Relations

- parent(X, Y): X is a parent of Y.
  - parent(pam, bob).
  - parent(tom, bob).
  - parent(tom, liz).
  - parent(bob, ann).
  - parent(bob, pat).
  - parent(pat, jim).

- male(X): X is a male.
  - female(pam).
  - female(pat).
  - female(ann).
  - female(liz).

- female(X): X is a female.
  - male(tom).
  - male(bob).
  - male(jim).

- mother(X, Y): X is the mother of Y.

\[ \forall X, Y. \ parent(X, Y) \land \ female(X) \Rightarrow mother(X, Y) \]

- In Prolog: mother(X, Y) :- parent(X, Y), female(X).
Representing relations

parent(pam, bob).
parent(tom, bob).
parent(tom, liz).
parent(bob, ann).
parent(bob, pat).
parent(pat, jim).
female(pam).
female(pat).
female(ann).
female(liz).
male(tom).
male(bob).
male(jim).

More Relations

- grandparent(X, Y) :- parent(X, Z), parent(Z, Y).
- sibling(X, Y) :- parent(Z, X), parent(Z, Y), X ≠ Y.
More Relations

- cousin(X,Y) :- .......
- greatgrandparent(X,Y) :- ........
- greatgreatgrandparent(X,Y) :- ........
- ancestor(X,Y) :- ...

Computations in Prolog

|?- mother(M, bob).
|?-. parent(M, bob), female(M).
|?-. M=pam, female(pam).
    M = pam

|?- father(M, bob)
|?-. parent(M, bob), male(M)
    (i) |?- M=pam, male(pam).
         fail
    (ii) |?- M=tom, male(tom).
         M = tom
Prolog Execution

Call: Call a predicate (invocation)
Exit: Return an answer to the caller
Fail: Return to caller with no answer
Redo: Try next path to find an answer

Syntax of Prolog Programs

- A program is a sequence of clauses.
- Each clause is of the form head :- body.
- Head is a term.
- Body is a comma-separated list of terms.
- Clause with an empty body is called a fact.
- A clause is also sometimes called a rule.
Terms

- Atomic data
- Variables
- Structures

Atomic Data

- **Numeric constants:** Integers, floating point numbers (e.g. 1024, -42, 3.1415, 6.023e23 ...)
- **Atoms:**
  - Strings of characters enclosed in single quotes (e.g. 'cram', 'Stony Brook')
  - Identifiers: sequence of letters, digits, underscore, beginning with a letter (e.g. cram, r2d2, x_24).
Variables

- Variables are denoted by identifiers beginning with an *Uppercase letter or underscore* (e.g. X, Index, _param).
- Underscore, by itself, represents an *anonymous variable*.
- Different occurrences of the same variable in a clause denote the same data.
- Each occurrence of an anonymous variable is treated as a different data.
- Variables are implicitly declared upon first use.
- Variables are not typed.
- Variables can be assigned only once, but that value can be further refined.

Structures

(We’ll come to this topic later . . .)
Meaning of Logic Programs

- **Declarative Meaning:** What are the logical consequences of a program?
- **Procedural Meaning:** For what values of the variables in the query can I prove the query?

### Declarative Meaning

```prolog
big(bear). brown(bear).
big(elephant). black(cat).
small(cat). gray(elephant).

dark(Z) :- black(Z). dark(Z) :- brown(Z).
dangerous(X) :- dark(X), big(X).
```

- Logical consequence of a program $L$ is the smallest set such that
  - All facts of the program are in $L$
  - If $H : \neg B_1, B_2, \ldots, B_n$ is an instance of a clause in the program such that $B_1, B_2, \ldots, B_n$ are all in $L$, then $H$ is also in $L$.
- For the above program we get big(bear) and dark(bear) and consequently dangerous(bear).
Procedural Meaning of Prolog

big(bear).  brown(bear).  dark(Z) :- black(Z).
big(elephant).  black(cat).  dark(Z) :- brown(Z).
small(cat).  gray(elephant).  dangerous(X) :- dark(X), big(X).

...
Derivations

big(bear). brown(bear).
big(elephant). black(cat).
small(cat). gray(elephant).
dark(Z) :- black(Z). dark(Z) :- brown(Z).
dangerous(X) :- dark(X), big(X).

dangerous(Q) :-
  dark(Q), big(Q).
  
dark(Q) :- black(Q). dark(Q) :- brown(Q).
  
  black(Q), big(Q) brown(Q), big(Q)

  black(cat). big(cat) big(bear)

  fail succeed

Structures

- If \( f \) is an identifier and \( t_1, t_2, \ldots, t_n \) are terms, then \( f(t_1, t_2, \ldots, t_n) \) is a term.

- In the above, \( f \) is called a functor and \( t_i \) is an argument.
- Structures are used to group related data items together (in some ways similar to struct in C and objects in Java).
- Structures are used to construct trees (and, as a special case, lists).
Trees

- Example: expression trees:
  \[ \text{plus}(\text{minus}(\text{num}(3), \text{num}(1)), \text{star}(\text{num}(4), \text{num}(2))) \]

- Data structures may have variables. And the same variable may occur multiple times in a data structure.

Matching

(We'll later introduce unification, a related operation that has logical semantics).

- \( t_1 = t_2 \): find substitutions for variables in \( t_1 \) and \( t_2 \) that make the two terms identical.

  Yes, with \( X = 1, Y = 4 \).

  Yes, with \( X = 1, Y = 4 \).
Accessing arguments of a structure

- Matching is the predominant means for accessing a structure’s arguments.
- Let date(’Sep’, 1, 2005) be a structure used to represent dates, with the month, day and year as the three arguments (in that order).
- Then date(M, D, Y) = date(’Sep’, 1, 2005) makes M = ’Sep’, D = 1, Y = 2005.
- If we want to get only the day, we can write date(_, D, _) = date(’Sep’, 1, 2005). Then we get D = 1.
Lists

Prolog uses a special syntax to represent and manipulate lists.

- \([1,2,3,4]\): represents a list with 1, 2, 3 and 4, respectively.
- This can also be written as \([1 \mid [2,3,4]]\): a list with 1 as the head (its first element) and \([2,3,4]\) as its tail (the list of remaining elements).
- If \(X = 1\) and \(Y = [2,3,4]\) then \([X\mid Y]\) is same as \([1,2,3,4]\).
- The empty list is represented by \([\ ]\).
- The symbol “\(|\)’’ (called cons) and is used to separate the beginning elements of a list from its tail.
  For example: \([1,2,3,4] = [1 \mid [2,3,4]]\)
  \(= [1 \mid [2 \mid [3,4]]]\)
  \(= [1,2 \mid [3,4]]\)

Lists (contd.)

- Lists are special cases of trees.
  For instance, the list \([1,2,3,4]\) is represented by the following structure:

- The function symbol .\(/2 is the list constructor.
  \([1,2,3,4]\) is same as \(.\(1, .\(2, .\(3, .\(4, []\))))\))
Programming with Lists — I

First example: member/2, to find if a given element occurs in a list:

**The program:**

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

**Example queries:**

```
member(s, [l,i,s,t])
member(X, [l,i,s,t])
member(f(X), [f(1), g(2), f(3), h(4), f(5)])
```

Programming with Lists — II

append/3: concatenate two lists to form the third list.

**The program:**

```
append([], L, L).
append([X|Xs], Ys, [X|Zs]) :- append(Xs, Ys, Zs).
```

**Example queries:**

```
append([f,i,r], [s,t], L)
append(X, Y, [s,e,c,o,n,d])
append(X, [t,h], [f,o,u,r,t,h])
```
Programming with Lists — III

Define a predicate, \texttt{len/2} that finds the length of a list (first argument).

\textbf{The program:}
\begin{verbatim}
len([], 0).
len([_|Xs], N+1) :- len(Xs, N).
\end{verbatim}

\textbf{Example queries:}
\begin{verbatim}
len([], X)
len([l,i,s,t], 4)
len([l,i,s,t], X)
\end{verbatim}

\textbf{Arithmetic}

\texttt{| ?- 1+2 = 3.}
\texttt{no}

- In \textit{Predicate logic}, the basis for Prolog, the only symbols that have a meaning are the predicates themselves.
- In particular, function symbols are uninterpreted: have no special meaning and can only be used to construct data structures.
- Meaning for arithmetic expressions is given by the \textit{built-in} predicate "is":
  - \texttt{X is 1 + 2} succeeds, binding \texttt{X} to \texttt{3}.
  - \texttt{3 is 1 + 2} succeeds.
  - General form: \texttt{R is E} where \texttt{E} is an expression to be evaluated and \texttt{R} is matched with the expression's value.
  - \texttt{Y is X + 1} will give an error if \texttt{X} does not (yet) have a value.
The list length example revisited

Define a predicate, \texttt{length/2} that finds the length of a list (first argument).

The program:

\begin{verbatim}
length([], 0).
length([_|Xs], M) :- length(Xs, N), M is N+1.
\end{verbatim}

Example queries:

\begin{verbatim}
length([], X)
length([l,i,s,t], 4)
length([l,i,s,t], X)
length(List, 4)
\end{verbatim}

---

Conditional Evaluation

Consider the computation of \( n! \), i.e. the factorial of \( n \).

\texttt{factorial(N, F) :- ...}

- \( N \) is the input parameter; and \( F \) is the output parameter.
- The body of the rule specifies how the output is related to the input.
- For factorial, there are two cases: \( N \leq 0 \) and \( N > 0 \).
  - \( N \leq 0 \): \( F = 1 \)
  - \( N > 0 \): \( F = N \times (N - 1)! \)

\texttt{factorial(N, F) :-}
  \( N > 0 \)
  \( \rightarrow \) \( N1 \) is \( N \)-1, factorial(N1, F1), \( F \) is \( N \times F1 \)
  ; \( F = 1 \)
).
More Prolog Syntax

- Assignments with arithmetic expressions is done using the keyword “is”.
- If-then-else is written as (\textit{cond} \rightarrow \textit{then-part} ; \textit{else-part})
- If more than one action needs to be performed in a rule, they are written one after another, separated by a comma.
- Arithmetic expressions are not directly used as arguments when calling a predicate; they are first evaluated, and then passed to the called predicate.

Arithmetic Operators

- Integer/Float operators: +, -, *, /
- Integer operators: mod, // (div)
- Int \leftrightarrow Float operators: floor, ceiling
- Comparison operators: <, >, =<, >=, =:=, =\neq
Programming with Lists

Define delete/3, to remove a given element from a list. E.g. delete([1,2,3], 2, X) should succeed with X = [1,3]. (called select/3 in XSB’s basics library)

The program:
delete([X|Ys], X, Ys).
delete([Y|Ys], X, [Y|Zs]) :- delete(Ys, X, Zs).

Example queries:
delete([l,i,s,t], s, X)
delete([l,i,s,t], X, Y)
delete(X, s, [l,i,t])
delete(X, Y, [l,i,s,t])

Permutations

Define permute/2, to find a permutation of a given list. E.g. permute([1,2,3], X) should return X=[1,2,3] and upon backtracking, X=[1,3,2], X=[2,1,3], X=[2,3,1], X=[3,1,2], and X=[3,2,1].

Hint: What is the relationship between the permutations of [1,2,3] and the permutations of [2,3]?

<table>
<thead>
<tr>
<th>permute([2,3],Y)</th>
<th>permute([1,2,3],Y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>[2,3]</td>
<td>[1,2,3]</td>
</tr>
<tr>
<td></td>
<td>[2,1,3]</td>
</tr>
<tr>
<td></td>
<td>[2,3,1]</td>
</tr>
<tr>
<td>[3,2]</td>
<td>[1,3,2]</td>
</tr>
<tr>
<td></td>
<td>[3,1,2]</td>
</tr>
<tr>
<td></td>
<td>[3,2,1]</td>
</tr>
</tbody>
</table>
The program:

\[
\text{permute}([], []). \\
\text{permute}([X|Xs], Ys) :- \text{permute}(Xs, Zs), \text{delete}(Ys, X, Zs).
\]

The Issue of Efficiency

Define a predicate, \( \text{rev}/2 \) that finds the reverse of a given list.
E.g. \( \text{rev}([1,2,3], X) \) should succeed with \( X = [3,2,1] \).

Hint: what is the relationship between the reverse of \( [1,2,3] \) and the reverse of \( [2,3] \)?

The program:

\[
\text{rev}([], []). \\
\text{rev}([X|Xs], Ys) :- \text{rev}(Xs, Zs), \text{append}(Zs, [X], Ys).
\]

How long does it take to evaluate \( \text{rev}([1,2,\ldots,n], X) \)?

\[
T(n) = T(n-1) + \text{time to add 1 element to the end of an } n-1 \text{ element list} \\
T(n) = T(n-1) + n - 1 \\
T(n) = O(n^2)
\]
Making \texttt{rev/2} faster

- Keep an \textit{accumulator}: a stack all elements seen so far. i.e. a list, with elements seen so far in reverse order.

\begin{verbatim}
rev([X|Xs], AccBefore, AccAfter) :-
  rev(Xs, [X|AccBefore], AccAfter).
\end{verbatim}

- So, \texttt{rev([1,2,3], [], X)}
  calls \texttt{rev([2,3], [1], X)}
  calls \texttt{rev([3], [2,1], X)}
  calls \texttt{rev([], [3,2,1], X)}

- Base case:
  \texttt{rev([], Acc, Acc)}.

- Top-Level:
  \texttt{rev(L1, L2) :- rev(L1, [], L2)}.

Tree Traversal

Assume you have a binary tree, represented by

- \texttt{node/3} facts (for internal nodes: \texttt{node(a,b,c)} means that \texttt{a} has \texttt{b} and \texttt{c} as children).
- \texttt{leaf/1} facts (for leaves: \texttt{leaf(a)} means that \texttt{a} is a leaf).

Write a predicate \texttt{preorder/2} that traverses the tree (starting from a given node) and returns the list of nodes in pre-order.

\begin{verbatim}
preorder(Root, [Root]) :- leaf(Root).
presoer(Root, [Root|L]) :-
  node(Root, Child1, Child2),
  preorder(Child1, L1),
  preorder(Child2, L2),
  append(L1, L2, L).
\end{verbatim}

The program takes \(O(n^2)\) time to traverse a tree with \(n\) nodes.
Difference Lists

- The lists in Prolog are singly-linked; hence we can access the first element in constant time, but need to scan the entire list to get the last element.
- Unlike functional languages like Lisp or SML, we can use **variables in datastructures**.
- We can exploit this to make lists “open tailed”:
  - E.g. when \( X = [1, 2, 3|Y] \), \( X \) is a list with 1, 2, 3 as its first three elements, followed by \( Y \).
  - Now if \( Y = [4|Z] \) then \( X = [1, 2, 3, 4|Z] \).
  - We can think of \( Z \) as “pointing to” the end of \( X \).
  - We can now add an element to the end of \( X \) in constant time!! (e.g. \( Z = [5 | W] \))
- Open-tailed lists are also called **difference lists**.

Tree Traversal, Revisited

```prolog
preorder1(Node, List, Tail) :-
    node(Node, Child1, Child2),
    List = [Node|List1],
    preorder1(Child1, List1, Tail1),
    preorder1(Child2, Tail1, Tail).

preorder1(Node, [Node|Tail], Tail) :- leaf(Node).

preorder(Node, List) :- preorder1(Node, List, []).
```

The program takes \( O(n) \) time to traverse a tree with \( n \) nodes.
Difference Lists: Conventions

(Chap. 8.5.3 of Bratko)

- An difference list is represented by two variables: one referring to the entire list, and another to its (uninstantiated) tail.
  e.g. \( X = [1,2,3|Z] \).

- Most Prolog programmers use the notation \( \text{List} - \text{Tail} \) to denote a list \( \text{List} \) with tail \( \text{Tail} \).
  Note that “−” is used as a data structure symbol (not used here for arithmetic).

- The preorder traversal program may be written as:

  ```prolog
  preorder1(Node, [Node|L]-T) :- node(Node, Child1, Child2),
                                    preorder1(Child1, L-T1),
                                    preorder1(Child2, T1-T).
  preorder1(Node, [Node|T]-T).
  ```