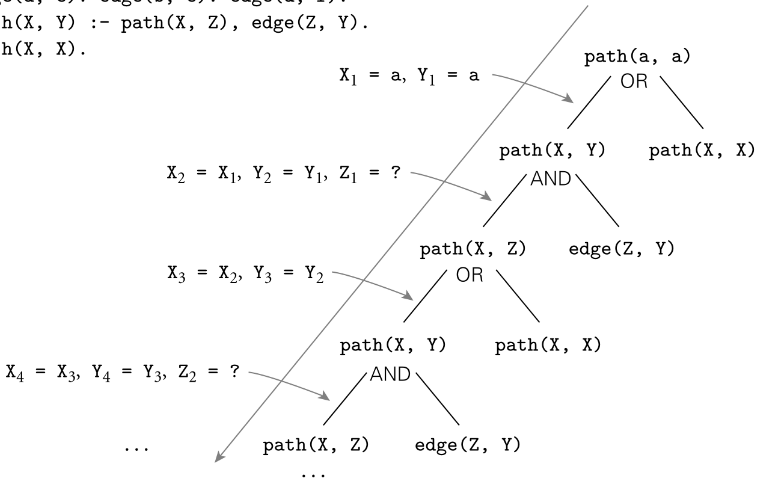
Example. The following code may find all pairs `(X,Y)` with `path(X,Y)`.

```
edge(a, b). edge(b, c). edge(c, d).
edge(d, e). edge(b, e). edge(d, f).
path(X, X).
path(X, Y) :- edge(Z, Y), path(X, Z).
?- path(a, a).
true.
```

The following code will produce an error with `path(X,Y)`.

```
edge(a, b). edge(b, c). edge(c, d).
edge(d, e). edge(b, e). edge(d, f).
path(X, Y) :- path(X, Z), edge(Z, Y).
path(X, X).
?- path(a, a).
ERROR (Memory Limit)
```

```
edge(a, b). edge(b, c). edge(c, d).
edge(d, e). edge(b, e). edge(d, f).
path(X, Y) :- path(X, Z), edge(Z, Y).
path(X, X).
```



Imperative feature 2: Cut

The cut ! is a zero-argument predicate:

- As a subgoal, it is always satisfied.
- Side effect: It commits the interpreter to whatever choices have been made since unifying the parent goal with the left-hand side of the current rule, including the choice of that unification itself.

Example.

```
member(X, [X | _]).
member(X, [_ | T]) :- member(X, T).
prime_candidate(X) :- member(X, candidates), prime(X).
?- prime_candidate(6).
```

Assumption:

- `prime(X)` is expensive. and `prime(6)` is false.
- `candidates` is the list `[6,5,3,6,2,6,6]`.

The search may execute `prime(6)` four times, every time with the same result.

To cut down search time:

```
member(X, [X | _]) :- !.              % Rule 1
member(X, [_ | T]) :- member(X, T).   % Rule 2
```

Cut: When `member(X,L)` unifies with the head of Rule 1, it will no longer be unified with Rule 2 in further search.

Imperative feature 3: Arithmetic

Arithmetic symbols are predicate symbols, not arithmetic operators:

- `2+3` is a two-argument predicate `+(2,3)`, not 5

```
?- (2 + 3) = 5.
false.
```

- `is` is a built-in predicate that unifies the first argument with the arithmetic value of the second argument, which must be instantiated.

```
?- is(5, 2+3).
true.
?- X is 2+3.
X = 5.
?- is(2+3, 5).
false.
?- 5 is Y+3.
ERROR.
```

- Enumerate all natural numbers:

```
natural(1).
natural(N) :- natural(M), N is M+1.
my_loop(N) :- natural(I), write(I), nl, I=N, !.
                                % nl is 'new line' predicate
```

Extended Example: Tic-Tac-Toe

Game rule.

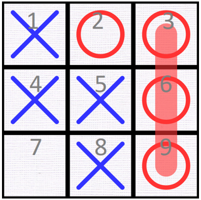
- Play on a 3×3 grid.
- Two players, `X` (agent) and `O`, take turns placing markers in empty squares.
- Any player wins if they place three markers in a (horizontal, vertical, or diagonal) line.

We will develop a Prolog program for Tic-Tac-Toe.

- **Squares:** `1,2,3,4,5,6,7,8,9`
- **Agent perception:** `x(i)`, `o(i)` indicating the markers in square `i`.

```
full(A) :- x(A).   full(A) :- o(A).
empty(A) :- \+(full(A))     % the built-in \+ succeeds if
                            % its argument cannot be proven.
```

- **Agent action:** `move(A)`. Define the next move of the Prolog agent.



Knowledge base:

- Winning conditions:

```
ordered_line(1, 2, 3). ordered_line(4, 5, 6).
ordered_line(7, 8, 9). ordered_line(1, 4, 7).
ordered_line(2, 5, 8). ordered_line(3, 6, 9).
ordered_line(1, 5, 9). ordered_line(3, 5, 7).
line(A, B, C) :- ordered_line(A, B, C).
line(A, B, C) :- ordered_line(A, C, B).
line(A, B, C) :- ordered_line(B, A, C).
line(A, B, C) :- ordered_line(B, C, A).
line(A, B, C) :- ordered_line(C, A, B).
line(A, B, C) :- ordered_line(C, B, A).
```

- `move(A) :- good(A), empty(A).`

We will need to define "good" moves, and use ordering to indicate preference among good moves.

Good move 1. Winning.

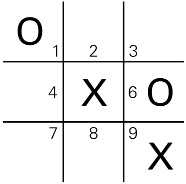
```
win(A) :- x(B), x(C), line(A,B,C).
```

Good move 2. Prevent opponent from winning.

```
block_win(A) :- o(B), o(C), line(A,B,C).
```

Good move 3. Create a “split” situation

```
split(A) :- x(B), x(C), different(B,C),
    line(A,B,D), line(A,C,E), empty(D), empty(E).
same(A,A).
different(A,B) :- \+(same(A,B)).    % negation as failure
```



Good move 4. Get two in a row in such a way that the opponent's blocking move won't build towards three in a row.

```
strong_build(A) :- x(B), line(A,B,C), empty(C), \+(risky(C)).
risky(C) :- o(D), line(C,D,E), empty(E).
```

Good move 5. Get two in a row in such a way that the opponent's blocking move won't become a split.

```
weak_build(A) :- x(B), line(A,B,C), empty(C), \+(double_risky(C)).
double_risky(C) :- o(D), o(E), different(D,E), line(C,D,F),
    line(C,E,G), empty(F), empty(G).
```

Strategy for the Prolog agent.

```
good(A) :- win(A).        good(A) :- block_win(A).
good(A) :- split(A).      good(A) :- strong_build(A).
good(A) :- weak_build(A).
```

Last resort. If none of the above, pick an unoccupied square, giving priority to the centre, the corners, and the sides in order.

```
good(5).    % centre
good(1).    good(3).    good(7).    good(9).    % corners
good(2).    good(4).    good(6).    good(8).    % sides
```

Summary of The Topic

The following are the main knowledge points covered:

- Declarative programming is different from imperative programming in that a declarative program describes what output you want, rather than how to produce the output.
- **Logic programming**: A declarative programming paradigm that aim to decompose a programming language into a logic component (to represent knowledge), and a control component (to perform inference).
- **Prolog**: is an important logic programming language that aims to implement a first-order knowledge base.
- **Prolog inference**: Backward chaining starts from the goal and works backwards while traversing the subgoal tree using backtracking.
- Prolog does not fully correspond to FO logic:
  - Ordering
  - Arithmetics
  - Cut