An example of user defined data structures.

The Problem: Recall that a binary search tree (with integer labels) is either:
1. the empty tree empty,
or
2. a node labelled with an integer N, that has a left subtree and a right subtree, each of which is a binary search tree such that the nodes in the left subtree are labelled by integers strictly smaller than N, while those in the right subtree are strictly greater than N.

**Data Types in Prolog**

The primitive data types in prolog can be combined via structures to form complex datatypes:

<structure> ::= <functor>(<arg1>, <arg2>, ...)

Example in the case of binary search trees we have:

```
<bstree> ::= empty
           | node(<number>, <bstree>, <bstree>)
node(15, node(2, node(0, empty, empty),
               node(10, node(9, node(3, empty, empty),
                     empty),
               node(12, empty, empty))),
node(16, empty, node(19, empty, empty)))
```
**Binary Search Trees II**

- **The Problem:** Define a unary predicate `isbtree` which is true only of those trees that are binary search trees.

- **The Program**

```prolog
isbtree(empty).
isbtree(node(N,L,R)) :- number(N), isbtree(L), isbtree(R), smaller(N,empty), smaller(N,node(M,L,R)) :- N < M, smaller(N,L), smaller(N,R),
smaller(N,empty).
smaller(N,node(M,L,R)) :- N < M, smaller(N,L), smaller(N,R),
smaller(N,node(M,L,R)) :- N > M, bigger(N,L), bigger(N,R).
```

**Watch it work:**

```prolog
| ?- isbtree(node(9,node(3,empty,empty),empty)). |
| ?- isbtree(node(9,node(3,empty,empty),empty)). true |
yes
```

**Binary Search Trees III**

- **The Problem:** Define a relation which tells whether a particular number is in a binary search tree.

  `mymember(N,T)` should be true if the number N is in the tree T.

- **The Program**

```prolog
mymember(K,node(K,_,_)).
mymember(K,node(N,S,_)) :- K < N, mymember(K,S).
mymember(K,node(N,_,T)) :- K > T, mymember(K,T).
```
Watch it work:

?- [btree].
?- [mymember].
?- member(3,
    node(10, node(9, node(3, empty, empty), empty),
    node(12, empty, empty))).
true ?
yes

Examples (1): list

list(Xs):- Xs is a list.
    list([]).
    list([X|Xs]):- list(Xs).

Examples (2): member

member(Element,List):- Element is an element of the list List.
    member(X, [X|Xs]).
    member(X, [Y|Ys]):- member(X, Ys).
**Examples (3): prefix**

prefix(Prefix, List) :- Prefix is a prefix of List.
prefix([], Ys).
prefix([X|Xs], [X|Ys]) :- prefix(Xs, Ys).

**Examples (4): suffix**

suffix(Suffix, List) :- Suffix is a suffix of List.
suffix(Xs, Xs).
suffix(Xs, [Y|Ys]) :- suffix(Xs, Ys).

**Examples (5): sublist**

sublist(Sub, List) :- Sub is a sublist of List.

a. Suffix of a prefix
   sublist(Xs, Ys) :- prefix(Ps, Ys), suffix(Xs, Ps)

b. Prefix of a suffix
   sublist(Xs, Ys) :- prefix(Xs, Ss), suffix(Ss, Ys)

c. Recursive definition of a sublist
   sublist(Xs, Ys) :- prefix(Xs, Ys)
   sublist(Xs, [Y|Ys]) :- sublist(Xs, Ys)
**Examples (6): member using sublist**

```
Member(X, Xs) :- sublist([X], Xs).
```

**Examples (7): Suffix using append**

```
append(Xs, Ys, XsYs) :- XsYs is the result of concatenating the lists Xs and Ys.
append([], Xs, Ys).
append([X|Xs], Ys, [X|Zs]) :- append(Xs, Ys, Zs)
```

**Examples (8): Sublist using append**

d. Suffix of a prefix using append
```
sublist(Xs, AsXsBs) :-
    append(As, XsBs, AsXsBs), append(Xs, Bs, XsBs)
```
e. Prefix of a suffix using append
```
sublist(Xs, AsXsBs) :-
    append(AsXs, Bs, AsXsBs), append(As, Xs, AsXs)
```
Examples (9): Prefix, member and adjacent using append

prefix(Xs, Ys) :- append(Xs, As, Ys).
suffix(Xs, Ys) :- append(As, Xs, Ys).
member(X, Ys) :- append(As, [X|Xs], Ys).
adjacent(X, Y, Zs) :- append(As, [X, Y|Ys], Zs).

Examples (10): Reversing a list

reverse(List, Tsil) :- Tsil is the result of reversing List.
a. Naïve reverse
reverse([], []) .
reverse([X|Xs], Zs) :- reverse(Xs, Ys), append(Ys, [X], Zs).
b. Reverse-accumulate
reverse(Xs, Ys) :- reverse(Xs, [], Ys).
reverse([X|Xs], Acc, Ys) :- reverse(Xs, [X|Acc], Ys).
reverse([], Ys, Ys).

Unification

- Unification is a (slightly) more general form of pattern matching. In that pattern variables can appear in both the pattern and the target.
- The following summarizes how unification works:
  1. a variable and any term unify
  2. two atomic terms unify only if they are identical
  3. two complex terms unify if they have the same functor and their arguments unify.
**Prolog Search Trees Summary**

1. Goal Order affects solutions
2. Rule Order affects solutions
3. Gaps in Goals can creep in
4. More advanced Prolog programming manipulates the searching

**Sublists (Goal Order)**

- Two definitions of S being a sublist of Z use:
  
  myappend([], Y, Y).
  
  myappend([H|X], Y, [H|Z]) :- myappend(X, Y, Z).

  &
  myprefix(X, Z) :- myappend(X, Y, Z).
  
  mysuffix(Y, Z) :- myappend(X, Y, Z).

  Version 1
  
  sublist1(S, Z) :- myprefix(X, Z), mysuffix(S, X).

  Version 2
  
  sublist2(S, Z) :- mysuffix(S, X), myprefix(X, Z).

  Version 3
  
  sublist3(S, Z) :- mysuffix(Y, Z), myprefix(S, Y).

**Watch them work:**

```
| ?- sublist.
consulting...sublist.plyes
| ?- sublist([a], [a,b,c]).

no
| ?- sublist2([a], [a,b,c]).

Fatal Error: global stack overflow ...
```
• So what’s happening? If we ask the question:
  
  sublist1([e], [a,b,c]).
  
  this becomes
  
  prefix(X,[a,b,c]), suffix([e],X).
  
  and using the guess-query idea we see that the first
goal will generate four guesses:
  
  []  [a]  [a,b]  [a,b,c]
  
  none of which pass the verify goal, so we fail.

• On the other hand, if we ask the question:
  
  sublist2([e], [a,b,c])
  
  this becomes
  
  suffix([e],X),prefix(X,[a,b,c]).
  
  using the guess-query idea note:
  
  Goal will generate an infinite number of guesses.
  
  [e]  [_,e]  [_,_,e]  [_,_,_,e]  [_,_,_,_,e]  
  
  ....
  
  None of which pass the verify goal, so we never terminate

• Rule Order affects Solutions

• Compare the two versions of append

append([], Y, Y).
append([H|X], Y, [H|Z]) :- append(X,Y,Z).

append2([], Y, Y) :- append2(X,Y,Z).
append2([H|X], Y, [H|Z]) :- append2(X,Y,Z).
append2([], Y, Y).
Prolog

Watch it:

?- [append].
consulting....append.pl
   yes
?- append(X,[c],Z).
   X = [] Z = [c]?
   X = [A] Z = [A,c]?
   X = [A,B] Z = [A,B,c]?
   yes
?- append2(X,[c],Z).
   Fatal Error: local stack overflow ...

Prolog Search Trees

- Sometimes Prolog Tries TOO HARD
- The Programs:
  fac(0,1).
  fac(N,M) :- N1 is N-1, fac(N1,M1), M is N * M1.
- The Problem:
  |- ?- [fac]. consulting....fac.pl
     yes
  |- ?- fac(4,X).
     X = 24?
     yes
  |- ?- fac(4,X), X=55.
     Fatal Error: local stack overflow ...
- What's Happening??

What's Happening?

(trace)
? - fac(2,X), X=55.
  1 1 Call: fac(2,_33) ?
  2 2 Call: _33 is 2-1 ?
  3 2 Exit: 1 is 2-1 ?
  4 3 Call: fac(1,_128) ?
  5 3 Exit: fac(1,1) ?
  6 3 Call: _128 is 1-1 ?
  7 3 Exit: 0 is 1-1 ?
  8 3 Call: fac(0,_170) ?
  9 3 Exit: fac(0,1) ?
  10 3 Call: _198 is 1*1 ?
  11 3 Exit: 1 is 1*1 ?
  12 3 Call: fac(1,_11) ?
  13 3 Exit: fac(1,1) ?
  14 3 Exit: 0 is 1*2 ?
  15 3 Call: _15 is 1*2 ?
  16 3 Exit: 2 is 1*2 ?
  17 3 Exit: fac(2,2) ?
**What's Happening?**

8 1 Call: 2=55 ?
8 1 Fail: 2=55 ?
1 1 Redo: fac(2,2) ?
3 2 Redo: fac(1,1) ?
5 3 Redo: fac(0,1) ?
6 4 Call: \_197 \_is \_0\_1 ?
6 4 Exit: \_1 \_is \_0\_1 ?
7 4 Call: fac(-1,\_222) ?

**Solution to this Problem: The Cut**

- **A cut**, written

  !

  is a goal that always succeeds

  when reached it alters the subsequent search tree.

**Cut Defined**

B := C_1, ... C_K, !, C_{K+1}, ... C_J.

- When applied during a search, and the cut ! is reached, then if at some later stage this rule fails, then Prolog backtracks past the

  C_K, ... C_1, B

  without considering anymore rules for them.

- In particular, this and any subsequent rule for B will not be tried.
Cut Example

- So using the cut in the guess-verify clause style:

  conclusion(S) :- guess(S), !, verify(S).

  eliminates all but the first guess

The solution for Factorial:

- Want to stipulate "Once you've found the solution as1 for fac(0,A), don't look for any others."

  fac(0,1) :- !.
  fac(N,M) :- N1 is N-1, fac(N1,M1), M is N * M1.

Negation as Unsatisfiability

- Consider the predicate can_marry(X,Y) meaning that X can legally marry Y provided that they are no more closely related than first cousins.

  can_marry(X,Y) :- X\=Y, nonsibling(X,Y),
                noncousin(X,Y).

  where

  nonsibling(X,Y):-X=Y.
  nonsibling(X,Y):-mother(M1,X),mother(M2,Y),M1\=M2.
  nonsibling(X,Y):-father(F1,X),father(F2,Y),F1\=F2.

  but

  | ?- nonsibling(albert,alice).
  | no
What’s Going On?

- For nonsibling(X,Y) to be true, it must be that X (and Y) have a common parent. However, albert has no parents stated as facts in the database.
- Therefore, nonsibling(albert,Y) fails.
- **Prolog uses a Closed World Model.**
- We might try writing
  
  ```prolog
  nonsibling(X,Y) :- no_parent(X).
  nonsibling(X,Y) :- no_parent(Y).
  ```
- But how can we express the *absence* of a fact in the database? We could add
  ```prolog
  no_parent(albert).
  ```
- But that’s tedious.

What’s Going On?

- We’d like to define nonsibling(X,Y) to be true whenever sibling(X,Y) is false, but something like
  ```prolog
  nonsibling(X,Y) :- \ sibling(X,Y).
  ```
- is non-Horn

“Solution”

- The apparent solution in Prolog is to use the “cut-fail” combination to simulate negation.
  ```prolog
  nonsibling(X,Y) :- sibling(X,Y), !, fail.
  nonsibling(X,Y).
  ```
- so
  ```prolog
  nonsibling(jeffrey,george)
  ```
  will fail, since sibling(jeffrey,george) succeeds. ! succeeds, and fail fails (but ! prevents any further backtracking).
  ```prolog
  nonsibling(albert,alice)
  ```
  will succeed, since sibling(albert,alice) fails, causing the first rule to fail; then the second rule succeeds (always).
Generalized “not” predicate

- Define
  
  not(A) :- call(A), !, fail.
  not(_).

  SO
  
  can_marry(A, B) :- A \ne B,
  not(sibling(A, B)),
  not(cousin(A, B)).

- But caution -- not doesn’t behave exactly like logical negation. It merely means “not satisfiable”, i.e. “unable to be proved true.”