Accumulators
More on Arithmetic
and
Recursion

listlen (L , N)

- **L** is a list of length **N** if ...

  listlen ([] , 0).

  listlen ([H | T] , N) :- listlen (T , N1) , N is N1 + 1.

  > **On searching for the goal, the list is reduced to empty**
  >
  > **On back substitution, once the goal is found, the counter is incremented from 0**

- Following is an example sequence of goals (**left hand column)** and back substitution (**right hand column**)

listlen([a, b, c] , N).	**N <== N1 + 1**
listlen([b, c] , N1).	**N1 <== N2 + 1**
listlen([c] , N2).	**N2 <== N3 + 1**
listlen([] , N3).	**N3 <== 0**

Abstract the counter

- The following abstracts the counter part from listlen.

  ```
  addUp ( 0 ).
  addUp ( C )  :-  addUp ( C1 ) , C is C1 + 1.
  ```

- Notice the recursive definition occurs on a counter one smaller than in the head.

Count Up

- ◊ An alternate method is to count on the way to the fixed point value in the query
- ◊ The auxiliary counter accumulates the result on the way to the goal.

```
adder ( C )  :-  adder ( 0 , C ).  ;Introduce auxiliary counter
adder ( C , C )  :-  nl , write ( 'a ' ).
```

> The goal is reached when the auxiliary counter reaches the fixed point count value

```
adder ( Acc1 , C )  :-  write ( 'b ' ) , Acc2 is Acc1 + 1
                                      , adder ( Acc2 , C ).
```

> The predicates in black always succeed, side effect is to write to the terminal – can see order of rule execution

listLen(L,N) – 2

◊ We can define list length using an accumulator

```
listln ( L , N )  :-  lenacc ( L , 0 , N ).
```

> Introduce the auxiliary counter – length of list L when added to the accumulator is N

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

◊ Following is a sequence of goals

```
listln  ( [ a , b , c ] ,  N ).
lenacc ( [ a ,  b , c ] , 0 , N ).    N <== N1
lenacc ( [ b , c ] , 1 , N1 ).        N1 <== N2
lenacc ( [ c ] , 2 , N2 ).            N2 <== N3
lenacc ( [] , 3 , N3 ).               N3 <== 3
```

Accumulator – Using vs Not Using

- The definition styles reflect two alternate definitions for counting
  - » **Recursion – counts (accumulates) on back substitution.**
    - > **Goal becomes smaller problem**
    - > **Do not use accumulator**
  - » **Iteration – counts up, accumulates on the way to the goal**
    - > **Accumulate from nothing up to the goal**
    - > **Goal "counter value" does not change**
- Some problems require an accumulator
  - » **see parts assembly**

Factorial using recursion

◊ Following is a recursive definition of factorial

Factorial (N) = N * Factorial (N – 1)

```
factr ( N , F)  -- F is the factorial of N
factr ( 0 , 1 ).
factr ( N , F )  :-  J  is  N – 1  ,   factr ( J , F1 )
                     , F  is  N * F1.
```

◊ The problem **(J , F1)** is a smaller version of **(N , F)**

◊ Work toward the fixed point of a trivial problem

◊ Does not work for **factr (N ,120)** and **factr (N , F)**.

» **Cannot do arithmetic J is N – 1 because N is undefined.**

Factorial using iteration – accumulators

◊ An iterative definition of factorial

```
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.
```

◊ The last two arguments are the goal and they remain the same throughout.

◊ The first two arguments are the accumulator and they start from a fixed point and accumulate the result

◊ Works for queries **factr (N ,120)** and **factr (N , F)** because values are always defined for the **is** operator.

Fibonacci – Ordinary Recursion

◊ Following is a recursive definition of the fibonacci series. For reference here are the first few terms of the series

Index –	0	1	2	3	4	5	6	7	8	9	10	11	12
Value –	1	1	2	3	5	8	13	21	34	55	89	144	233

**Fibonacci (N) = Fibonacci (N – 1)
+ Fibonacci (N – 2).**

```
fib ( 0 , 1 ).
fib ( 1 , 1 ).
fib ( N , F )  :-  N1  is  N – 1 ,  N2  is  N – 2
                ,  fib ( N1 , F1 ) , fib ( N2 , F2 )
                ,  F  is  F1 + F2.
```

◊ Does not work for queries **fib (N , 8)** and **fib (N , F)**

» **Values for is operator are undefined.**

Fibonacci – Tail Recursion

◊ A tail recursive definition of the fibonacci series

> Tail recursion is equivalent to iteration

```
fibt ( 0 , 1 ).
fibt ( 1 , 1 ).
fibt ( N , F )  :-  fibt ( 2 , 1 , 1 , N , F ).

fibt ( N , Last2 , Last1 , N , F )  :-  F  is  Last2 + Last1.
fibt ( I , Last2 , Last1 , N , F )  :-  J  is  I + 1
                                      ,  Fi  is  Last2 + Last1
                                      ,  fibt ( J , Last1 , Fi , N , F ).
```

◊ Works for queries **factr (N , 120)** and **factr (N , F)**

» **values are always defined for is operator.**

Parts Assembly – The Problem 1

◊ Parts assembly is the problem of accumulating all the parts for a product from a definition of the components of each part

◊ Consider a bicycle we could have

> the following basic components

basicPart(spokes). basicPart(rim). basicPart(tire).

basicPart(inner_tube). basicPart(handle_bar).

basicPart(front_ fork). basicPart(rear_fork).

> the following definitions for sub assemblies

assembly(bike, [wheel, wheel, frame]).

assembly(wheel, [spokes, rim, wheel_cushion]).

assembly(wheel_cushion, [inner_tube, tire]).

assembly(frame, [handle_bar, front_fork, rear_fork]).

Parts Assembly – The Problem 2

- We are interest in obtaining a parts list for a bicycle.

  [rear_ fork , front_ fork , handle_bar , tire , inner_tube , rim , spokes , tire , inner_tube , rim , spokes]

  > **We have two wheels so there are two tires, inner_tubes, rims and spokes.**

- Using accumulators we can avoid wasteful re-computation as in the case for the ordinary recursion definition of the fibonacci series

Parts Assembly – Accumulator 1

- partsof (X ,P) – **P** is the list of parts for item **X**
- partsacc (X , A , P) – parts_of (**X**) || **A** = **P**.

  partsof (X , P) :- partsacc (X , [] , P).

|| is catenate (math append)

> **Basic part – parts list contains the part**

partsacc (X , A , [X | A]) :- basicPart (X).

> **Not a basic part – find the components of the part**

partsacc (X , A , P) :- assembly (X , Subparts) ,

> **parsacclist – parts_of (Subparts) || A = P**

partsacclist (Subparts , A , P).

Parts Assembly – Accumulator 2

◊ parsacclist (ListOfParts , AccParts , P)

– parts_of (**ListOfParts**) || **AccParts** = **P**

> **No parts no change in accumulator**

partsacclist ([] , A , A).

partsacclist ([P | Tail] , A , Total) :-

> **Get the parts for the first on the list**

partsacc (P , A , HeadParts)

> **And catenate with the parts obtained from the rest of the ListOfParts**

, partsacclist (Tail , HeadParts , Total).

Difference Lists and Holes

- The accumulator in the parts assembly program is a stack
  - **Items are stored in the reverse order in which they are found**
- How do we store accumulated items in the same order in which they are formed?
  - **Use a queue**
- Difference lists with holes are equivalent to a queue

Examples for Holes

- ◊ Consider the following list

  [a , b , c , d | X]

  > **X is a variable indicating the tail of the list. It is like a hole that can be filled in once a value for X is obtained**

- ◊ For example

  Res = [a , b , c , d | X] , X = [e , f].

  > **Yields**

  Res = [a , b , c , d , e , f]

Examples for Holes – 2

◊ Or could have the following with the hole going down the list

Res = [a , b , c , d | X]

> more goal searching gives X = [e , f | Y]

> more goal searching gives Y = [h , i , j]

> Back substitution Yields

Res = [a , b , c , d , e , f , h , i , j]

PartsAssembly – Difference List 1

◊ partsofd (X , P) – **P** is the list of parts for item **X**

◊ partsdiff (X , Hole , P) – parts_of (**X**) || **Hole** = **P**

> **Hole and P are reversed compared to Clocksin & Mellish (v3, v4) to better compare with accumulator version.**

partsofd (X , P) :- partsdiff (X , [] , P).

> **Base case we have a basic part, then the parts list contains the part**

partsdiff (X , Hole , [X | Hole]) :- basicPart (X).

PartsAssembly – Difference List 2

> Not a base part, so we find the components of the part

partsdiff (X , Hole , P) :- assembly (X , Subparts)

> parsdifflistd – parts_of (Subparts) II Hole = P

, partsdifflist (Subparts , Hole , P).

PartsAssembly – Difference Lists 3

◊ parsdifflist (ListOfParts , Hole , P)
  – parts_of (**ListOfParts**) || **Hole** = **P**

partsdifflist ([] , Hole , Hole).

partsdifflist ([P | Tail] , Hole , Total) :-

> **Get the parts for the first on the list**

partsdiff (P , Hole1 , Total)

> **And catenate with the parts obtained from the rest of the ListOfParts**

, partsdifflist (Tail , Hole , Hole1).

Compare Accumulator with Hole

```
partsof  ( X , P ) :- partsacc  ( X , [] , P ).      Accumulator
partsofd ( X , P ) :- partsdiff ( X , [] , P ).      Difference/Hole

partsacc  ( X , A    , [ X | A ] )    :- basicPart ( X ).
partsdiff ( X , Hole , [ X | Hole ] ) :- basicPart ( X ).

partsacc  ( X , A    , P ) :- assembly ( X , Subparts )
                            , partsacclist ( Subparts , A , P ).

partsdiff ( X , Hole , P ) :- assembly ( X , Subparts )
                            , partsdifflist ( Subparts , Hole , P ).
```

Compare Accumulator with Hole – 2

```
partsacclist  ( [] ,    A    ,  A    ).
partsdifflist (  []  ,  Hole , Hole ).


partsacclist   ( [ P | Tail ] , A    , Total )
                   :-  partsacc  ( P ,  A    , HeadParts )
                     ,  partsacclist  ( Tail , HeadParts , Total ).

partsdifflist ( [ P | Tail ] , Hole , Total )
                  :-  partsdiff ( P ,  Hole1 ,    Total      )
                  ,   partsdifflist ( Tail  ,    Hole  ,  Hole1 ).
```