Accumulators

More on Arithmetic
and
Recursion

Shakil M. Khan
adapted from Gunnar Gotshalks

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

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 (i.e. result computed so far) 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
  - 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
  - parts explosion problem
  - summing a list of numbers

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

- 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 fibt (N , 120) and fibt (N , F)
  - values are always defined for is operator.

sum a list of numbers

- sumList(List, Total) asserts List is a list of numbers and Total = + / List
  - uses an accumulator
  - sumListA asserts (+ / List) + Acc = Total

```
sumList(List, Total) :- sumListA(List, 0, Total).
sumListA([],Acc, Acc).
sumListA([First|Rest], Acc, Total) :-
                    NewAcc is Acc + First,
                    sumListA(Rest, NewAcc, Total).
```


parts explosion – the problem 1

- parts explosion 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 explosion – the problem 2

- we are interested 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 explosion – accumulator 1

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

|| is append

```
partsof ( X , P ) :- partsacc ( X , [] , P ).
    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 explosion – accumulator 2

- partsacclist (ListOfParts , AccParts , P)
  - parts_of (ListOfParts) || AccParts = P

> no parts ➔ no change in accumulator

```
partsacclist ( [] , A , A ).
partsacclist ( [ Head | Tail ] , A , Total ) :-
```

> get the parts for the first on the list

```
partsacc ( Head , 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 explosion 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]

parts explosion – 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 (v5) 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).

parts explosion – 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) || Hole = P

  , partsdifflist (Subparts , Hole , P).

parts explosion – difference lists 3

- parsdifflist (ListOfParts , Hole , P)
  - parts_of (ListOfParts) || Hole = P

```
partsdifflist ( [] , Hole , Hole ).
partsdifflist ( [ Head | Tail ] , Hole , Total ) :-
```

- get the parts for the first on the list

```
          partsdiff ( Head , Hole1 , Total )
```

- and catenate with the parts obtained from the rest of the ListOfParts

```
          , partsdifflist ( Tail , Hole , Hole1 ).
```

- Hole1 is the "total" of Tail

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

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


Cut & Not

Shakil M. Khan
adapted from Gunnar Gotshalks

cut – !

- cut, the ! operator, is used to
  - not waste time on useless choices
    1. know that if current rule fails then trying further rules for the current predicate is useless
       – if you got this far then, this is the only rule to try
    2. stop after one solution – do not look for alternate solutions
    3. if you continue with this predicate, you will not find a solution
       – use of ! , fail
- cut commits to all choices made when entering parent goal – the predicate at the head of the rule
  » cannot be re-satisfied on backtracking

```
a :- b, c, !, d, e.
```


confirming choice of rule

- rule 2 for intersection has confirmation use of cut
  - intersection (A , B , C) – A ∩ B = C

```
intersection ( [] , B , [] ).
intersection ( [ Ah | At ] , B , [ Ah | Ct ] )
    :- member ( Ah , B ) , ! , intersection ( At , B , Ct ).
```

  - rule 2 is applicable when head (A) in B

```
intersection ( [ Ah | At ] , B , C )
    :- intersection ( At , B , C ).
```

  - rule 3 is applicable when head (A) not-in B
- once we have established that Ah is a member of B, then if we backtrack over the member predicate, there is no need to consider rule 3

stopping – found first solution

- consider the predicate sum_to (N ,T), where T is the sum of the integers 1 .. N

```
sum_to ( 1 , 1 ).
sum_to ( N , T ) :- N1 is N – 1 ,
                    sum_to ( N1 , T1 ) ,
                    T is T1 + N.
```

- the above program works as long as N is an integer >= 1
  - but there is only one solution, there is no point in trying rule 2 if rule 1 is ever satisfied
  - if ; return is used Prolog loops until memory is exhausted searching for a non existent second solution

stopping – found first solution – 2

- so introduce cut into the first case

```
sum_to ( 1 , 1 ) :- ! .
sum_to ( N , T ) :- N1 is N – 1 ,
                    sum_to ( N1 , T1 ) ,
                    T is T1 + N.
```

- now only one solution is found. Search terminates without infinite loop.
  - also example of choice of rule. Once rule 1 has been picked no point in trying rule 2

cut-fail in action

- consider the following

  avg_taxpayer(X) :- foreigner(X), !, fail.
  avg_taxpayer(X) :- …
- fail always fails
- definition makes use of cut fail to terminate when the person is a foreigner (even if that person has all other qualities of an average taxpayer, denoted by … above)

not

- when a rule has the following form

  head :- A , B , C , D.
- you can think of
  - A as being a guard to trying B, C, D
  - A, B as being a guard to trying C, D
  - A, B, C as being a guard to trying D
- for example the use of member (Ah , B) in the rule 2 for intersection

not – 2

- the predicate not (P) is used as a guard to select cases as in the following
  - Q ([H | T] , ...) :- not (H = [_ | _]) , P (H , ...) .
    > only try P if H does not have a head and tail
  - Q ([H | T] , ...) :- not (H = [ ]) , P (H , ...) .
    > only try P if H is not the empty list
  - Q ([H | T] , X , ...) :- not (H = X) , P (H , ...) .
    > only try P if H is not equal to X

not – definition

- not is not built into some Prologs (it is in SWI Prolog) as its interpretation depends upon what you want it to mean

Prolog searches are based on
closed universe
truth is relative to the database

- yes means the query can be satisfied by the database
- no means the query cannot be satisfied by the database
  - it does not mean the query is false, just unsatisfiable

not – definition

- the following is the definition of not as defined in utitlities.pro and in Clocksin & Mellish

```
not ( P ) :- call ( P ) , ! , fail.
not ( _ ) .
```

- rule 1 tries call (P)
  - call queries the database with the predicate P
  - analogous to eval in Lisp
- if the call succeeds, then the ! , fail combination says fail and do not try the second rule
  - so if P gives yes, then not (P) gives no
- if the call fails, then rule 2 is tried and always succeeds
  - so if P gives no, then not (P) gives yes

not definition – consequence

- the following shows that not as defined has side effects
  - a double negative is not equivalent to a positive!

cut & not equivalence

- cut and not can be used interchangeably with a change in rule structure
  - note the use of B as a guard

```
A :- B , C.                A :- B , ! , C.
A :- not ( B ) , D.        A :- D.
```

- if B succeeds then success or failure of A depends upon C
- if B fails, then success or failure of A depends upon D

cut is dangerous

- using cut we are taking advantage of the way Prolog searches the database
- consider the predicate number_of_parents (X , N)
  - X has N parents defined as follows

```
number_of_parents ( adam , 0 ) :- ! .
number_of_parents ( eve , 0 ) :- !.
number_of_parents ( X , 2 ).
```

- definition works correctly if we query such as the following when using ; return – the cut prevents finding extra solutions for adam and eve

```
number_of_parents ( adam , N ). ==> 0
number_of_parents ( eve , N ). ==> 0
number_of_parents ( wilhelma , N ). ==> 2
```

cut is dangerous – 2

- but fails on the following queries

  number_of_parents (adam , 2). ==> yes
  number_of_parents (eve , 2). ==> yes
- change the definition to

  number_of_parents (adam , N) :- ! , N = 0.
  number_of_parents (eve , N) :- ! , N = 0.
  number_of_parents (X , 2).
- or change the definition to

  number_of_parents(adam , 0) :- ! .
  number_of_parents(eve , 0) :- !.
  number_of_parents(X,2) :- X \= adam , X \= eve.
- still fail on queries such as the following, expecting backtracking to enumerate all the possibilities

  number_of_parents (Who , N).

cut is dangerous – moral

- if you introduce cuts to obtain correct behavior when the goals are of one form, there is no guarantee that anything sensible will happen if goals of another form start appearing

- it follows that it is only possible to use cut reliably if you have a clear policy about how your rules are going to be used. If you change this policy, all the uses of cut must be reviewed