10. Logic Programming With Prolog

Overview
- Logic Programming
- Prolog

Note: These notes cover Section 11.3 of the textbook excluding 11.3.2.

Logic Programming
- Logic programming is a form of declarative programming
- A program is a collection of axioms
  - Each axiom is a Horn clause of the form:
    \[ H \leftarrow B_1, B_2, ..., B_n \]
    where \( H \) is the head term and \( B_i \) are the body terms
  - Meaning \( H \) is true if all \( B_i \) are true
- A user of the program states a goal (a theorem) to be proven
  - The logic programming system attempts to find axioms using inference steps that imply the goal (theorem) is true

Resolution
- To deduce a goal (theorem), the logic programming system searches axioms and combines sub-goals
- For example, given the axioms:
  \[ C \leftarrow A, B. \]
  \[ D \leftarrow C. \]
- To deduce goal \( D \) given that \( A \) and \( B \) are true:
  - **Forward chaining** deduces that \( C \) is true:
    \[ C \leftarrow A, B \]
    and then that \( D \) is true:
    \[ D \leftarrow C \]
  - **Backward chaining** finds that \( D \) can be proven if sub-goal \( C \) is true:
    \[ D \leftarrow C \]
    the system then deduces that the sub-goal is \( C \) is true:
    \[ C \leftarrow A, B \]
    Since the system could prove \( C \) it has proven \( D \)

Prolog
- Uses backward chaining
  - More efficient than forward chaining for larger collections of axioms
- Interactive (hybrid compiled/interpreted)
- Applications: expert systems, artificial intelligence, natural language understanding, logical puzzles and games
- Popular system: SWI-Prolog
  - Login `linprog.cs.fsu.edu`
  - Type: `pl` to start SWI-Prolog
  - Type: `halt`, to halt Prolog (note that a period is used as a command terminator)
Prolog Terms

- Terms are symbolic expressions that form the building blocks of Prolog
  - A Prolog program consists of terms
  - Data structures processed by a Prolog program are terms
- A term is either
  - a variable: a name beginning with an upper case letter
  - a constant: a number or string
  - an atom: a symbol or a name beginning with a lower case letter
  - a structure of the form: functor(arg1, arg2, ..., argn)
    where functor is an atom and argi are terms
- Examples:
  - X, Y, ABC, and Alice are variables
  - 7, 3.14, and "hello" are constants
  - foo, bAR, and + are atoms
  - bin_tree(foo, bin_tree(bar, glarch)) and +(3,4) are structures

Prolog Clauses

- A program consists of a database of Horn clauses
- Each clause consists of a head predicate and body predicates:
  \[ H \leftarrow B_1, B_2, ..., B_n \]
  - A clause is either a rule, e.g.
    \[ \text{snowy}(X) \leftarrow \text{rainy}(X), \text{cold}(X). \]
    Meaning "If X is rainy and X is cold then this implies that X is snowy"
  - Or a clause is a fact, e.g.
    \[ \text{rainy}(\text{rochester}). \]
    Meaning "Rochester is rainy."
    This fact is identical to the rule with true as the body predicate:
    \[ \text{rainy}(\text{rochester}) \leftarrow \text{true}. \]
- A predicate is a term (must be an atom or a structure)
  - \text{rainy}(\text{rochester})
  - \text{member}(X,Y)
  - true

Queries and Goals

- Queries are used to "execute" goals
  - A query is interactively entered by a user after a program is loaded and stored in the database
  - A query has the form
    \[ ?- G_1, G_2, ..., G_n \]
    where \( G_i \) are goals
- A goal is a predicate to be proven true by the programming system
  - Example program with two facts:
    \[ \text{rainy}(\text{seattle}). \]
    \[ \text{rainy}(\text{rochester}). \]
  - Query with one goal to find which city C is rainy (if any):
    \[ ?- \text{rainy}(C). \]
    Response by the interpreter:
    \[ C = \text{seattle}; \]
    Type a semicolon ; to get next solution:
    \[ C = \text{rochester}; \]
    Type another semicolon ;:
    no
    (no more solutions)

Example

- Program with three facts and one rule:
  \[ \text{rainy}(\text{seattle}). \]
  \[ \text{rainy}(\text{rochester}). \]
  \[ \text{cold}(\text{rochester}). \]
  \[ \text{snowy}(X) \leftarrow \text{rainy}(X), \text{cold}(X). \]
- Query and response:
  \[ ?- \text{snowy}(\text{rochester}). \]
  yes
- Query and response:
  \[ ?- \text{snowy}(\text{seattle}). \]
  no
- Query and response:
  \[ ?- \text{snowy}(\text{paris}). \]
  no
Example (cont’d)

- Program:
  rainy(seattle).
  rainy(rochester).
  cold(rochester).
  snowy(X) :- rainy(X), cold(X).
- ?- snowy(C).
  C = rochester

because rainy(rochester) and cold(rochester) are sub-goals that are both true facts in the database

- snowy(X) with X=seattle is a goal that fails, because cold(X) fails, triggering backtracking

Backtracking

- For every successful match of a (sub-)goal with a head predicate of a clause, the system keeps this execution point in memory together with the current variable bindings to enable backtracking
- An unsuccessful match later forces backtracking in which alternative clauses are searched that match (sub-)goals
- Backtracking unwinds variable bindings to allow establishing new bindings

Unification and Variables

- In the previous notes we saw the use of variables, e.g. C and X
- A variable is instantiated to a term as a result of unification
- Unification takes place when goals are matched to head predicates of rules and facts
  - Goal in query: rainy(C)
  - Fact in database: rainy(seattle)
  - Unification is the result of the goal-fact match: C = seattle
- Unification is recursive:
  - An uninstantiated variable unifies with anything, even with other variables which makes them identical (aliases)
  - An atom unifies with an identical atom
  - A constant unifies with an identical constant
  - A structure unifies with another structure if the functor and number of arguments are the same and the corresponding arguments unify recursively
- Once a variable is instantiated to a non-variable term, it cannot be changed and cannot be instantiated with a term that has a different structure

Unification Examples

- The built-in predicate = (A,B) succeeds if and only if A and B can be unified
- The goal = (A,B) may be written as A = B
  - ?- a = a.
    yes
  - ?- a = 5.
    no
  - ?- 5 = 5.0.
    no
  - ?- a = X.
    X = a
  - ?- foo(a,b) = foo(a,b).
    yes
  - ?- foo(a,b) = foo(X,b).
    X = a
  - ?- foo(X,b) = Y.
    Y = foo(X,b)
  - ?- foo(Z,Z) = foo(a,b).
    no
Lists

- A list is of the form: 
  \[ [elt_1, elt_2, \ldots, elt_n] \]
  where \( elt \) are terms
- The special list form
  \[ [elt_1, elt_2, \ldots, elt_n | tail] \]
denotes a list whose tail list is \( tail \)

PEG饽 Transport

?- [a,b,c] = [a|T].
  T = [b,c]

?- [a,b,c] = [a,b|T].
  T = [c]

?- [a,b,c] = [a,b,c|T].
  T = []

List Membership

- List membership is tested with the \texttt{member} predicate, defined by
  \begin{align*}
  \text{member}(X_1, [X_1|T_1]). \\
  \text{member}(X_1, [H_1|T_1]) :& \text{- member}(X_1, T_1).
  \end{align*}
- Execution:
  \begin{itemize}
  \item \texttt{member(b, [a,b,c])} does not match predicate \texttt{member}(X_1, [X_1|T_1]).
  \item \texttt{member(b, [a,b,c])} matches predicate \texttt{member}(X_1, [H_1|T_1])
    with \( X_1 = b, H_1 = a \), and \( T_1 = [b,c] \).
  \item Sub-goal to prove: \texttt{member}(X_1, T_1) with \( X_1 = b \) and \( T_1 = [b,c] \).
  \item \texttt{member(b, [a,b,c])} matches predicate \texttt{member}(X_2, [X_2|T_2]).
    with \( X_2 = b \) and \( T_2 = [c] \).
  \item The sub-goal is proven, so \texttt{member(b, [a,b,c])} is proven (deduced).
  \end{itemize}
- Note: variables are "local" to a clause (just like the formal arguments of a function)
- Local variables such as \( X_1 \) and \( X_2 \) are used to indicate a match of a (sub)-goal and a head predicate of a clause.

Predicates are Relations

- Predicates are not functions with distinct inputs and outputs
- Predicates are more general and define \textit{relationships} between objects (terms)
  \begin{itemize}
  \item \texttt{member(b, [a,b,c])} \textit{relates} term \( b \) to the list that contains \( b \)
  \item ?- \texttt{member(X, [a,b,c]).}
    \begin{align*}
    X &= a ; \ % \text{ type } ';': \text{ to try to find more solutions} \\
    X &= b ; \ % \ldots \text{ try to find more solutions} \\
    X &= c ; \ % \ldots \text{ try to find more solutions} \\
    \text{no}
    \end{align*}
  \item ?- \texttt{member(b, [a,Y,c]).}
    \begin{align*}
    Y &= b \\
    \text{no}
    \end{align*}
  \item ?- \texttt{member(b, L).}
    \begin{align*}
    L &= [b|_G255] \\
    \text{therefore, } L \text{ is a list with } b \text{ as head and } _G255 \text{ as tail, where}
    \text{ } _G255 \text{ is a new variable}
    \end{align*}
  \end{itemize}
- List appending predicate:
  \begin{itemize}
  \item \texttt{append([], A, A)}.
  \item \texttt{append([H|T], A, [H|L]) : append(T, A, L)}.
  \item ?- \texttt{append([a,b,c], [d,e], X)}.
    \begin{align*}
    X &= [a,b,c,d,e] \\
    ?- \texttt{append(Y, [d,e], [a,b,c,d,e])}.
    Y &= [a,b,c] \\
    ?- \texttt{append([a,b,c], Z, [a,b,c,d,e])}.
    Z &= [d,e]
    \end{align*}
  \end{itemize}

Imperative Control Flow

- Prolog offers a few built-in constructs to support a form of control-flow
  \begin{itemize}
  \item \texttt{not G} negates a (sub-)goal \( G \)
  \item \texttt{! (cut)} terminates backtracking for a predicate and within the body of the clause of that predicate
  \item \texttt{fail} always fails
  \end{itemize}
- Examples
  \begin{itemize}
  \item ?- \texttt{not member(b, [a,b,c]).}
    \begin{align*}
    \text{no}
    \end{align*}
  \item ?- \texttt{not member(b, [])}.
    \begin{align*}
    \text{yes}
    \end{align*}
  \item Define:
    \begin{align*}
    \text{if(Cond, Then, Else)} &::= \text{Cond}, \ Then. \\
    \text{if(Cond, Then, Else)} &::= \text{Else.}
    \end{align*}
  \item ?- \texttt{if(true, X=a, X=b).}
    \begin{align*}
    X &= a ; \ % \text{ type } ';': \text{ to try to find more solutions} \\
    \text{no}
    \end{align*}
  \item ?- \texttt{if(fail, X=a, X=b).}
    \begin{align*}
    X &= b ; \ % \text{ type } ';': \text{ to try to find more solutions} \\
    \text{no}
    \end{align*}
  \item ?- \texttt{if(fail, a=b, X=b).}
    \begin{align*}
    \text{no}
    \end{align*}
  \end{itemize}
- The cut makes sure that the \texttt{Cond} is not executed again upon backtracking and that the second \texttt{if}-clause is not executed when \texttt{Cond} is true when backtracking.
- Therefore, this example would not work without the cut when backtracking.

Example: Bubble Sort

- \texttt{bubble(List, Sorted) :-}
append(InitList, [B,A|Tail], List),
    A < B,
append(InitList, [A,B|Tail], NewList),
bubble(NewList, Sorted).
bubble(List, List).
?- bubble([2,3,1], L).
append([], [2,3,1], [2,3,1]),
3 < 2, fails: backtrack
append([2], [3,1], [2,3,1]),
1 < 3,
append([2], [1,3], NewList₁), this makes: NewList₁=[2,1,3]
bubble([2,1,3], L),
append([], [2,1,3], [2,1,3]),
1 < 2,
append([], [1,2,3], NewList₂), this makes:
NewList₂=[1,2,3]
    bubble([1,2,3], L).
append([], [1,2,3], [1,2,3]),
2 < 1, fails: backtrack
append([1], [2,3], [1,2,3]),
3 < 2, fails: backtrack
append([1,2], [3], [1,2,3]), does not unify:
backtrack
bubble([1,2,3], L). try second bubble-clause which makes
L=[1,2,3]
bubble([2,1,3], [1,2,3]).
bubble([2,3,1], [1,2,3]).

Example: Tic-Tac-Toe

- Board layout:
  \[
  \begin{array}{ccc}
    1 & 2 & 3 \\
    4 & 5 & 6 \\
    7 & 8 & 9 \\
  \end{array}
  \]

- Facts:
  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).

Example: Tic-Tac-Toe (cont’d)

- Rules to find line of three (permuted) cells:
  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).

- How to make a good move to a cell:
  move(A) :- good(A), empty(A).
  Which cell is empty?

- Which cell is full?
  full(A) :- x(A).
  full(A) :- o(A).

- Which cell is best to move to? (check this in this order)
  good(A) :- win(A). % a cell where we win
  good(A) :- block_win(A). % a cell where we block the
    opponent from a win
  good(A) :- split(A). % a cell where we can make a split to
    win
  good(A) :- block_split(A). % a cell where we block the
    opponent from a split
  good(A) :- build(A). % choose a cell to get a line
  good(5). % choose a cell in a good location
  good(1).
  good(3).
  good(7).
  good(9).
  good(2).
  good(4).
  good(6).
  good(8).

Example: Tic-Tac-Toe (cont’d)

- How to find a winning cell:
  win(A) :- x(B), x(C), line(A,B,C).
  Choose a cell to block the opponent from choosing a winning cell:
  block_win(A) :- o(B), o(C), line(A,B,C).
  Choose a cell to split for a win later:
  split(A) :- x(B), x(C), not (B = C), line(A,B,D), line(A,C,E),
    empty(D), empty(E).

- Choose a cell to block the opponent from making a split:
  block_split(A) :- o(B), o(C), not (B = C), line(A,B,D),
    line(A,C,E), empty(D), empty(E).
  Choose a cell to get a line:
  build(A) :- x(B), line(A,B,C), empty(C).
Example: Tic-Tac-Toe (cont’d)

- Board positions:

```
  O  
 X O  
  X  
```

- Are stored as facts in the database:
  x(7).
o(5).
x(4).
o(1).

- Move query:
  ?- move(A).
  A = 9

Arithmetic

- Arithmetic is essential for many computations in Prolog
- The `is` predicate evaluates an arithmetic expression and instantiates a variable with the result
  - For example
    
    \[ X = 2\sin(1) \]
    
    instantiates \( X \) with the result of \( 2\sin(1) \)

- Example
  - A predicate to compute the length of a list:
    
    ```
    length([], 0).
    length([H|T], N) :- length(T, K), N is K + 1.
    ```
    
    where the first argument of `length` is a list and the second is the computed length
  - Example query:
    
    ```
    ?- length([1,2,3], X).
    X = 3
    ```