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: listlen([], 0).

C1: listlen([X|L], N) :- listlen(L, N1), N is N1 + 1.

Recursive search:

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

Back substitution:

N2=N3+1=1 N1=N2+1=2 N=N1+1=3

Accumulators (cont.)

- With accumulator:

```
listlen(L,N) :- lenacc(L, 0, N).
lenacc([], A, A).
lenacc([H|T], A, N):- A1 is A+1, lenacc(T, A1, N).
```

- Predicate lenacc(L, A, N) is true if the length of L when added to A is N.
  - Example:

  lenacc([a,b,c], 2, 5).
  lenacc([a,b,c], 0, 3).

Accumulators (cont.)

With accumulators:

C0: *listlen(L,N) :- lenacc(L, 0, N).*

C1: lenacc([], A, A).

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

Recursive search:

G0: :- listlen([a,b,c], N).	Resolve with C0
G1: :- lenacc([a,b,c], 0, N).	Resolve with C2, [A_1/0], $A1_1$ is 1.
G1: :- lenacc([b,c], 1, N).	Resolve with C2, [A_2/1], $A1_2$ is 2.
G2: :- lenacc([c], 2, N).	Resolve with C2, [A_3/2], $A1_3$ is 3.
G3: :- 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 **<u>recursion</u>**
  - Counts (or builds up the final answer) on back substitution
  - Can be expensive, or explosive!
- With accumulator:
  - Implements **<u>iteration</u>**
  - 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:

  factr(0, 1).

  factr(N, F) :- N1 is N-1, factr(N1, F1), F is N*F1.

- For a query such as :- 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 :-factr(N, 120) or :-factr(N,F).

  Cannot do the arithmetic! Right side of 'is' is undefined.

Factorial- recursion vs. iteration

- Iterative definition:

facti (N ,F) :- facti (0, 1, N, F).
facti (N, F, N, F).
facti (I, Fi, N, F) :- invariant (I, Fi, J, Fj), facti (J, Fj, N, F).
*invariant (I, Fi, J, Fj) :- J is I + 1, Fj is J * Fi.*

- *First two arguments are accumulators*

- *Right hand side of 'is' is defined for queries such as :-facti(N, 120) and :-facti(N,F).*

I	Fi
0	1
1	1
2	2
3	6
4	24
5	120

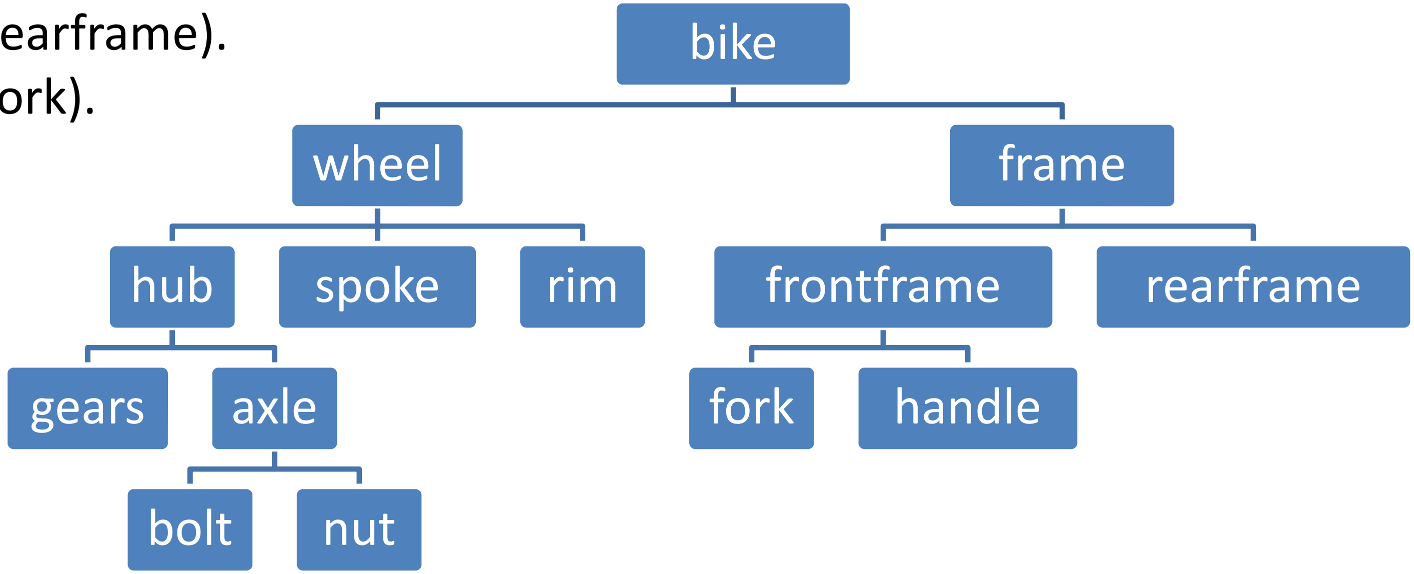
invariant(0,1,1,1)

invariant(3,6,4,24)

Parts Problem

- 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).
....
```



Parts Problem (cont.)

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

```
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).
```

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

Parts Problem (cont.)

- We can use an accumulator to avoid extra work:

```
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.

Compare!

```
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:

```
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).
```

Compare!

```
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

:- partsdif(handle, $Hole1_4$, $Hole1_3$), partsdiflist([], $Hole1_2$, $Hole1_4$), partsdiflist([], [],$Hole1_2$).

... $Hole1_3$/[handle|$Hole1_4$]

:- partsdiflist([], $Hole1_2$, $Hole1_4$), partsdiflist([], [],$Hole1_2$).

$Hole1_4$/$Hole1_2$

:- partsdiflist([], [],$Hole1_2$).

$Hole1_2$/[]

- Back substitution:

P	= [rearframe|$Hole1_1$]	P/[rearframe|$Hole1_1$]
	=[rearframe, fork|$Hole1_3$]	$Hole1_1$/[fork|$Hole1_3$]
	=[rearframe, fork, handle|$Hole1_4$]	$Hole1_3$/[handle|$Hole1_4$]
	=[rearframe, fork, handle|$Hole1_2$]	$Hole1_4$/$Hole1_2$
	=[rearframe, fork, handle|[]]	$Hole1_2$/[]
	=[rearframe, fork, handle]	

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]

  [L|Hole]- Hole= 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 [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