Summary

Logical (and versus functional)

2020

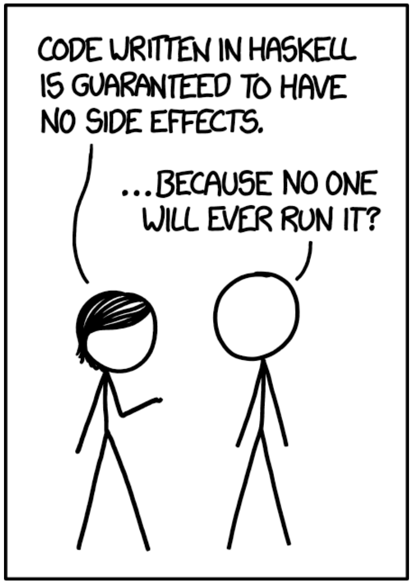
Instructor: Odelia Schwartz

Programming paradigms

- Imperative
- Functional
- Logical

Functional languages

- Imperative: based on Von Neumann
- Functional: based on mathematical functions
- Important feature of functional: no side effects; no variables; no states
- Last decade: increase in interest and use of functional languages. What languages?



Functional languages

- Last decade: increase in interest and use of functional languages. What languages?

  (Scheme)
  ML
  Haskell
  F#
  Scheme / Lisp
  Clojure

Programming paradigms

- Imperative
- Functional
- Logical

Logical programs: declarative rather than procedural
Only desired results (and collections of facts and rules) specified, rather than detailed procedure for producing Results

Syntax and semantics very different from imperative

Towards logical languages: applications

- Relational Database Management Systems
  e.g., Structured Query Database (SQL) is non procedural (tables of information; relations between tables)

- Expert systems
  Designed to emulate user expertise; lots of facts and relations in databases. Use inference rules to infer new facts. Example: with Prolog

Towards logical languages: applications

Fairly recent example: IBM Watson won jeopardy challenge

https://www.cs.miami.edu/home/odelia/teaching/csc419_spring20/syllabus/IBM_Watson_Prolog.pdf

Natural Language Processing With Prolog in the IBM *Watson* System

Adam Lally
IBM Thomas J. Watson Research Center
Paul Fodor Stony Brook University
24 May 2011

https://www.youtube.com/watch?v=P18EdAKuC1U

Scheme list functions: car, cdr

- car takes a list and returns first element

(car ‘(a b c))	returns a
(car ‘((a b) c d))	returns (a b)
(car ‘a)	error since a is not a list
(car ‘(a))	returns a
(car ‘())	error….

Scheme list functions: car, cdr

- cdr takes a list and returns list after removing first element

(cdr ‘(a b c))	return (b c)
(cdr ‘((a b) c d))	returns (c d)
(cdr ‘a)	error
(cdr ‘(a))	returns ()
(cdr ‘())	error

Scheme: append

```
(define (append lis1 lis2)
(cond
 ((null? lis1) lis2)
   (else (cons (car lis1)
     (append (cdr lis1) lis2)))
))
```

- Repeatedly place elements of first list into second list

Scheme: append

```
(define (append lis1 lis2)
(cond
 ((null? lis1) lis2)
   (else (cons (car lis1)
     (append (cdr lis1) lis2)))
))
```

- Reminding ourselves of cons (run it on csi):

(cons ‘(a b) ‘(c d))

(cons ‘((a b) c) ‘(d (e f)))

"

ML list operations

- hd, tl are ML’s version of Scheme CAR, CDR
- Literal lists in brackets [3,5,7]; [] empty list
- :: used for cons

4::[3,5,7] evaluates to?

[4,3,5,7]

ML append

```
fun append ([],lis2) = lis2
| append(h::t,lis2) =
h::append(t,lis2);
```

ML versus Scheme append

```
fun append ([],lis2) = lis2
| append(h::t,lis2) =
h::append(t,lis2);
```

```
(define (append lis1 lis2)
(cond
 ((null? lis1) lis2)
   (else (cons (car lis1)
     (append (cdr lis1) lis2)))
))
```

Haskell

Some list capabilities:

```
main :: IO ()
main =  do
        print([1,3..])
```

Keeps going infinitely…

In practice lazy; can use as much as you want

Haskell

Lazy evaluation – let's run some code

```
squares = [n*n | n <- [0..]]

main :: IO ()
main =  do
  print(squares)
```

Infinite…

Haskell

Lazy evaluation – let's run some code

Compare to:

```
squares = [n*n | n <- [0..5]]

main :: IO ()
main =  do
  print(squares)
```

Not infinite…

Haskell

Some list capabilities:

```
main :: IO ()
main =  do
        print(5:[2,7,9])
```

Like cons

Haskell

Some list capabilities:

```
main :: IO ()

main =  do
      print(head ([2,7,9]))
```

Like car

Prolog

- File lists_simple2.pl

```
new_list([H|T], H, T).
```

Example:

```
?- new_list([apple,prune,grape,kumquot],X,Y).
X = apple,
Y = [prune, grape, kumquot].
```

Returns head and tail

Prolog

- File lists_simple2.pl

```
new_list([H|T], H, T).
```

Example:

```
?- new_list(X,apple,[prune, grape, kumquot]).
X = [apple, prune, grape, kumquot].
```

Constructs list

Prolog

- File lists_simple2.pl

```
new_list([H|T], H, T).
```

Example:

```
?- new_list([apple,prune,grape,kumquot],X,Y).
X = apple,
Y = [prune, grape, kumquot].
```

Returns head and tail

```
?- new_list(X,apple,[prune, grape, kumquot]).
X = [apple, prune, grape, kumquot].
```

Constructs list

Prolog

- File lists_simple2.pl

```
new_list([H|T], H, T).
```

```
?- new_list([apple,prune,grape,kumquot],prune,
[prune, grape, kumquot]).
```

Returns false.

Haskell

append: Heads and tails and cons like...

```
append ([],lis2) = lis2
append(h:t,lis2) = h:append(t,lis2)

main :: IO ()
main =  do
print(append([1..3],[4..6]))
```

Compare to ML

Haskell:

```
append ([],lis2) = lis2
append(h:t,lis2) = h:append(t,lis2)
```

ML:

```
fun append ([],lis2) = lis2
| append(h::t,lis2) = h::append(t,lis2)
```

Compare to ML, Scheme

Haskell:

```
append ([],lis2) = lis2
append(h:t,lis2) = h:append(t,lis2)
```

ML:

```
fun append ([],lis2) = lis2
| append(h::t,lis2) =
h::append(t,lis2)
```

Scheme:

```
(define (append lis1 lis2)
(cond
 ((null? lis1) lis2)
   (else (cons (car lis1)
     (append (cdr lis1) lis2)))
))
```

Haskell

Quicksort

```
sort [] = []
sort (h:t) =
       sort [b | b <- t, b <= h]
       ++ [h] ++
       sort [b | b <- t, b > h]

main :: IO ()
main =  do
       print([1, 2] ++ [3,4])
       print(sort [25, 1, 3])
       print(sort [9, 6, 25, 1, 3])
```

Compare to some imperative languages!

```
 * A Pascal quicksort.
 ****************************************************************************}
PROGRAM Sort(input, output);
    CONST
        { Max array size. }
        MaxElts = 50;
    TYPE
        { Type of the element array. }
        IntArrType = ARRAY [1..MaxElts] OF Integer;

    VAR
        { Indexes, exchange temp, array size. }
        i, j, tmp, size: integer;

        { Array of ints }
        arr: IntArrType;

    { Read in the integers. }
    PROCEDURE ReadArr(VAR size: Integer; VAR a: IntArrType);
        BEGIN
            size := 1;
            WHILE NOT eof DO BEGIN
                readln(a[size]);
                IF NOT eof THEN
                    size := size + 1
            END
        END;

           PROCEDURE QuicksortRecur(start, stop: integer);
            VAR
                m: integer;

                { The location separating the high and low parts. }
                splitpt: integer;

            { The quicksort split algorithm.  Takes the range, and
              returns the split point. }
            FUNCTION Split(start, stop: integer): integer;
                VAR
                    left, right: integer;       { Scan pointers. }
                    pivot: integer;             { Pivot value. }

                { Interchange the parameters. }
                PROCEDURE swap(VAR a, b: integer);
                    VAR
                        t: integer;
                    BEGIN
                        t := a;
                        a := b;
                        b := t
                    END;

                BEGIN { Split }
                    { Set up the pointers for the hight and low sections, and
                      get the pivot value. }
                    pivot := arr[start];
                    left := start + 1;
                    right := stop;

                    { Look for pairs out of place and swap 'em. }
                    WHILE left <= right DO BEGIN
                        WHILE (left <= stop) AND (arr[left] < pivot) DO
                            left := left + 1;
                        WHILE (right > start) AND (arr[right] >= pivot) DO
                            right := right - 1;
                        IF left < right THEN
                            swap(arr[left], arr[right]);
                    END;

                    { Put the pivot between the halves. }
                    swap(arr[start], arr[right]);

                    { This is how you return function values in pascal.
                      Yeccch. }
                    Split := right
                END;

            BEGIN { QuicksortRecur }
                { If there's anything to do... }
                IF start < stop THEN BEGIN
                    splitpt := Split(start, stop);
                    QuicksortRecur(start, splitpt-1);
                    QuicksortRecur(splitpt+1, stop);
                END
            END;

        BEGIN { Quicksort }
            QuicksortRecur(1, size)
        END;

    BEGIN
        { Read }
        ReadArr(size, arr);

        { Sort the contents. }
        Quicksort(size, arr);

        { Print. }
        FOR i := 1 TO size DO
            writeln(arr[i])
    END.
```

Source: http://sandbox.mc.edu/~bennet/cs404/doc/qsort_pas.html

```
    /*************************************************************************
     *  Quicksort code from Sedgewick 7.1, 7.2.
     *************************************************************************/
    public static void quicksort(double[] a) {
        shuffle(a);                           // to guard against worst-case
        quicksort(a, 0, a.length - 1);
    }

    // quicksort a[left] to a[right]
    public static void quicksort(double[] a, int left, int right) {
        if (right <= left) return;
        int i = partition(a, left, right);
        quicksort(a, left, i-1);
        quicksort(a, i+1, right);
    }

    // partition a[left] to a[right], assumes left < right
    private static int partition(double[] a, int left, int right) {
        int i = left - 1;
        int j = right;
        while (true) {
            while (less(a[++i], a[right]))      // find item on left to swap
                ;                               // a[right] acts as sentinel
            while (less(a[right], a[--j]))      // find item on right to swap
                if (j == left) break;           // don't go out-of-bounds
            if (i >= j) break;                  // check if pointers cross
            exch(a, i, j);                      // swap two elements into place
        }
        exch(a, i, right);                      // swap with partition element
        return i;
    }

    // is x < y ?
    private static boolean less(double x, double y) {
        comparisons++;
        return (x < y);
    }

    // exchange a[i] and a[j]
    private static void exch(double[] a, int i, int j) {
        exchanges++;
        double swap = a[i];
        a[i] = a[j];
        a[j] = swap;
    }

    // shuffle the array a[]
    private static void shuffle(double[] a) {
        int N = a.length;
        for (int i = 0; i < N; i++) {
            int r = i + (int) (Math.random() * (N-i));   // between i and N-1
            exch(a, i, r);
        }
    }



    // test client
    public static void main(String[] args) {
        int N = Integer.parseInt(args[0]);

        // generate N random real numbers between 0 and 1
        long start = System.currentTimeMillis();
        double[] a = new double[N];
        for (int i = 0; i < N; i++)
            a[i] = Math.random();
        long stop = System.currentTimeMillis();
        double elapsed = (stop - start) / 1000.0;
        System.out.println("Generating input:  " + elapsed + " seconds");

        // sort them
        start = System.currentTimeMillis();
        quicksort(a);
        stop = System.currentTimeMillis();
        elapsed = (stop - start) / 1000.0;
        System.out.println("Quicksort:   " + elapsed + " seconds");

        // print statistics
        System.out.println("Comparisons: " + comparisons);
        System.out.println("Exchanges:   " + exchanges);
    }
}
```

Source: http://www.cs.princeton.edu/introcs/42sort/QuickSort.java.html

Prolog

```
% ['likes.pl'].
% Based on sebesta book
% control d, to exit

likes(jake,chocolate).
likes(jake,apricots).
likes(jake,bananas).
likes(darcie,licorice).
likes(darcie,apricots).
likes(darcie,bananas).
```

In compiler type:

```
[‘likes.pl’].
trace.
likes(jake,X), likes(darcie,X).
```

Prolog

In compiler type:
[‘likes.pl’].
trace.
likes(jake,X), likes(darcie,X).

```
    Call: (7) likes(jake, _G1097) ? creep
    Exit: (7) likes(jake, chocolate) ? creep
    Call: (7) likes(darcie, chocolate) ? creep
    Fail: (7) likes(darcie, chocolate) ? creep
    Redo: (7) likes(jake, _G1097) ? creep
    Exit: (7) likes(jake, apricots) ? creep
    Call: (7) likes(darcie, apricots) ? creep
    Exit: (7) likes(darcie, apricots) ? creep
X = apricots ;
```

Prolog

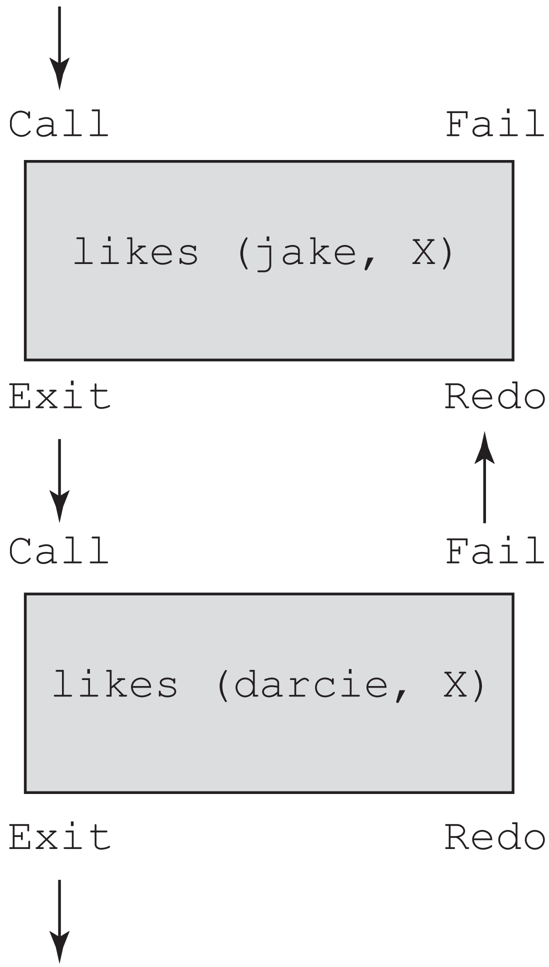
In compiler type:
[‘likes.pl’].
trace.
likes(jake,X), likes(darcie,X).

(after ;)

```
X = apricots ;
  Redo: (7) likes(darcie, apricots) ? creep
  Fail: (7) likes(darcie, apricots) ? creep
  Redo: (7) likes(jake, _G1097) ? creep
  Exit: (7) likes(jake, bananas) ? creep
  Call: (7) likes(darcie, bananas) ? creep
  Exit: (7) likes(darcie, bananas) ? creep
X = bananas.
```

Prolog

Control flow model for
likes(jake,X), likes(darcie,X)



Prolog

List structure

- Prolog uses syntax of ML and Haskell to specify lists
- Example: [apple, prune, grape, kumquat]
  [ ] empty list
- Prolog also has head and tail:

  [x | y]

  denotes a list with head x and tail y
- Similar to? Most similar to Haskell (x : y) and ML (x :: y) format. Also conceptually related to car, cdr of Scheme.

Append full function:

append([], List, List)

Lists to be appended — result

When empty list appended to any other List, the other List is the result

```
append ( [Head | List_1], List_2, [Head | List_3]) :-
append(List_1, List_2, List_3)
```

Lists to be appended — result

In the recursive step, this implication is essentially adding The same **Head** to the first given list, and to the resulting third new list

Prolog

- **append**

```
append([], List, List)
append ( [Head | List_1], List_2, [Head | List_3]) :-
append(List_1, List_2, List_3)
```

```
[trace]  ?- append([bob,jo],[jake, darcie],Family).
   Call: (6) append([bob, jo], [jake, darcie], _G1110) ? creep
   Call: (7) append([jo], [jake, darcie], _G1189) ? creep
   Call: (8) append([], [jake, darcie], _G1192) ? creep
   Exit: (8) append([], [jake, darcie], [jake, darcie]) ? creep
   Exit: (7) append([jo], [jake, darcie], [jo, jake, darcie]) ? creep
   Exit: (6) append([bob, jo], [jake, darcie], [bob, jo, jake, darcie]) ? creep
Family = [bob, jo, jake, darcie].
```

Prolog

Append in ML reminder:

```
fun append ([],lis2) = lis2
| append(h::t,lis2) = h::append(t,lis2);
```

Scheme…

```
(cons (car list1) (append (cdr list1) list2))
```

Head of first — append remaining of first with second

Prolog:

```
append([], List, List)
append ( [Head | List_1], List_2, [Head | List_3]) :-
append(List_1, List_2, List_3)
```

Prolog append more flexible than Scheme/ML!

```
append([], List, List)
append ( [Head | List_1], List_2, [Head | List_3]) :-
append(List_1, List_2, List_3)
```

Let's try:

- append(X,Y,[a,b,c]).

Returns:

X = []
Y = [a,b,c]

X = [a]
Y = [b,c]

X = [a,b]
Y = [c]

X = [a,b,c]
Y = []

Prolog

- Member: Is Element a member of List?

```
Member( Element, [Element | _ ] ).
Member(Element, [_ | List] ) :- member(Element, List).
```

Overall:

First statement base condition: If Element is in the head of the list, succeeds (true)

Recursion: If Element is in the tail of the list, then recurse on the tail of the list

Prolog

- Member: Is Element a member of List?

```
Member( Element, [Element | _ ] ).
Member(Element, [_ | List] ) :- member(Element, List).
```

Compare to Scheme:

```
(define (member atm lis)
  (cond
    ((null? lis) #f)
    ((eq? atm (car lis)) #t)
    (else (member atm (cdr lis)))
  )
)
```

Prolog

- Member: Is Element a member of List?

```
Member( Element, [Element | _ ] ).
Member(Element, [_ | List] ) :- member(Element, List).
```

Compare to Haskell:

```
member n (m:x)
   | x==[] = False
   | m/=n = member n x
   | m==n = True
   | otherwise = False
main :: IO ()
main =  do
   print(member 5 [3,7,6])
```

Prolog

- Member: let's do trace on Prolog.

trace.
member(a,[b,a,c]).

[trace] ?- member(a,[b,a,c]).
   Call: (6) member(a, [b, a, c]) ? creep
   Call: (7) member(a, [a, c]) ? creep
   Exit: (7) member(a, [a, c]) ? creep
   Exit: (6) member(a, [b, a, c]) ? creep
true .

Returns true

Member(Element, [Element | _]).
Member(Element, [_ | List]) :- member(Element, List).

Prolog

- Member: let’s do trace on Prolog.

```
?- trace.
[trace]  ?- member(a,[b,c,d]).
   Call: (6) member(a, [b, c, d]) ? creep
   Call: (7) member(a, [c, d]) ? creep
   Call: (8) member(a, [d]) ? creep
   Call: (9) member(a, []) ? creep
```

recursion on tail


```
Member( Element, [Element | _ ] ).
Member(Element, [_ | List] ) :- member(Element, List).
```

Prolog

- Member: let's do trace on Prolog.

```
?- trace.
[trace]  ?- member(a,[b,c,d]).
   Call: (6) member(a, [b, c, d]) ? creep
   Call: (7) member(a, [c, d]) ? creep
   Call: (8) member(a, [d]) ? creep
   Call: (9) member(a, []) ? creep
   Fail: (9) member(a, []) ? creep
   Fail: (8) member(a, [d]) ? creep
   Fail: (7) member(a, [c, d]) ? creep
   Fail: (6) member(a, [b, c, d]) ? creep
false.
```

Failures

Member(Element, [Element | _]).
Member(Element, [_ | List]) :- member(Element, List).

Prolog

- Member: let's do trace on Prolog.

```
?- member(X,[a,b,c]).
```

Answer?


```
Member( Element, [Element | _ ] ).
Member(Element, [_ | List] ) :- member(Element, List).
```

Prolog

- Member: let’s do trace on Prolog.

?- member(X,[a,b,c]).

Answer?

X = a ;
X = b ;
X = c ;

Member(Element, [Element | _]).
Member(Element, [_ | List]) :- member(Element, List).

Prolog

- Assignment hint

For the union of two lists, you need to consider what happens if the head of the first list is a member of the second list, and what happens when it is not. You are allowed to use the built in Prolog member function.

First, consider the base of the recursion.
Then, for the actual recursion: consider for List1, List2, List3, what happens when the right side of the implication includes member(H, List2) and union(List1, List2, List3) (what is the left side of the implication?).
Then consider what the implication should be when H is not a member of List2.

Prolog

Questions?