Lists

We can represent lists as binary trees where list elements are left children

\[ \text{'.'(1, '.')(2, empty list)} \]

Is the list [1,2]

(how to represent the empty list?)

Lists

The empty list is the atom []

\[ \text{'.'(1, '.')(2, [])} \]

Is the list [1,2][] which is the list [1,2]

:- [1,2][] = [1,2].

:- display([1,2][]).

Write the following procedures

- car(?List, ?Head)
  - Succeeds if Head is the first element of List (the head)
- cdr(?List, ?Tail)
  - Succeeds if Tail is the sub-list of List without its first element (the tail)
- cons(?List, ?Head, ?Tail)
  - Succeeds if Head is the first element of List (the head) and Tail is the sub-list of List without its first element (the tail)

Implement a Stack

- push(?Element, +OldStack, -NewStack)
- pop(+OldStack, -NewStack, -Element)
- top(+Stack, -Element)
- empty(+Stack)
List Manipulation

- `member(?Term, ?List)`
  - Succeeds if `Term` unifies with a member of the list `List`.
- `delete(?Element, ?List1, ?List2)`
  - Succeeds if `List2` is `List1` less an occurrence of `Element` in `List1`.
- `append(?List1, ?List2, ?List3)`
  - Succeeds if `List3` is the result of appending `List2` to `List1`.
- `reverse(+List, ?Reversed)`
  - Succeeds if `Reversed` is the reversed list `List`.
- `length(?List, ?N)`
  - Succeeds if the length of list `List` is `N`.

Recursion

% my_length1.pl
my_length([], 0).
my_length([Head|Tail], Count) :-
  my_length(Tail, Sum),
  Count is Sum + 1.

% my_length2.pl
my_length(L, Total):- my_length(L, 0, Total).
sub_my_length([], Total, Total).
sub_my_length([Head|Tail], Sum, Total) :-
  Count is Sum + 1,
  sub_my_length(Tail, Count, Total).

Tail recursion

% my_length2.pl
my_length(L, Total):- my_length(L, 0, Total).
sub_my_length([], Total, Total).
sub_my_length([Head|Tail], Sum, Total) :-
  Count is Sum + 1,
  sub_my_length(Tail, Count, Total).

Naïve reverse

reverse_naive([], []).
reverse_naive([Head|Tail1], Reversed) :-
  reverse_naive(Tail1, Tail2),
  append(Tail2, [Head], Reversed).

Reverse with Accumulator and Tail Recursion

:- reverse_acc(List, [], Tsil).
reverse_acc([], Reversed, Reversed).
reverse_acc([Head|Tail], Rest, Reversed) :-
  reverse_acc(Tail, [Head|Rest], Reversed).
Generalizing Accumulators: Difference Structures

Probably one of the most ingenious programming techniques ever invented (by Sten-Åke Tärnlund, 1975) yet neglected by mainstream computer science.

A way of using variables as ‘holes’ in data structures that are filled in by unification as the computation proceeds.

Variables and Unification

\[- \text{test([a, b, c, X], X, Y)} = \text{test(Z, d, Z)}\]

\[Z = [a, b, c, X]\]
\[X = d\]
\[Y = Z = [a, b, c, d]\]

Variables and Unification

\[- \text{test([a, b | L1], L1, L2)} = \text{test(L3, [c, d], L3)}\]

\[L3 = [a, b] L1\]
\[L1 = [c, d]\]
\[L2 = L3 = [a, b, c, d]\]

Difference Lists

We represent the list \([1, 2, 3]\) by the difference:

\[\text{[1, 2, 3|L1} – L1\]

This is conceptual, this is only a structure and no difference is computed

Appending Difference Lists

we want to append list

\[L1 – LL1 = [a, b, c | LL1} – LL1\]

With

\[L2 – LLL2 = [d, e, f | LL2} – LL2\]

And get

\[L3 – LL3 = [a, b, c | [d, e, f | LL3}] – LL3\]

Appending Difference Lists

/* diff.pl */

diff_append(L1 – L2, L2 – LL3, L1 – LL3).
Member of Difference Lists

/* diff.pl */
member(_, L1 – L1) :- !, fail.
member(X, [X] – _).
member(X, [_|L1] – L2):-
    member(X, L1 – L2).

% need occur check, why?