Prolog Introduction (cont'd)

Clocksin & Mellish Ch 1 & 2

Shakil M. Khan
adapted from Gunnar Gotshalks

structures

- structures are a means of grouping a collection of other objects
  - structures are also called compound terms, or complex terms
  - the name of a structure is called a functor
  - the items within a structure are called components
- the general pattern is

```
functor ( component_1 , component_2 ,
            ...
          component_n )
```

structures (cont'd)

- components can also be structures – recursive definition
- e.g.:

```
functor ( functor1 ( comp1, comp2 ),
          component_2 ,
          ...
          component_n )
```


example structures

- books have authors and titles, so we could have

```
book ( dickens , great_expectations )
```

- people have books; in particular, Leila could have Great Expectations

```
has ( leila , book ( dickens , great_expectations ) )
```

- facts in Prolog are structures where the predicate is the functor of a structure and the arguments of the predicate are the components of the structure

characters

- prolog is based on the ASCII character set
- characters are treated as small integers 0 .. 127
- characters may be
  - printed
  - read from a file or keyboard
  - compared
  - take part in arithmetic operations
- characters are distinguished as
  - printing – visible on the paper
  - nonprinting – look like whitespace

operators

- all predicates in Prolog are functors, even ,, ; and, :-
- a rule such as
  - dwarf (Person) :- brother (Person , Other) ,
    dwarf (Other) .

  is a shorthand for
  - :- (dwarf (Person)
    , , (brother (Person , Other)
    , dwarf (Other)
)
).

operators (cont'd)

- arithmetic and relational operators are also functors, thus
  - a + b * c internally is + (a , * (b , c))
- this is inconvenient so Prolog permits operators to be written in standard infix notation
  - you will learn later how you can define your own infix operators

arithmetic

- the arithmetic operators do not do arithmetic; no assignments are made
  - it is simply pattern matching – infix operators are simply a convenience for expressing a structure
- e.g.:
  - 5 = 4 + 1. → no
  - 4 + 1 = 4 + 1. → yes
  - 1 + 4 = 4 + 1. → no
- use the operator is to do arithmetic
  - 5 is 4 + 1. → yes 1 + 4 is 4 + 1. → no
- arithmetic is only done on the right!
- right hand side is evaluated using arithmetic, then a pattern match is made with the left hand side.

arithmetic (cont'd)

- can use variables in arithmetic expressions for pattern matching
  - A = 4 + 1. → A has the pattern "4+1"
    - spaces removed
  - A is 4 + 1. → A has as value the pattern 5
- more complex example
  - B is 3 + 2 , C is B * 5 , A is C + B.
    → B = 5, C = 25, A =30

lists

- as in Lisp, lists are a ubiquitous structure in Prolog; the syntax changes
  - ( ) are used to delimit structure components and to provide precedence for operators, so using them for lists as well would be confusing
- the structure is
  - [item-1 , item-2 , ... , item-n]
- e.g.
  - [a , b , c]
  - [a , [b , c] , [[[d]]] , e , [ ]]
- The empty list is [ ]

lists (cont'd)

- the square bracket notation is a shorthand in place of using the functor **.** or dot

  [a , b , c] is really .(a , .(b , .(c , [])))
- as in Lisp, lists have a head (car / first) and a tail (cdr / rest), thus

  [Head | Tail]
- but you do not have operators to extract the head and tail, all you have is pattern matching
  - we will look at example Prolog utilities on lists to demonstrate
- Empty list has no head or tail

  [ ] ≠ [_ | _]

Utility programs

In utilities.pro discussed at various times throughout rest of the course

Shakil M. Khan

adapted from Gunnar Gotshalks

member (I , L)

- item I is a member of the list L
  - reduce the list (second rule)
    until first in list (first rule) or
    empty (no rule, so fail)
  - member (X , [X | _]).
    member (X , [_ | Z]) :- member (X , Z).
- note the use of the anonymous variable _
  - we do not care about the value of the rest in the first rule, nor the value of first in the second rule
  - typically use it when it is the only instance of that variable in the rule

append (L1, L2 , R)

- R is the result of appending list L2 to the end of list L1
  - cannot redefine in Qunitus Prolog.
  - append ([] , L , L).
    appending to nil yields the original list
  - append ([X | L1] , L2 , [X | L3])
    :- append (L1 , L2 , L3).
    simultaneous recursive descent on L1 & L3 first of the left list is the first of the result
- pattern
  - L1 = a b c L2 = 2 3 4 5 R = a b c 2 3 4 5

append (L1 , L2 , R) – cont'd

- queries – ask for results in all combinations; not like Java or C where functions are programmed for only one query
  - append ([1 , 2 , 3] , [a , b , c] , R).
    what is the result of appending L1 and L2?
  - append (L1 , [a , b , c] , [1 , 2 , 3 , a , b , c]).
    what L1 gives [1 , 2 , 3 , a , b , c] when appended with [a , b , c] ?
  - append ([1 , 2 , 3] , L2 , [1 , 2 , 3 , a , b , c]).
    what L2 gives [1 , 2 , 3 , a , b , c] when appended to [1 , 2 , 3] ?

append (L1 , L2 , R) – cont'd

  - append (L1 , L2 , [1 , 2 , 3 , a , b , c]).
    what L1 and L2 gives [1 , 2 , 3 , a , b , c] when L2 is appended to L1?
  - append (L1 , L2 , R).
    what L1 and L2 give R? Infinite number of answers
  - append (Before , [Middle | After] , List).
    if middle is defined we can get the before and after
    append (Before, [4 | After], [1,2,3,4,5,6,7]).

trace –
append (P, [a] , [1 , 2 , 3 , a])

- variables are renamed every time a rule is used for matching
  - append ([] , L , L).
    append ([X | L1] , L2 , [X | L3])
    :- append (L1 , L2 , L3).
- try to match rule 1
  - P = [], [a] = L_1, [1,2,3,a] = L_1
- 1 – fail, try to match rule 2
  - P = [X_2 | L1_2], [a] = L2_2, [1,2,3,a] = [X_2 | L3_2]
  - succeed with X_2 = 1, L2_2 = [a], L3_2 = [2,3,a]

trace – (cont'd)
append (P, [a] , [1 , 2 , 3 , a])

  - append ([] , L , L).
    append ([X | L1] , L2 , [X | L3])
    :- append (L1 , L2 , L3).
- try to match rule 1 append(L1_2, [a], [2,3,a])
  - L1_2 = [], [a] = L_3, [2,3,a] = L_3
- 2 – fail, try to match rule 2
  - L1_2 = [X_4 | L1_4], L2_4 = [a], [2,3,a] = [X_4 | L3_4]
  - succeed with X_4 = 2, L2_4 = [a], L3_4 = [3,a]
- try to match rule 1 append(L1_4, [a], [3,a])
  - L1_4 = [], [a] = L_5, [3,a] = L_5

trace – (cont'd)
append (P, [a] , [1 , 2 , 3 , a])

- append ([] , L , L).
  append ([X | L1] , L2 , [X | L3])
  :- append (L1 , L2 , L3).
- 3 – fail, try to match rule 2
  - L1_4 = [X_6 | L1_6], [a] = L2_6, [3,a] = [X_6 | L3_6]
  - succeed with X_6 = 3, L2_6 = [a], L3_6 = [a]
- try to match rule 1 append(L1_6, [a], [a])
  - L1_6 = [], [a] = L_7, [a] = L_7
- succeed, recursion stops, backtrack and substitute values

trace – (cont'd)
append (P, [a] , [1 , 2 , 3 , a])

- in step 3
  L1 _4 = [3 | []] = [3]
- in step 2 we had
  L1_2 = [X_4 | L1_4], L2_4 = [a],
  [2,3,a] = [X_4 | L3_4]
  - succeed with X_4 = 2, L2_4 = [a], L3_4 = [3,a]
  - and from step 3, L1_4 = [3]
  - thus L1_2 = [2, 3]
- in step 1 we had
  P = [X_2 | L1_2], [a] = L2_2, [a,1,2,3] = [X_2 | L3_2]
  - succeed with X_2 = 1, L2_2 = [a], L3_2 = [2,3,a]
  - and from step 2, L1_2 = [2, 3]
  - thus P = [1, 2, 3]

delete (I , L , R)

- R is the result of deleting item I from the list L.
  - delete (X , [X | Y] , Y).
    like saying L = (cons (car L) (cdr L)) in Lisp
  - delete (X , [Y | W] , [Y | Z]) :- delete (X , W , Z).
    check the rest of the list if not the first item;
    analogous to (cons (car L) (delete (cdr L)) in Lisp

prefix (P , L)

- P is the prefix of the list L; it can be defined using append as follows:
  - prefix (P , L) :- append (P , _ , L).
    P is a prefix of L if something, including nil, can be suffixed to P to form L

prefix (P , L) – cont'd

- we can define prefix in terms of itself as follows
  - List YYYYYYXXXXX → XXXXX
    Prefix PPPPPP → empty
    ^^^^^ check equality until prefix is exhausted
- the base case is having the empty list as the prefix

  prefix ([] , _).
- the recursive case is having the first items on the prefix and the list being the same, and the reduced prefix and list satisfy the prefix property

  prefix ([A | B] , [A | C]) :- prefix (B , C).

suffix (S , L)

- S is the suffix of the list L; it can be defined using append as follows
  - suffix (S , L) :- append (_ , S , L).

    S is a suffix of L if something, including nil, can be prefixed to S to form L

suffix (S , L) – cont'd

- we can define suffix in terms of itself as follows
  - List PPPPPPXXXXX
    Suffix YYYYY
    ^^^^^ reduce to the prefix part of the List
- in the base case the suffix is the list
  suffix (L , L).
- the recursive case is to reduce the size of the prefix of the list
  suffix (S , [_ | L]) :- suffix (S , L).

sublist (S , L)

- S is a sublist of L can be defined using append as follows
  - sublist (S, L) :- append (_ , S, Lt), append (Lt, _ , L).
    - S is a sublist of L if something, including nil, can be prefixed to S to form the list Lt
    - and something, including nil, can be suffixed to Lt to form L
- in other words, S is a sublist of L if there exists a prefix P to S and a suffix T to S such that
  L = P |+| S |+| T
  - where |+| means concatenation

sublist (S , L) – cont'd

- we can define sublist in terms of itself and prefix as follows
  - List PPPPSSSSSXXXXXX → SSSSSXXXXXX
    Sublist YYYYY → YYYYY
    ^^^^ reduce the prefix part of the List
- in the base case the sublist is the prefix of the list

  sublist (S , L) :- prefix (S , L).
- the recursive case is to reduce the size of the prefix of the list

  sublist (S , [_ | L]) :- sublist (S , L).

Example programs

Showing things to look for

Shakil M. Khan
adapted from Gunnar Gotshalks

infinite loops

- avoid circular definitions
  - parent (A, B) :- child (B, A).
    child (C, D) :- parent (D, C).
- easy to see here but as database grows you can forget what is in it and circularity can creep in

infinite loops – left recursion

- left recursion can cause problems
  - person (X) :- person (Y) , mother (Y, X).
    person (eve).
  - the query person (P) loops indefinitely as the first rule is found first on every recursive call
  - second rule is only tried if first rule fails
- reordering the rules will correct the problem if only the first answer is wanted
- heuristic: put facts before rules

infinite loops – left recursion (cont'd)

- left recursion can cause problems – continued
  - person (eve).
    person (X) :- person (Y) , mother (Y, X).
  - assuming mother fails, the query person (P) loops indefinitely after P = eve
- left recursion is the problem
- do not assume Prolog will find the facts and rules – need to know how searching works

multiple answers – isList, weakList

- the textbook gives the following predicate, but it loops on the query isList (X).
  - isList ([A | B]) :- isList (B).
    isList ([]).
- it can be defined just as well by putting the fact first
  - isList ([]).
    isList ([A | B]) :- isList (B).
- but gives more than one answer for the query isList (X), but does not loop forever
- for the latter query, to have only one answer, can assert the following
  - weak_isList ([]).
    weak_isList ([_ | _]).

why is weak_isList weak?

- the strong definition says a list must have the correct structure and must end in nil
- the weak definition simply says the list must have the correct structure for one level and says nothing about nil except for the empty list
- e.g. recall [...] is shorthand for the structure .(...)
  - isList (.(a , [])). → yes
  - isList (.(a , .(b , []))). → yes
  - isList (.(a , .(b, .(c , [])))). → yes
  - isList (.(a , b)). → no
  - isList (.(a , .(b , c , []))). → no
- but all responses are yes for weak_isList

mapping

- consider the problem of translating a sentence from one form to another
- e.g. as in the following "dialogue", the second sentence is a translation of the preceding sentence
  - you are a computer
    I am not a computer
  - do you speak french
    no I speak german
  - assume the following simplistic translations
    - you → I
    - are → am not
    - do → no
    - french → german

mapping (cont'd)

- let us represent sentences as a list of words
  - you are a computer → [you , are , a , computer]
- we represent the list of words to change as a set of change rules
  - change (you , i).
    change (are , [am , not]).
    change (french , german).
    change (do , no).
    change (X , X). /* catch all to make no changes */

mapping (cont'd)

- then the translation rules can be the following
  - alter([] , []).
    alter ([H | T] , [X | Y]) :- change (H, X), alter (T, Y).
- then we can translate our example sentences
  - alter ([you, are, a, computer] , Trans).
    trans = [i , am , not , a , computer]
  - try using ;<return> on the above; explain why there are multiple answers; try a trace to see what is happening
  - we need a method to prevent multiple answers

mapping (cont'd)

- try the inverse – with ; <return>
  - alter (Org , [i , am , not , a , computer]).
- try a variable – with ; <return>
  - alter ([you , are , a , X] , Trans)

warning – caution – danger

- logic and a finite database can lead to strange and unexpected results
- use with extreme caution

info

- drop date: July 2
- I'll take the highest of the 2 class tests and make it worth 40%!
- assignment/report 2
- test 2:
  - July 9th (2 weeks from today)