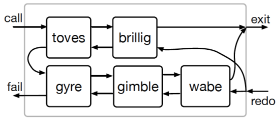
# 1 Prolog

## Sample Box Model

```
slithy :- toves, brillig.
slithy :- gyre, gimble, wabe.
toves :- outgrabe, vorpal.
toves :- manxome.
brillig :- jubjub.
outgrabe.
vorpal :- manxome.
gyre :- manxome.
gyre :- outgrabe.
manxome.
gimble :- outgrabe.
gimble:- brillig.
gimble :- vorpal.
wabe.
```




## Sample Proof Procedure

```
wff(A,B) :- terminal(A,Z), binop(Z,Y), wff(Y,B).
wff(A,B) :- unop(A,Z), wff(Z,B).
wff(A,B) :- terminal(A,B).

binop([&|T],T).
binop([or|T],T).
unop([not|T],T).

terminal([Term|T],T) :- term(Term).

term(p).
term(q).
term(r).
```

```
wff([p,&,not,q,or,r],V).

yes(V) :- wff([p,&,not,q,or,r],V).
yes(V) :- terminal([p,&,not,q,or,r],Z1), binop(Z1,Y1), wff(Y1,V).
yes(V) :- term(p), binop([&,not,q,or,r],Y1), wff(Y1,V).
yes(V) :- binop([&,not,q,or,r],Y1), wff(Y1,V).
yes(V) :- wff([not,q,or,r],V).
yes(V) :- unop([not,q,or,r],Z2),wff(Z2,V).
yes(V) :- wff([q,or,r],V).
yes(V) :- terminal([q,or,r],Z3), binop(Z3,Y3), wff(Y3,V).
yes(V) :- term(q), binop([or,r],Y3), wff(Y3,V).
yes(V) :- binop([or,r],Y3), wff(Y3,V).
yes(V) :- wff([r],V).
yes(V) :- terminal([r],V).
yes([]) :- term(r).
yes([]).
```

## Sample Most General Unifier

```
(a) p(W; b; f (W)) and p(h(X; Z); X; Y).
(b) ok([fun; course; dream]; A; Y) and ok([A j R]; R; A).
(c) bar(A; [o; l j B]; B) and bar([p; r j C];C; [o; g])

(a) { W / h(b, Z), X / b, Y / f(h(b, Z))}
(b) does not unify because A would have to be 'fun' and [course, dream]
(c) {A / [p, r, o, l, o, g], B / [o, g], C / [o, l, o, g]}.
```

## Delete Exactly 1 Elements

```
% del1(E, L, R) is true when R is a list with the same elements as
% list L (in the same order) but with one instance of E removed
del1(E,[E|T],T).
del1(E,[H|T],[H|R]) :-
    del1(E,T,R).
```

## Delete $n$ Elements

```
%  deln(N,E,L,R)  is true when R is the result of removing N instances of E from L.
deln(0,_,[],[]).
deln(N,E,[E|L],R) :-
    N>0,
    N1 is N-1,
    deln(N1,E,L,R).
deln(N,E,[H|L],[H|R]) :-
    deln(N,E,L,R).
% ?- deln(2,a,[a,v,a,t,a,r],R).

% Here is another solution
%  deln_alt(N,E,L,R)  is true when R is the result of removing N instances of E from L.
deln_alt(0,_,R,R).
deln_alt(N,E,[E|L],R) :-
    N>0,
    N1 is N-1,
    deln_alt(N1,E,L,R).
deln_alt(N,E,[H|L],[H|R]) :-
    N>0,
    deln_alt(N,E,L,R).
```

## Remove Duplicates (Keep 1st Occurrence)

```
myremoveduplicates_helper([], _, []).
myremoveduplicates_helper([H|T], Duplicates, RecursiveResult) :-
    member(H, Duplicates),
    myremoveduplicates_helper(T, Duplicates, RecursiveResult).
myremoveduplicates_helper([H|T], Duplicates, [H|RecursiveResult]) :-
    \+ member(H, Duplicates),
    myremoveduplicates_helper(T, [H|Duplicates], RecursiveResult).

myremoveduplicates(Lst, Result) :-
    myremoveduplicates_helper(Lst, [], Result).
```

## Replace

```
replace(_, _, [], []).
replace(Old, New, [Old | L], [New | R]) :-
    replace(Old, New, L, R).
replace(Old, New, [Head | L], [Head | R]) :-
    dif(Old,Head),
    replace(Old, New, L, R).
```

## Apply

```
% apply(L,S,R) is true if list L becomes R according to substitution S
% where a substitution is a list of sub(from,to) terms
apply([],_,[]).
apply([H|L],S,[HR|R]):-
    rep(H,S,HR),
    apply(L,S,R).

% rep(E,S,R) is true if E gets replaced by R according to substitution S
rep(H,[],H).
rep(H,[sub(H,R)|_],R).
rep(H,[sub(H1,_)|S],R) :-
    dif(H,H1),
    rep(H,S,R).
```

```
% appla(A,S,R) is true if arithmetic expression A becomes R according to substitution S
appla(A,S,R) :-
    atomic(A),
    rep(A,S,R).
appla((A+B),S,(AR+BR)) :-
    appla(A,S,AR),
    appla(B,S,BR).
appla((A*B),S,(AR*BR)) :-
    appla(A,S,AR),
    appla(B,S,BR).
```

## Reverse Flatten

```
flatr([[A|B]|T],R0,R2) :-
    flatr([A|B],R1,R2),
    flatr(T,R0,R1).
```

## Shuffle

```
% shuffle(L1,L2,L3) is true if L3 is an interleaving of the elements of L1 and L2
shuffle(L,[],L).
shuffle([],L,L) :- dif(L,[]).
shuffle([H|T],L,[H|R]) :- shuffle(T,L,R), dif(L,[]).
shuffle(T,[H|L],[H|R]) :- shuffle(T,L,R), dif(T,[]).
```

## Merge Sort

```
split([],[],[]).
split([X],[X],[]).
split([X,Y|R],[X|R1],[Y|R2]) :- split(R,R1,R2).

merge([],L,L).
merge(L,[],L) :- dif(L,[]).
merge([A|L1],[B|L2],[A|R]) :-
    A =< B,
    merge(L1,[B|L2],R).
merge([A|L1],[B|L2],[B|R]) :-
    A > B,
    merge([A|L1],L2,R).

msort([],[]).
msort([A],[A]).
msort([A,B|L],S) :-
    split([A,B|L],L1,L2),
    msort(L1,S1),
    msort(L2,S2),
    merge(S1,S2,S).
```

## BTree (With Append)

```
% Tree to list (with append)
tolist(empty,[]).
tolist(node(LT,Val,RT),L):-
    tolist(LT,L0),
    tolist(RT,L1),
    append(L0,[Val|L1],L).
```

## BTree (Without Append)

```
% Tree to list (without append)
tolist_dl(T,L) :-
    tolist3(T,L,[]).

% tolist3(T, L1, L2) is true if L1-L2 is preorder list of values in tree T
tolist3(empty, L, L).
tolist3(node(LT,Val,RT),L0,L2):-
    tolist3(LT,L0,[Val|L1]),
    tolist3(RT,L1,L2).
```

## Derivative

```
% deriv(E,X,DE) is true if DE is the derivatibe of E with respect to X
deriv(X,X,1).
deriv(C,X,0) :- atomic(C), dif(C,X).
deriv(A+B,X,DA+DB) :-
    deriv(A,X,DA),
    deriv(B,X,DB).
deriv(A*B,X,A*DB+B*DA) :-
    deriv(A,X,DA),
    deriv(B,X,DB).

% Some optional extras
deriv(sin(E),X,cos(E)*DE) :-
    deriv(E,X,DE).
deriv(cos(E),X,-sin(E)*DE) :-
    deriv(E,X,DE).
```

## Simplify

```
%simplify(Exp, Exp2) is true if expression Exp simplifies to Exp2
simplify(X,X) :-
    atomic(X).
simplify((A+B),V) :-
    simplify(A, VA),
    simplify(B, VB),
    simp_vals(VA+VB, V).
simplify((A*B),V) :-
    simplify(A, VA),
    simplify(B, VB),
    simp_vals(VA*VB, V).

%simplify_vals(Exp, Exp2) is true if expression Exp simplifies to Exp2,
% where the arguments to Exp have already been simplified
simp_vals(0+V,V).
simp_vals(V+0,V).
simp_vals(A+B,AB) :-
    number(A),number(B),
    AB is A+B.
simp_vals(A+B,A+B).
simp_vals(0*_,0).
simp_vals(_*0,0).
simp_vals(V*1,V).
simp_vals(1*V,V).
simp_vals(A*B,AB) :-
    number(A),number(B),
    AB is A*B.
simp_vals(A*B,A*B).
```

## Arithmetic Operations

```
% def(F,A,FA) function F on argument A returns FA
def(sq,X,X2) :- X2 is X*X.
def(plus,X,plus(X)).
def(plus(X),Y,Z) :- Z is X+Y.
def(gt,X,gt(X)).      % gt(X) is \Y -> X>Y
def(gt(X),Y,true) :- X>Y.
def(gt(X),Y,false) :- X =< Y.

% eval(E,V) is true if expression E evaluates to V
% this uses square brackets as parentheses, values separated by commas
eval(N,N) :- number(N).
eval([V],V).
eval([P,A|R],V) :-
    eval(A,AV),
    def(P,AV,R1),
    eval([R1|R],V).
```

## Fibonacci

```
fib(1,0).
fib(N,F) :- N>1, N1 is N-1, fib2(N1,1,1,F).
fib2(0,_,F,F).
fib2(N,F0,F1,F) :- N>0, FS is F0+F1, N1 is N-1, fib2(N1,F1,FS,F).
```

## Powerset

```
% Powerset
% Straight Recursion
powerset([],[[]]).
powerset([H|T], R) :-
    powerset(T,PT),
    applytoeach(H,PT,R).
applytoeach(_,[],[]).
applytoeach(H,[E|R],[[H|E],E|PR]) :-
    applytoeach(H,R,PR).

% With an accumulator
powset([],[[]]).
powset([H|T],R) :-
    powset(T,TP),
    add_to_each(H,TP,TP,R).
add_to_each(_,[],TP,TP).
add_to_each(H,[L|R],TP,[[H|L]|A]) :-
    add_to_each(H,R,TP,A).

% With an accumulator, alternative answer
powset2([],[[]]).
powset2([H|T],R) :-
    powset2(T,TP),
    add_to_each2(H,TP,TP,R).
add_to_each2(_,[],TP,TP).
add_to_each2(H,[L|R],TP,A) :-
    add_to_each2(H,R,[[H|L]|TP],A).
```

## Dutch Flag

```
% for testing, here is a definition of the colour of numbers
red(E) :- E < 10.
white(10).
blue(E) :- E>10.

% dutch_flag(L,D) is true if D conatins the elements of L where,
% the red elements are before the white elements, which are before the blue elements.

% dutch flag with difference lists
dutch_flag_dl(L,R):-
    partn_dl(L,R,W,W,B,B,[]).

%partn_dl(L,R1,R2,W1,W2,B1,B2) is true when
%  R1-R2 contains elements of L that are red
%  W1-W2 contains elements of L that are white
%  B1-B2 contains elements of L that are blue
partn_dl([],R,R,W,W,B,B).
partn_dl([H|T],[H|R1],R2,W1,W2,B1,B2) :-
    red(H),
    partn_dl(T,R1,R2,W1,W2,B1,B2).
partn_dl([H|T],R1,R2,[H|W1],W2,B1,B2) :-
    white(H),
    partn_dl(T,R1,R2,W1,W2,B1,B2).
partn_dl([H|T],R1,R2,W1,W2,[H|B1],B2) :-
    blue(H),
    partn_dl(T,R1,R2,W1,W2,B1,B2).
```

## Reverse Dutch Flag

```
% reversing dutch flag with difference lists

% rev_dutch_flag(L,D) is true if D conatins the elements of L where,
% the red elements are before the white elements, which are before the blue elements.
% the elements in each colour group are in reverse order that they are in L
rev_dutch_flag_dl(L,R):-
    rev_partn_dl(L,R,W,W,B,B,[]).

%rev_partn_dl(L,R1,R2,W1,W2,B1,B2) is true when
%  R1-R2 contains elements of L that are red in reverse order
%  W1-W2 contains elements of L that are white in reverse order
%  B1-B2 contains elements of L that are blue in reverse order
rev_partn_dl([],R,R,W,W,B,B).
rev_partn_dl([H|T],R1,R2,W1,W2,B1,B2) :-
    red(H),
    rev_partn_dl(T,R1,[H|R2],W1,W2,B1,B2).
rev_partn_dl([H|T],R1,R2,W1,W2,B1,B2) :-
    white(H),
    rev_partn_dl(T,R1,R2,W1,[H|W2],B1,B2).
rev_partn_dl([H|T],R1,R2,W1,W2,B1,B2) :-
    blue(H),
    rev_partn_dl(T,R1,R2,W1,W2,B1,[H|B2]).
```

# 2 Haskell

## Folding

$foldr \oplus v [a1, a2, ..., an] = a1 \oplus (a2 \oplus (... \oplus (an \oplus v)))$
$foldl \oplus v [a1, a2, ..., an] = (((v \oplus a1) \oplus a2) \oplus ...) \oplus an$

## Evaluation

Different Types of Evaluation

- call-by-value: evaluate argument before applying
- call-by-name: reduction of function first
- lazy evaluation (call-by-need): evaluate argument only once, only if needed
  1. evaluation from outside in
  2. otherwise (if it knows both arguments need to be evaluated) from left to right

Lazy Evaluation vs. Call-By-name

Lazy evaluation evaluates its arguments at most once, whereas call-by-name could evaluate an argument multiple times.

## Polymorphic Typing

Definition

Polymorphic typing is when a function can take many types.

## IO

```
do  v1 <- a1
    v2 <- a2
    ...
    vn <- an
    return (f v1 v2 ... vn)
```

## Tail Recursion

```
-- harmonic_tr n evaluates to the nth harmonic number
harmonic_tr n = harmonic_tr_help 1 1 n
    where
        -- harmonic nc h n evaluates to the nth harmonic number given that the nc-th harmonic number is h
        harmonic_tr_help nc v n
            | nc == n = v
            | otherwise = harmonic_tr_help (nc+1) (v+1/(nc+1)) n
```

## Delete Exactly 1 Element (First Occurrence)

```
-- del1 e lst     -- returns a list with one instance of e removed from list lst
-- need to make a choice of what happens if e is not in lst
-- it could
---- return lst
---- give a runtime error
---- return Nothing
del1 e (h:t)
    | e==h      = t
    | otherwise = e:del1 e t
```

## Delete Exactly 1 Element (All Possibilities)

```
delall :: Eq t => t -> [t] -> [[t]]
delall _ [] = []
delall e (h:t)
    | e==h  = t:[h:l | l <- delall e t]
    | otherwise = [h:l | l <- delall e t]
```

### Delete Exactly $n$ Elements

```haskell
-- delna n e lst returns a list of all of the lists that result from deleting exactly n occurrences of e from lst
delna :: Eq t => Int -> t -> [t] -> [[t]]
delna 0 _ lst = [lst]
delna n e [] = []
delna  n e (h:t)
    | e==h = (delna (n-1) e t) ++ cons_to_each h (delna n e t)
    | otherwise = cons_to_each h (delna n e t)
        where
            -- cons_to_each e lst  -- conses e to every element of lst
            cons_to_each _ [] = []
            cons_to_each e (h:t) = (e:h):cons_to_each e t
```

### Delete Exactly $n$ Elements (List Comprehensions)

```haskell
-- Note that this would be easier with list comprehensions:
delna2 :: Eq t => Int -> t -> [t] -> [[t]]
delna2 0 _ lst = [lst]
delna2 n e [] = []
delna2 n e (h:t)
    | e==h = (delna2 (n-1) e t) ++ [h:r | r <- delna2 n e t]
    | otherwise = [h:r | r <- delna2 n e t]
```

### Remove duplicates (Keep 1st Occurrence)

```haskell
myremoveduplicatesfirst :: Eq t => [t] -> [t]
myremoveduplicatesfirst lst = remdupfirst lst []
    where
        -- remdupfirst lst1 lst2   returns the elements in lst1 without duplicates that are not in lst2
        remdupfirst [] _ = []
        remdupfirst (h:t) lst2
            | h `elem` lst2 = remdupfirst t lst2
            | otherwise     = h : remdupfirst t (h:lst2)
```

### Remove Duplicates (Higher-order Functions)

```haskell
myremoveduplicates  lst = [ head l | l <- tails lst, l /= [], not (head l `elem` tail l)]
myremoveduplicates2  lst = [ h | (h:t) <- tails lst, not (h `elem` t)]    --using pattern matching
```

### Apply

```haskell
-- myapply lst sub where sub is a list of (x,y) pairs, replaces each occurrence of x by y in lst.
myapply :: Eq t => [t] -> [(t, t)] -> [t]
myapply []  _ = []
myapply (h:t) sub = app h sub :myapply t sub
    where
        -- app e sub gives the value e is replaced by according to sub
        app e [] = e
        app e ((x,y):r)
            | e==x = y
            | otherwise = app e r
```

### Apply (Higher-Order Functions)

```haskell
myapply lst sub = [(\res -> if res /= [] then head res else e) [b | (a,b) <- sub, a==e]  | e <- lst]
-- myapply "abcdec" [('a','f'), ('c','3'), ('g','7')]
-- myapply "baab" [('a','b'), ('b','a')]
```

```haskell
-- or even clearer....
-- head_with_default lst def   = head of list lst, otherwise (if there is no head) evaluates to def
head_with_default [] def = def
head_with_default (h:t) _     = h
myapply1 lst sub = [head_with_default [b | (a,b) <- sub, a==e]  e | e <- lst]

-- or even simpler
myapply2 lst sub = [head [b | (a,b) <- sub++[(e,e)], a==e] | e <- lst]
```

### Append (All Possibilities)

```haskell
--- iappend l = [(l1,l2)] where l1 appended to l2 gives l
--- returns list of all answers
iappend :: [t] -> [([t],[t])]
iappend [] = [([],[])]
iappend (h:t) = ([],(h:t)) : [(h:l1,l2) | (l1,l2) <- iappend t]
```

### Reverse (Pattern Matching)

```haskell
-- rev2 l1 l2 = reverse of list l2 followed by l1
rev2 l1 []      = l1
rev2 l1 (x:xs) = rev2 (x:l1) xs
-- rev2 [1,2,4,6] [11,23,45,56]

rev = rev2 []
```

### Reverse (Lambda Calculus)

```haskell
re2 = myfoldl (myflip (:)) []
```

### Tails

```haskell
tails1 :: [t] -> [[t]]
tails1 = foldr (\x (h:r) -> (x:h):(h:r) ) [[]]
```

### Split

```haskell
-- split (`elem` " ,.?!") "What? is this thing? called Love."  =>  ["What","is","this","thing","called","Love"]
split sep [] = []
split sep (h1:h2:t)
    | sep h1 = split sep (h2:t)
    | sep h2 = [h1]:split sep (h2:t)
    | otherwise = ((h1:w1):rest)
                    where w1:rest = split sep (h2:t)
split sep [h]    -- if previous patterns do not match the argument must be a single element list
    | sep h = []
    | otherwise = [[h]]
```

### Subsequence

```haskell
subsequence l1 l2 = starts l1 l2 || l2 /= [] && subsequence l1 (tail l2)
   where
    starts [] _ = True
    starts _ [] = False
    starts (h1:t1) (h2:t2) = h1==h2 && starts t1 t2
```

### Shuffle

```haskell
shuffle :: [t] -> [t] -> [[t]]  -- list of all shuffles
shuffle l1 [] = [l1]
shuffle [] l2 = [l2]
shuffle (h1:t1) (h2:t2) =
    [h1:e | e <- shuffle t1 (h2:t2)] ++ [h2:e | e <- shuffle (h1:t1) t2]
```

### Merge Sort

```haskell
split [] = ([],[])
split [x] = ([x],[])
split (x:y:r) =
    let (r1,r2) = split r
    in (x:r1, y:r2)

merge [] l = l
merge l [] = l
merge (a:l1) (b:l2)
    | a <= b = a: merge l1 (b:l2)
    | otherwise = b: merge (a:l1) l2

msort [] = []
msort [a] = [a]
msort l =
    let (l1,l2) = split l
    in merge (msort l1) (msort l2)
```

### BTree

```haskell
-- a binary tree where the nodes are labelled with integers
data BTree = Empty
            | Node BTree Int BTree
```

### BTreee tolist (With Append)

```haskell
-- tolist lst  retuens the list giving the inorder traversal of lst
tolist :: BTree -> [Int]
tolist Empty = []
tolist (Node lt val rt) =
    tolist lt ++ (val : tolist rt)
```

### BTreee tolist (Without Append)

```haskell
-- tolist without append (++), using accumulators
tolista :: BTree -> [Int]
tolista lst =
    tolist2 lst []

-- tolist2 tree lst   returns the the list of elements of tree followed by the elements of lst
tolist2 :: BTree -> [Int] -> [Int]
tolist2 Empty acc = acc
tolist2 (Node lt val rt) acc =
     tolist2 lt  (val : tolist2 rt acc)
```

### BSTree

```haskell
-- a binary search tree
data BSTree k v = Empty
                | Node k v (BSTree k v) (BSTree k v)
         deriving (Eq, Show, Read)
```

### BSTreee tolist (With Append)

```haskell
-- tolist lst  returns the list giving the inorder traversal of lst
tolist :: BSTree k v -> [(k,v)]
tolist Empty = []
tolist (Node key val lt rt) =
     tolist lt ++ ((key,val) : tolist rt)
```

### BSTree tolist (Without Append)

```haskell
tolista :: BSTree k v -> [(k,v)]
tolista lst =
    tolist2 lst []
     where
       -- tolist2 tree lst   returns the the list of elements of tree followed by the elements of lst
       -- tolist2 :: BSTree k v -> [(k,v)] -> [(k,v)]
       tolist2 Empty acc = acc
       tolist2 (Node key val lt rt) acc =
            tolist2 lt  ((key,val) : tolist2 rt acc)
```

### BSTree insert

```haskell
-- insert key val tree   returns the tree that results from inserting key with value into tree
insert :: Ord k => k -> v -> BSTree k v -> BSTree k v
insert key val Empty = Node key val Empty Empty
insert key val (Node k1 v1 lt rt)
    | key == k1 = Node key val lt rt
    | key < k1 = Node k1 v1 (insert key val lt) rt
    | key > k1 = Node k1 v1 lt (insert key val rt)
```

### BSTree insertv

```haskell
-- insertv key val tree   returns the previous value and the resulting tree
insertv :: Ord k => k -> v -> BSTree k v -> (Maybe v, BSTree k v)
insertv key val Empty = (Nothing, Node key val Empty Empty)
insertv key val (Node k1 v1 lt rt)
    | key == k1 = (Just v1, Node key val lt rt)
    | key < k1 = let (res,nt) = insertv key val lt
                 in (res, Node k1 v1 nt rt)
    | key > k1 = let (res,nt) = insertv key val rt
                 in (res, Node k1 v1 lt nt)
```

### BSTree btlookup

```haskell
-- lookup key tree   returns the value of key in tree
btlookup :: Ord k => k -> BSTree k v -> Maybe v
btlookup key (Node k1 v1 lt rt)
    | key == k1 = Just v1
    | key < k1  = btlookup key lt
    | otherwise = btlookup key rt
btlookup key Empty = Nothing
```

### BSTree value_in_tree

```haskell
value_in_tree BSEmpty _ = False
value_in_tree (BSNode key val lt rt) v =
    val==v || value_in_tree lt v || value_in_tree rt v
```

### BSTree lefttree

```haskell
lefttree :: BSTree k v -> Maybe (BSTree k v)
lefttree Empty = Nothing
lefttree (Node _ _ lt _) = Just lt
```

### Average (One Pass)

```haskell
-- sumn lst = (sum lst, lenght lst)
sumn :: [Int] -> (Int,Int)  -- (sum,lenght)
sumn [e] = (e,1)
sumn (h:t) =
    let (s,n) = sumn t
    in (s+h, n+1)
ave lst =
    let (s,n) = sumn lst
    in s `div` n
```

### Powerset

```haskell
powerset [] = [[]]
powerset (h:t) =
    let pset = powerset t
    in  [h:s | s <- pset] ++ pset
```

### Covert to Upper Case

```haskell
toUpper :: Char -> Char
toUpper x = toEnum( fromEnum x - fromEnum 'a' + fromEnum 'A')
```

## 3 Misc

Purpose of Uniform Resource Identifier (URI)

A URI is a constant that has a well-defined and standard meaning. It is useful because everyone who uses the constant means the same thing. (It does not imply that there is a unique URI for each thing (the unique names assumption); there can be multiple URIs for the same thing.)

Prolog vs. Haskell

Prolog can do something like del1(val(x,V),[val(y,3),val(x,7),val(x,2)],R). to return the value of x as well as the rest of the list. Haskel can work with higher-order function, e.g., generalizing this delete1 to act on other functions of the elements (not just equality).

Prolog

- constraint-based programming
- You can easily specify a set of constraints that may have more than one solution and Prolog will find all of them
- In Haskell one would have to manually implement a constraint solver and associated data structures to represent constraints, then write the constraints and pass them to the solver. Multiple execution paths are not directly supported.

Haskell

- static typechecking
- You can automatically catch many programming errors before even running the program - Haskell detects a lot of inconsistencies for you
- In Prolog test the program and reason about its behaviour. Sometimes you find bugs that would have been type errors when running the program and have to identify and fix them.

Main Advantage of Strong Typing

Strong typing help programmers eliminate many bugs before the program is put into production.

Advantage of Interactive Development

Programmers can experiment and test their code interactively, and to see results immediately.

Haskell Function Arguments

Q: In Haskell every function takes a single argument. What does this mean for functions that take 2 arguments, such as (+)?

A: (+) applied to a single argument returns a function which takes one argument and returns a value.

Difference List (Prolog vs. Haskel)

Prolog

```prolog
wff(A,B) :- terminal(A,Z), binop(Z,Y), wff(Y,B).
wff(A,B) :- unop(A,Z), wff(Z,B).
wff(A,B) :- terminal(A,B).

binop([&|T],T).
binop([or|T],T).
unop([not|T],T).

terminal([Term|T],T) :- term(Term).

term(p).
term(q).
term(r).
```

Haskell

```haskell
-- this uses the same naming conventions for variables as in the prolog (except, lower case, of course)
wff :: [[Char]] -> [[[Char]]]
wff a =
   [ b | z <- terminal a, y <- binop z, b <- wff y]
   ++ [b | z <- unop a, b <- wff z]
   ++ terminal a

binop ("&":t) = [t]
binop ("or":t) = [t]
binop (_:_) = []
binop [] = []

terminal (term:t)
    | term `elem` ["p","q","r"] = [t]
    | otherwise = []
terminal [] = []

unop ("not":t) = [t]
unop (_:_) = []
unop [] = []
```