Accumulators & Difference Lists

York University CSE 3401

Vida Movahedi
Overview

• Accumulators
  – Length of a list
  – Sum of list of numbers
  – Reverse a list
  – Factorial
  – Parts problem

• Difference Lists
  – Parts problem
  – Reverse a list

[ref.: Clocksin- Chap.3 and Nilsson- Chap. 7]
[also Prof. Gunnar Gotshalks’ slides]
Accumulators

• Useful when we calculate a result depending on what we find while traversing a structure, e.g. a list

• Example: Finding the length of a list
  Example: listlen([a, b, c], 3)

• Without accumulator:
  listlen([], 0).
  listlen([X|L], N) :- listlen(L, N1), N is N1 + 1.
  – Recursively makes the problem smaller, until list is reduced to empty list
  – On back substitution, the counter is added up.
Accumulators (cont.)

Without accumulators:
C0: \text{listlen}([], 0).
C1: \text{listlen}([X|L], N) :- \text{listlen}(L, N1), N \text{ is } N1 + 1.

Recursive search:
G0: :- \text{listlen}([a,b,c], N). \quad \text{Resolve with } C1, \ N \text{ is } N1+1
G1: :- \text{listlen}([b,c], N1). \quad \text{Resolve with } C1, \ N1 \text{ is } N2+1
G2: :- \text{listlen}([c], N2). \quad \text{Resolve with } C1, \ N2 \text{ is } N3+1
G3: :- \text{listlen}([], N3). \quad \text{Resolve with } C0, [N3/0]

Back substitution:
N2=N3+1=1 \quad N1=N2+1=2 \quad N=N1+1=3
Accumulators (cont.)

• With accumulator:

\[
\text{listlen}(L,N) :- \text{lenacc}(L, 0, N).
\]

\[
\text{lenacc}([], A, A).
\]

\[
\text{lenacc}([H|T], A, N) :- A1 \text{ is } A+1, \text{lenacc}(T, A1, N).
\]

• Predicate \( \text{lenacc}(L, A, N) \) is true if the length of \( L \) when added to \( A \) is \( N \).
  
  – Example:
  \[
  \text{lenacc}([a,b,c], 2, 5).
  \]
  \[
  \text{lenacc}([a,b,c], 0, 3).
  \]
Accumulators (cont.)

With accumulators:

C0: \texttt{listlen}(L,N) :- \texttt{lenacc}(L, 0, N).

C1: \texttt{lenacc}([], A, A).

C2: \texttt{lenacc}([H|T], A, N):- A1 is A+1, \texttt{lenacc}(T, A1, N).

Recursive search:

G0: :- \texttt{listlen}([a,b,c], N). Resolve with C0

G1: :- \texttt{lenacc}([a,b,c], 0, N). Resolve with C2, [A_1/0], A_1 is 1.

G1: :- \texttt{lenacc}([b,c], 1, N). Resolve with C2, [A_2/1], A_2 is 2.

G2: :- \texttt{lenacc}([c], 2, N). Resolve with C2, [A_3/2], A_3 is 3.

G3: :- \texttt{lenacc}([], 3, N). Resolve with C1, [A_4/3, N/3]

N=3

No Back substitution!
Sum of a list of numbers

• Without accumulator:
  sumList([], 0).
  sumList([H|T], N):- sumList(T, N1), N is N1+H.

  For a query such as :- sumlist([1, 2, 3], N).
  1) Recursive search until reduced to empty list
  2) Back substitution to calculate N

• With accumulator
  sumList(L, N):- sumacc(L, 0, N).
  sumacc([], A, A).
  sumacc([H|T], A, N):- A1 is A+H, sumacc(T, A1, N).
Accumulators- with vs. without

- Without accumulator:
  - Implements **recursion**
  - Counts (or builds up the final answer) on back substitution
  - Can be expensive, or explosive!

- With accumulator:
  - Implements **iteration**
  - Counts (or builds up the final answer) on the way to the goal
  - Accumulator (A) changes from nothing to the final answer
  - The final value of the goal (N) does not change until the last step
Reverse a list- recursion vs. iteration

- Without accumulator (O(n^2)):
  
  ```
  reverse([], []).
  reverse([X|L], R) :- reverse(L, L1), append (L1, [X], R).
  ```

- With accumulator (O(n)):
  
  ```
  reverse(L, R): revacc(L, [], R).
  revacc([], A, A).
  revacc([H|T], A, R) :- revacc(T, [H|A], R).
  ```

  ```
  :- reverse([a,b,c], R).  =>  :- revacc([a,b,c], [], R).
  :- revacc([b,c], [a], R).  =>  :- revacc([c], [b,a], R).
  :- revacc([], [c,b,a], R).  =>  R=[c,b,a]
  ```
Factorial- recursion vs. iteration

- Recursive definition:
  
  \[ \text{factr}(0, 1). \]
  
  \[ \text{factr}(N, F) :\text{ N1 is N-1, factr}(N1, F1), F \text{ is } N\times F1. \]

- For a query such as :- \text{factr}(5, F).
  
  (1) Recursive search reduces problem to the boundary condition (factorial of 0)
  
  (2) Back substitution calculates final answer.

- For a query such as :- \text{factr}(N, 120) or :- \text{factr}(N,F).
  
  Cannot do the arithmetic! Right side of ‘is’ is undefined.
Factorial - recursion vs. iteration

• Iterative definition:

  \[ \text{facti} (N, F) :- \text{facti} (0, 1, N, F). \]
  \[ \text{facti} (N, F, N, F). \]
  \[ \text{facti} (I, Fi, N, F) :- \text{invariant} (I, Fi, J, Fj), \text{facti} (J, Fj, N, F). \]
  \[ \text{invariant} (I, Fi, J, Fj) :- J \text{ is } I + 1, Fj \text{ is } J \times Fi. \]

• First two arguments are accumulators

• Right hand side of ‘is’ is defined for queries such as \text{facti}(N, 120) and \text{facti}(N, F).

<table>
<thead>
<tr>
<th>I</th>
<th>Fi</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>24</td>
</tr>
<tr>
<td>5</td>
<td>120</td>
</tr>
</tbody>
</table>
• Assume we have a database of assemblies required for a bike, for example:

  assembly(bike, [wheel, wheel, frame]).
  assembly(frame, [rearframe, frontframe]).
  assembly(frontframe, [fork, handle]).

  ....
  basicpart(rearframe).
  basicpart(fork).

  ....
To find the parts to assemble a bike, we can write:

\[
\text{partsof}(X, [X]) :- \text{basicpart}(X).
\]
\[
\text{partsof}(X, P) :- \text{assembly}(X, \text{Subparts}), \text{partsofList}(	ext{Subparts}, P).
\]
\[
\text{partsofList}([], []). \quad \text{partsofList}([\text{Head} | \text{Tail}], P) :- \text{partsof}(	ext{Head}, \text{Headparts}),
\text{append} \left( \text{Headparts}, \text{Tailparts}, P \right).
\]

- Expensive computation
- Also wasteful (e.g. finding parts of a ‘wheel’ twice)
Parts Problem (cont.)

- We can use an accumulator to avoid extra work:

  ```prolog
  partsof(X, P) :- partsacc(X, [], P).
  partsacc(X, A, [X|A]) :- basicpart(X).
  partsacc(X, A, P):- assembly(X, Subparts), partsacclist(Subparts, A, P).
  partsacclist([], A, A).
  partsacclist([H|Tail], A, P):- partsacc(H, A, Headparts), partsacclist(Tail, Headparts, P).
  ```

Note:
- `partacc(X, A, P)` means: parts of `X` added to list `A` results in list `P`.
- `partsacclist(L, A, P)` means: parts of elements in `L` added to list `A` results in list `P`.
partsof(X, [X]) :- basicpart(X).
partsof(X, P) :- assembly(X, Subparts), partsofList(Subparts, P).
partsofList([], []).
partsofList([Head | Tail], P) :-
    partsof(Head, Headparts),
    partsofList(Tail, Tailparts),
    append(Headparts, Tailparts, P).

partsof(X, P) :- partsacc(X, [], P).
partsacc(X, A, [X | A]) :- basicpart(X).
partsacc(X, A, P) :- assembly(X, Subparts), partsacclist(Subparts, A, P).
partsacclist([], A, A).
partsacclist([H | Tail], A, P) :-
    partsacc(H, A, Headparts),
    partsacclist(Tail, Headparts, P).
Example

:- partof(frame, P).
:- partsacc(frame, [], P).

...

:- partsacclist([rearframe, frontframe], [], P).
:- partsacc(rearframe, [], Hp), partsacclist([frontframe], Hp, P).

... Hp/[rearframe]
:- partsacclist([frontframe], [rearframe], P).
:- partsacc(frontframe, [rearframe], Hp1), partsacclist([], Hp1, P).

...

:- partsacclist([fork, handle], [rearframe], Hp1), partsacclist([], Hp1, P).
:- partsacc(fork, [rearframe], Hp2), partsacclist([handle], Hp2, Hp1),
    partsacclist([], Hp1, P).

... Hp2/[fork, rearframe]
:- partsacclist([handle], [fork, rearframe], Hp1), partsacclist([], Hp1, P).

... Hp1/[handle, fork, rearframe]
:- partsacclist([], [handle, fork, rearframe], P)

=> P/ [handle, fork, rearframe]
Difference Lists

• But the list is in reverse order!

• Here is a way to get the part list in the correct order:

\[
\text{partsof}(X, P) :- \text{partsdif}(X, [], P).
\]

\[
\text{partsdif}(X, \text{Hole}, [X|\text{Hole}]) :- \text{basicpart}(X).
\]

\[
\text{partsdif}(X, \text{Hole}, P) :- \text{assembly}(X, \text{Subparts}),
\]

\[
\text{partsdiflist}(\text{Subparts, Hole, P}).
\]

\[
\text{partsdiflist}([], \text{Hole, Hole}).
\]

\[
\text{partsdiflist}([H|\text{Tail}], \text{Hole, P}) :- \text{partsdif}(H, \text{Hole1, P}),
\]

\[
\text{partsdiflist}(\text{Tail, Hole, Hole1}).
\]
partsof(X, P) :- partsacc(X, [], P).
partsacc(X, A, [X|A]) :- basicpart(X).
partsacc(X, A, P):- assembly(X, Subparts), partsacclist(Subparts, A, P).
partsacclist([], A, A).
partsacclist([H|Tail], A, P):- partsacc(H, A, Headparts),
partsacclist(Tail, Headparts, P).

partsof(X, P) :- partsdif(X, [], P).
partsdif(X, Hole, [X|Hole]) :- basicpart(X).
partsdif(X, Hole, P):- assembly(X, Subparts), partsdiflist(Subparts, Hole, P).
partsdiflist([], Hole, Hole).
partsdiflist([H|Tail], Hole, P):- partsdif(H, Hole1, P),
partsdiflist(Tail, Hole, Hole1).
Example

:- partof(frame, P).
:- partsdif(frame, [], P).

... 

:- partsdiflist([rearframe, frontframe], [], P).
:- partsdif(rearframe, Hole1_1, P), partsdiflist([frontframe], [], Hole1_1).

... P/[rearframe|Hole1_1]

:- partsdiflist([frontframe], [], Hole1_1).
:- partsdif(frontframe, Hole1_2, Hole1_1), partsdiflist([], [], Hole1_2).

... 

:- partsdiflist([fork, handle], Hole1_2, Hole1_1), partsdiflist([], [], Hole1_2).
:- partsdif(fork,Hole1_3, Hole1_1), partsdiflist([handle], Hole1_2, Hole1_3), partsdiflist([], [], Hole1_2).

... Hole1_1/[fork|Hole1_3]

:- partsdiflist([handle ], Hole1_2, Hole1_3), partsdiflist([], [], Hole1_2).
Example

\[-\text{partsdif}(\text{handle}, \text{Hole}_{14}, \text{Hole}_{13}), \text{partsdiflist}([], \text{Hole}_{12}, \text{Hole}_{14}),
\text{partsdiflist}([], [], \text{Hole}_{12}).\]

\[-\text{partsdiflist}([], \text{Hole}_{12}, \text{Hole}_{14}),
\text{partsdiflist}([], [], \text{Hole}_{12}).\]

\[-\text{Hole}_{14}/\text{Hole}_{12}\]
\[-\text{partsdiflist}([], [], \text{Hole}_{12}).\]
\[-\text{Hole}_{12}/[]\]

- **Back substitution:**

  \[P = [\text{rearframe}|\text{Hole}_{11}]\]

  \[=[\text{rearframe}, \text{fork}|\text{Hole}_{13}]\]

  \[=[\text{rearframe}, \text{fork}, \text{handle}|\text{Hole}_{14}]\]

  \[=[\text{rearframe}, \text{fork}, \text{handle}|\text{Hole}_{12}]\]

  \[=[\text{rearframe}, \text{fork}, \text{handle}][[]]\]

  \[=[\text{rearframe}, \text{fork}, \text{handle}]\]

  \[P/[\text{rearframe}|\text{Hole}_{11}]\]

  \[\text{Hole}_{11}/[\text{fork}|\text{Hole}_{13}]\]

  \[\text{Hole}_{13}/[\text{handle}|\text{Hole}_{14}]\]

  \[\text{Hole}_{14}/\text{Hole}_{12}\]

  \[\text{Hole}_{12}/[]\]
Difference List

• The idea in the previous code was to have a HOLE in the tail of the list to be instantiated later by Prolog.

• Why is it called a difference list?
  The name comes from list differences:
  \([a,b,c,d,e] - [d,e] = [a,b,c]\)
  \([a,b,c|X] - X = [a,b,c]\)

\([\text{L|Hole}] - \text{Hole} = \text{L}\) for any list L and any list assigned to Hole

• A list L is represented by the difference between another list in the form \([\text{L|Hole}]\) and one of its sublists (the tail of the list, Hole) that must be an unknown.
  – The empty list is represented by \(X-X\)
  – \([a,b,c]\) is represented by \([a,b,c|X]-X\)
Reverse using difference lists

reverse(X,Y) :- rev(X, Y-[]).

rev([], X-X).
rev([X|Y], Z-W) :- rev(Y, Z- [X|W]).

:- reverse([a,b], R).

  :- rev([a,b], R-[]).
  :- rev([b], R-[a]).
  :- rev([], R- [b,a]).
  \[R=[b,a]\]

- Can reverse a list of n elements in n+2 resolution steps.
Accumulators vs. Difference Lists

• Accumulators:
  – Are like **stacks**
  – They can eliminate the back substitution step.
  – Can be used to lower complexity

• Difference Lists:
  – Are like **queues**
  – Can be used to preserve order of elements
  – Can be used to lower complexity