Programming Languages
functional part 6

2020

Instructor: Odelia Schwartz

Functional declarations ML

- Format:

  fun name (parameters) = body;

ML Language

https://www.tutorialspoint.com/execute_smlnj_online.php

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 list operations

- Number of elements in a list

```
fun length([]) = 0
| length(h::t) = 1 + length(t);
length([1,3,5])
```

ML list operations

- Append function

```
fun append ([],lis2) = lis2
| append(h::t,lis2) = h::append(t,lis2);
append([1,2],[3,4]);
```

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)))
))
```

ML list operations

- Let’s each try fun adder

  adder([1,2,3]) should return 6

ML list operations

- Let's each try fun adder

```
fun adder([]) = 0

| adder (h::t)=h+adder(t);

adder([1,2,3,4,5]);
```

Names bound to values (constants)

- Format:

```
val new_name = expression;
```

Names bound to values (constants)

- Format:

val new_name = expression;

Binds the value to name once and cannot be rebound (nothing like an assignment statement in an imperative language!)

"

Names bound to values (constants)

- Format:

```
val new_name = expression;
```

Example: usually used with a let statement:

```
fun area(radius) =
let val radius = 2.7
    val pi = 3.14159
in pi*radius*radius
end;
```

Higher order functions

- map

```
map(fn x =>x*x*x)[1,3,5];
```

Higher order functions

- map

```
map(fn x =>x*x*x)[1,3,5];
```

Note: different interpreters have slightly different notation; book notation different

Higher order functions

- Composing two functions

h = f o g

(lower case o)

Higher order functions

- Composing two functions

h = f o g

Example: (run it)

```
fun times10(x) = 10*x;
times10(5);
fun plus3(y) = 3 + y;
plus3(4);
val h = times10 o plus3;
h(7)
```

Currying

f(a,b)

- Function with more than one parameters, essentially takes one parameter at a time

Currying

f(a,b)

- Function with more than one parameters, essentially takes one parameter at a time
- For function **f(a,b)**, first replace **a** with its parameter value

Currying

f(a,b)

- Function with more than one parameters, essentially takes one parameter at a time

- For function **f(a,b)**, first replace **a** with its parameter value

- Given **a**, results in function **f(b)**

Currying

f(a,b)

- Function with more than one parameters, essentially takes one parameter at a time
- For function **f(a,b)**, first replace **a** with its parameter value
- Given **a**, results in function **f(b)**

Currying

fun add a b = a + b

- Function with more than one parameters, essentially takes one parameter at a time
- For function **f(a,b)**, first replace **a** with its parameter value
- Given **a**, results in function **f(b)**

Why would we want this?

Currying

fun add a b = a + b

- Currying interesting because allows to create new or partial functions where not all parameters are provided

- **fun add5 x = add 5 x**

Currying

Let’s try to run this:

```
fun add a b = a + b
fun add5 x = add 5 x
val num = add5 10
```

Currying

Let's try to run this:

```
fun add a b = a + b
fun add5 x = add 5 x

val num = add5 10
```

Currying

Let’s try to run this:

```
fun add a b = a + b
fun add5 x = add 5 x
val num = add5 10

fun times10(x) = 10*x;
times10(5);
```

Currying

Let’s try to run this:

```
fun add a b = a + b
fun add5 x = add 5 x
val num = add5 10

fun times10(x) = 10*x;
times10(5);

val h = times10 o add5;
h(7)
```

Curried functions also in Scheme

```
(define (add x y) (+ x y))
```

Curried:

```
(define (add y) (lambda (x) (+ y x)))
```

Curried functions also in Scheme

```
(define (add x y) (+ x y))
```

Curried:

```
(define (add y) (lambda (x) (+ y x)))
```

```
((add 3) 4)
```

- Like first calling with param 3 and then 4

Try csi…

ML

- Impact on Haskell, Ocam, F#
- ML is not purely functional; has mutable arrays

Haskell

- Truly functional

Haskell

- Truly functional
- Similar to ML: in syntax, strongly typed with type inferencing

Haskell

- Truly functional
- Similar to ML: in syntax, strongly typed with type inferencing (but a little more flexible than ML)
- Different from ML: entirely functional with no side effects; functions can be overloaded; lazy (more later)

Haskell

https://www.jdoodle.com/execute-haskell-online

```
square x = x * x

main :: IO ()
main =  do
      putStr "Square of x = "
      print(square 10)
      print(square 10.2)
```

Haskell

https://www.jdoodle.com/execute-haskell-online

```
square x = x * x

main :: IO ()
main =  do
      putStr "Square of x = "
      print(square 10)
      print(square 10.2)
```

Haskell

https://www.jdoodle.com/execute-haskell-online

```
square x = x * x

main :: IO ()
main =  do
      putStr "Square of x = "
      print(square 10)
      print(square 10.2)
```

Unlike ML can take any type!

Haskell

Some list capabilities:

```
main :: IO ()
main =  do
      print([1,3..11])
```

Haskell

Some list capabilities:

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

Keeps going infinitely…

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

Some list capabilities:

```
main :: IO ()
main =  do
      print([1..20])
```

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

Haskell

Some list capabilities:

```
main :: IO ()

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

Like car
Works with and without
Function parentheses

Haskell

Some list capabilities:

```
main :: IO ()

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

Like cdr
Works with and without
Function parentheses

Haskell

Some list capabilities:

```
main :: IO ()
main =  do
     print([1..4]++[5..8])
```

append

Pattern Matching

Differences from ML:

- No reserved words to introduce functions
- Some other format differences

Pattern Matching

Example:

```
fact 0 = 1
fact 1 = 1
fact n = n * fact(n-1)
```

Pattern Matching

Example:

```
fact 0 = 1
fact 1 = 1
fact n = n * fact(n-1)

main :: IO ()
main =  do
  print(fact(4))
```

Haskell versus ML

Pattern Matching

Haskell:

```
fact 0 = 1
fact 1 = 1
fact n = n * fact(n-1)
```

ML:

```
fun fact(0) = 1
  | fact(1) = 1
  | fact(n:int) = n*fact(n-1);
```

Pattern Matching

Could also do with guards:

```
fact n
   | n==0 = 1
   | n==1 = 1
   | n>1 = n*fact(n-1)

main :: IO ()
main =  do
   print(fact(4))
```

Haskell versus ML

Pattern Matching

Haskell:

```
fact n
   | n==0 = 1
   | n==1 = 1
   | n>1 = n*fact(n-1)
```

ML:

```
fun fact(0) = 1
   | fact(1) = 1
   | fact(n:int) = n*fact(n-1);
```

Pattern Matching

Could also do with otherwise:

```
fact n
   | n==0 = 1
   | n==1 = 1
   | otherwise = n*fact(n-1)

main :: IO ()
main =  do
  print(fact(4))
```

Pattern Matching

Could also do with otherwise:

```
sub n
   | n<10 = 0
   | n>100 = 2
   | otherwise = 1

main :: IO ()
main =  do
  print(sub(1))
```

Haskell

lengths: heads and tails and cons like...

```
products [] = 1
products(a:x) = a * products x
fact n = products[1..n]


main :: IO ()
main =  do
       print(fact 5)
```

Haskell

lengths: eads and tails and cons like...

```
lengths [] = 0
lengths (h:t) = 1 + lengths t

main :: IO ()
main =  do
     print(lengths[1,6,3,9,5])
```

Haskell

lengths: heads and tails and cons like...

```
lengths [] = 0
lengths (h:t) = 1 + lengths t

main :: IO ()
main =  do
        print(lengths[5..10])
```

Haskell

lengths: Heads and tails and cons like…

```
lengths [] = 0
lengths (h:t) = 1 + lengths t

main :: IO ()
main =  do
      print(lengths[5..10])
```

sums: Heads and tails and cons like...

Let's all write a function…

Haskell

sums: Heads and tails and cons like...

```
sums [] = 0
sums (h:t) = h + sums t

main :: IO ()
main =  do
      print(sums[1..4])
```

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)))
))
```

List Comprehensions

```
cube = [n*n*n | n<-[1..50]]

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

Haskell

What kind of sort is this?

```
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

Haskell

Compare to some imperative languages!
Haskell much more elegant...

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

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

Haskell

Lazy evaluation

- Allow infinite lists
- Expressions only evaluated if needed

Haskell

Lazy evaluation – let's run some code

```
positives = [0..]

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

Haskell

Lazy evaluation – let’s run some code

```
evens = [2,4..]

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

Haskell

Lazy evaluation – let's run some code

```
evens = [2,4..]

main :: IO ()
main =  do
  print(head evens)
```

Haskell

Lazy evaluation – let's run some code

```
evens = [2,4..]

main :: IO ()
main =  do
  print(tail evens)
```

Haskell

Lazy evaluation – let's run some code

```
evens = [2,4..]

main :: IO ()
main =  do
  print(evens!!1)            -- first element
```

Haskell

Lazy evaluation – let's run some code

```
evens = [2,4..]

main :: IO ()
main =  do
  print(evens!!3)          -- third element
```

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

We'll also use:

```
member n (m:x)
   | m<n = member n x
   | m==n = True
   | otherwise = False

main :: IO ()
main =  do
   print(member 2 [1..4])
```

Haskell

We'll also use:

```
member n (m:x)
   | m<n = member n x
   | m==n = True
   | otherwise = False

main :: IO ()
main =  do
   print(member 2 [1..4])
   print(member 2 [1,3..])
```

Haskell

Lazy evaluation – let’s run some code

```
squares = [n*n | n <- [0..]]
member n (m:x)
  | m<n = member n x
  | m==n = True
  | otherwise = False

main :: IO ()
main =  do
  print(member 16 squares)

  print(member 15 squares)
```

Only evaluates
what is needed

Haskell

Lazy evaluation – let's run some code

```
squares = [n*n | n <- [0..]]
member n (m:x)
  | m<n = member n x
  | m==n = True
  | otherwise = False

main :: IO ()
main =  do
  print(member 16 squares)

  print(member 15 squares)
```

What is this doing?

Haskell

Lazy evaluation – let's run some code

```
squares = [n*n | n <- [0..]]
member n (m:x)
   | m<n = member n x
   | m==n = True
   | otherwise = False

main :: IO ()
main =  do
   print(member 16 squares)

   print(member 15 squares)
```

Checking if number
can be expressed as n*n
[0,1,4,9,16,25,36,49,64,81,100,121…