Accumulators & Difference Lists

York University
Department of Computer Science and Engineering
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]
Using 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.
(without) Accumulators

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

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

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

• With accumulator:

\[\text{\textit{listlen}}(L,N) \iff \text{\textit{lenacc}}(L, 0, N).\]
\[\text{\textit{lenacc}}([], A, A).\]
\[\text{\textit{lenacc}}([H\mid T], A, N) :\text{ A1 is A+1, } \text{\textit{lenacc}}(T, A1, N).\]

• ‘A’ is \textbf{length accumulated so far}.

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

With accumulators:
C0: \text{listlen}(L,N) :- \text{lenacc}(L, 0, N).
C1: \text{lenacc}([], A, A).
C2: \text{lenacc}([H|T], A, N) :- A1 is A+1, \text{lenacc}(T, A1, N).

Recursive search:

Resolve with C0
\text{Resolve with C2, [A_1/0], A1 is 1.}
\text{Resolve with C2, [A_2/1], A1 is 2.}
\text{Resolve with C2, [A_3/2], A1 is 3.}
\text{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).

- ‘A’ is sum accumulated so far.
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)):
  \( \text{reverse}([], []). \)
  \( \text{reverse}([X|L], R) :- \text{reverse}(L, L1), \text{append}(L1, [X], R). \)

• With accumulator (O(n)):
  \( \text{reverse}(L, R) : \text{revacc}(L, [], R). \)
  \( \text{revacc}([], A, A). \)
  \( \text{revacc}([H|T], A, R) :- \text{revacc}(T, [H|A], R). \)

• ‘A’ is reversed list accumulated so far.

  \( \text{- reverse}([a,b,c], R). \quad \Rightarrow \quad \text{- revacc}([a,b,c], [], R). \)
  \( \text{- revacc}([b,c], [a], R). \quad \Rightarrow \quad \text{- revacc}([c], [b,a], R). \)
  \( \text{- revacc}([], [c,b,a], R). \quad \Rightarrow \quad R=[c,b,a] \)
Factorial- recursion vs. iteration

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

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

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

• Iterative definition:

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

• First two arguments are accumulators, and show row of factorial table calculated so far.

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

<table>
<thead>
<tr>
<th></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>