2016.01.11
COMP302 – self-taught the languages F#
learn coding from imitating other's code
introduction to F#
What is it?
a functional language but it also has imperative, object-oriented and other features.
based on SML, but with some syntactic changes and integrated into the .NET framework.
standard meta language is not standard markup language.
Basic types:
integers, booleans, characters and strings, floats(floating point numbers)
compound types: allow you new type from old ones
unit, tuples, records(things go in it are named, easier to access than tuples), lists(an inductively defined type), user-defined inductive type.

' ' is char, " " is string
bindings: use "let", let name = expression
beware of F# indentation rules.
learn about verbose vs lightweight syntax

2016.01.11
recursion, list structures(inductive structure)

<u>In order to write recursion programs: don't think about program execution in recursion.</u>

```
let rec fact n = if n=0 item 1 else
                    n*fact(n-1).
```

how to think of it? **(these are the recursion principles!)**

1. there must be an exit. Make sure it will stop. in this case, n=0
2. the recursive call must make progress towards exit.
3. assume the recursive call works! Then show(convince yourself) that everything else works.

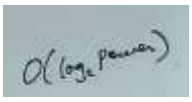
code sample to look at:

```
let iterfact n=
        let rec helper(x,y) = //y is accumulating parameter
                if x=0 then y
                else
                helper(x-1,x*y) //this is a tail recursive
        herper(n,1) // tail-recursive call outermost
```

//rundown of this program: it takes two arguments in the helper function and set the exit to be x=0 then y, else x decrements and stores the value into y

code sample: (Russian peasants exponential)

```
let rec rpe(b,power)=
        if (b=0) then 0
        elif (power =0) then 1
        elif (power =1) then b
        elif (power%2= 1) then b*rpe(b,power-1)
else
        let temp = rpe(b,power/2) in
                temp*temp       //this is
```




```
//why do we not write rpe(b,power/2)* rpe(b,power/2)
//because it is cutting the work in half, it is not an improvement
```

<u>HOW DO WE KNOW THIS WORK?</u>

<u>we assume the recursive call works by magic and list out the conditions and exit.</u>

2016.01.11
recursion, list structures(inductive structure)

next example:
compute a square root of 9, first we guess it is 1^2 which gives 9-1 = 8
the thinking behind this is

computing square roots in a fast way
improve -> 1/2(x/guess+guess)
x is 17, we start guessing from 1
½(17/1+1)=9
½(17/9+9)=5.49 ish, this is a better guess than 1.

code sample:

```
let delta =0.0001

let square u:float =u*u

let close guess x=(abs((square guess)-x)) < delta

let update guess x=(guess+x/guess)/2.0
//things like close and update shouldn't be publicly visible
//else the program will be vulnerable

let rec sqrt x guess=
        if (close guess x) then guess
        else (sqrt(update guess x) x)

let rec sqrt2(x, guess, error: float)=
        let close(guess, x)=(abs((square guess)-x)) < error

        let update(guess, x)=(guess+x/guess)/2.0

        if close(guess, x) then guess else sqrt2(x, update(guess, x), error)
```

what is a list(inductive structure)?
it is an item stored in a list

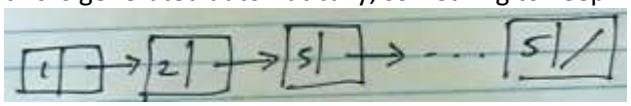
- empty: base case (no need to think inductively)
- item followed by a list of items

(gotta think of it inductively)
{a, b, c}: items
a::[] displays as [a]
b::[] displays as [b]
c::[] displays as [c]
now i have three lists, each of them are length 1
a**::**a::[] displays as [a;a]
(:: = list constructor: allows you to place an item at the front of a list only)

let's look at some examples

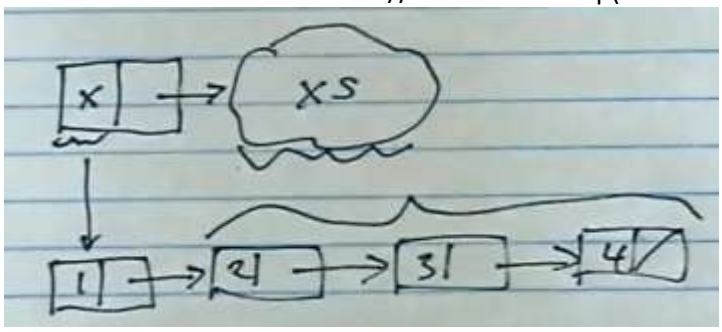
2016.01.11
recursion, list structures(inductive structure)

when typing in “let list = [1;2;3;4;5]
this is generated automatically, something to keep in mind



(
please use pattern matching to generate a list
instead of writing
let v = list.Head
let t = list.Tail
)
code sample:
if you use .head on an empty list you will get a runtime error

```
//this function takes two lists and zip them together,
//one from l1, one from l2, and repeat.
let rec badzip(l1,l2)=
        ifl1=[] then l2
        else
                (l1.Head)::(badzip (l2,l1.Tail))

let foo=badzip(liszt,liszt2)  //the pattern matching version

//this is a way better version than the badzip function
let rec zip(l1,l2)=
        match l1 with  //match it with one of the options
        | [] -> l2 //if l1 is empty then return l2
        | x:: xs -> x :: zip(l2,xs) //else take the first variable of l1 stick it to zip(l2,xs)
                                    //then the second loop will take the first variable of l2
                                    //and stick it to zip(the rest of l1, the rest of l2)
```



x:1
xs:[2;3;4]

code samples:

```
let l1=[1 .. 17] //this works, convenient, a list from 1 to 17

let l2=[1 ..2.. 17] //a list that increments by two from 1 to 17.
 (you can increment negatively as well)
```

2016.01.11
recursion, list structures(inductive structure)

code samples: **insertion sort**

```
let rec insert n list=
        match lst with   //let's match the list with something
        | [] -> [n]          //if there is nothing, stick the item n in the list
//conditions, you want the starting list to be sorted and the new list to be sorted
        | x:: xs -> if (n<x) then n::lst
                    else x::(insert n xs)
        //since x::xs is sorted already, if n is less than x then n is less than the rest,
        //which is stick to the front of the list
        //else n has to go for somewhere else, which goes to the magical recursion call.

let rec isort lst=
        match lst with
        | [] ->[]
        | x::xs -> insert x (isort xs)
        //you insert x into something,
        //what is something you ask?
        //it is isort xs, why? don't ask, it works by magic.
```

in the design phase you don't think of the stack and stuff, unless the program doesn't work, then you have to trace back and debug.

2016.01.13
comp 302:
more recusion functions on lists, append, list built-in functions, higher order function
option type
:: is a list constructor, it takes item x list produces(->) a new list
you **cant** use **[ ] :: item (type error)**
example:

```
let vowels = ['a';'e';'i';'o';'u']
let vowels2 = 'y' :: vowels   (add y in front of the vowels)
(* What if we want to add y at the end? vowels :: 'y' gives a type
error. which leads to append*)
```

:: creates a new cell and put the item there and create a pointer to point to other cell, O(1).

@ **append**(concatenate): list x list -> list
example:

```
let vowels3 = vowels @ ['y']
(* We had to make 'y' into a list (of length 1) by writing ['y]. *)
```

lists are being created.

example:

```
(* Our own append function. It is O(n), unlike cons which is O(1). *)
let rec myappend l1 l2 = [
  match l1 with
  | [] -> l2
  | x :: xs -> x :: (myappend xs l2) (recursive function)
```

example:
Reverse

```
(* Reverse done naively.  This is O(n^2). *)
let rec reverse l =
  (match l with
  | [] -> []
  | x::xs -> (reverse xs) @ [x])
this is slow but correct, now we are going to try again

idea:
cell 1 points c2 points to c3
reverse it to become
c1 being pointed by c2 which is being pointed by c3
a place where we are constructing the answer (accumulating parameter)
```

better reverse function with a helper function inside with an accumulator parameter

```
let rev l =
  let rec helper(l,acc) =
    match l with
    | [] -> acc (if empty list, I have finished traversing the list)
    | x :: xs -> helper(xs, x::acc)
        (else I have work to do,
        recursively call help my tail and take x::acc)
  helper(l,[]) //initialize acc to []
        //this runs in O(n)
```

2016.01.13
comp 302:
more recusion functions on lists, append, list built-in functions, higher order function
option type

```
(* Some built in functions: List.length, List.head, List.tail,
List.rev, List.zip.
The built in zip is not the same as my zip function.  It can take lists
of different types but they must have the same length.  *)
let pairs = List.zip odds evens
(* More interesting functions; these take functions as arguments. *)
(* These are all provided in Fsharp as primitives. *)
```

list.zip: takes a list x b list –> (a*b) list (-> = produces)

MAP(powerful function)
mymap: ('a->'b)->'a list -> 'b list
this is a higher order function, it takes a function and a list, give me a thing which gives b thing.
example:

```
let rec mymap f l =
  match l with
    | [] -> []
    | x :: xs -> f(x) :: (mymap f xs)
        //process each item (x) and build up a new list recursively.
```

this is a map on list

other examples of higher order function:
Filter: ('a-> bool) x 'a list -> 'a list
//give me a list of things, only the property i want gets pass by returning boolean values

```
let rec myfilt tester lst =
  match lst with
  | [] -> []
  | x :: xs ->
  if (tester x) then x :: (myfilt tester xs)
        (tester is boolean, return yes/no)
  else myfilt tester xs
```

Search
//search for some item in the list
//introduce **Option type to insure type safe**
//option type is a type constructor

```
let rec search item lst =
  match lst with
  | [] -> None (option types)
  | x :: xs ->
  if (x = item) then (Some x)
  else search item xs;;
```

option type:

```
if 'a is a type, option 'a is a new type, v is a value of type 'a
```

some v ∈ option 'a
None is in all option types

'a x 'a list -> option 'a

higher order function: the programme produces programme as answers

2016.01.13
comp 302:
more recusion functions on lists, append, list built-in functions, higher order function
option type

COMP302
2016.01.15 topics: set,list, fold, reduce, set/list comprehension, set/list/map collections
midterm is scheduled.
talk about list more

introduce new f# collection:
sets, it can be represent as [ ]
it is different from lists
read more about it online and will appear on A2

Map -> association
it provides a table of key and value
it is root elementary for data structures

now look at some functions

list.fold f e [x1,x2,…,xn]
= f(…(f(f(f(e,x1)x2)….,xn)
it combine e,x1 with x2 … xn
// Applies a function `f` to each element of the collection,
// threading an accumulator argument through the computation.
//`List.fold is the same as to List.iter when the accumulator is not used.`

reduce op [x0,x1,…,xn]
=op(x0, op(x1,op(x2,…, op(xn-1, xn)…)
// it takes an operator as parameter and reduce

myreduce op identify [x0,x1,…,xn]
= op(x0,op(x1,… op(xn,identify)…)

now look at some fun challenge function:
inter item list //it puts that item into every possible location in the list
= [[0;1;2;3];[1;0;2;3];[1;2;0;3];[1;2;3;0]]
//feel free to write this function on your free time if you can

back to look at examples:

```
(* This version of reduce is provided in Fsharp. *)

let rec myreduce op lst =
  match lst with
  | [] -> failwith "List argument cannot be empty"
  | x :: [] -> x
  | x :: xs -> op x (myreduce op xs)


(* I like this better.  It takes an identity element for the operation.
*)

let rec good_reduce op iden lst =
  match lst with
  | [] -> iden
  | x :: xs -> op x (good_reduce op iden xs)
```

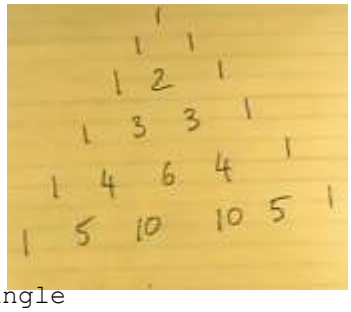
COMP302
2016.01.15 topics: set,list, fold, reduce, set/list comprehension, set/list/map collections
midterm is scheduled.

```
(*
val good_reduce : op:('a -> 'b -> 'b) -> iden:'b -> lst:'a list -> 'b
*)
(* > good_reduce (+) 0 (allnums 5);;
val it : int = 15
> good_reduce mult 1 (allnums 5);;
val it : int = 120
> let cons a b = a :: b;;
val cons : a:'a -> b:'a list -> 'a list
> let app3 a b = good_reduce cons b a;; //append
val app3 : a:'a list -> b:'a list -> 'a list
> app3 odds evens;; //put all odds in front of evens
val it : int list =
  [1; 3; 5; 7; 9; 11; 13; 15; 17; 19; 2; 4; 6; 8; 10; 12; 14; 16; 18;
20]
*)

(* Fold in Fsharp.  More powerful than reduce.
> List.fold;;
val it : (('a -> 'b -> 'a) -> 'a -> 'b list -> 'a) = <fun:clo@98-2>
*)
lets look at this function that uses list.fold

let allsame l = //are all the things in the list are the same
match l with
| [] -> true
| [x] -> true
| x::xs -> List.fold (fun foo y -> (foo && (x=y))) true xs
//test the equality of the things an return true
//generate an anonymous function, fun foo y
//which test if x is equal to y, y as the rest of the list
//which foo as true
// so if x=y is true && foo is true as stated, then returns true.

(* Example taken from Chris Smith's book. *)

let countVowels (str:string) =
  let charList = List.ofSeq str

  let accFunc (As, Es, Is, Os , Us) letter =
    if   letter = 'a' then (As + 1, Es, Is, Os, Us)
    elif letter = 'e' then (As, Es + 1, Is, Os, Us)
    elif letter = 'i' then (As, Es, Is + 1, Os, Us)
    elif letter = 'o' then (As, Es, Is, Os + 1, Us)
    elif letter = 'u' then (As, Es, Is, Os, Us + 1)
    else                   (As, Es, Is, Os, Us)

  List.fold accFunc (0,0,0,0,0) charList;;

(*
val countVowels : str:string -> int * int * int * int * int

> countVowels "Double double toil and trouble, Fire burn and cauldron
bubble.";;
val it : int * int * int * int * int = (3, 5, 2, 5, 6)*)
```

COMP302
2016.01.15 topics: set,list, fold, reduce, set/list comprehension, set/list/map collections
midterm is scheduled.
another topic: List comprehension

```
let listofSquares n = [ for i in 1 .. n do yield i * i ]
```

//i will get a list of squares of n from 1 to n



```
let listofCubes n = [ for i in 1 .. n -> i * i * i]
//i will get a list of cubes of n

let rec choose n m =
  if (m = 0) || (n = m) then 1
  else (choose (n - 1) (m - 1)) + (choose (n - 1) m)

let pascalRow n = [ for i in 0 .. n -> choose n i]
// this will produce the pascal row for pascal's triangle

(* The examples below are written so you can see how to write longer
examples
spread out over many lines. *)

let multiplesOf n =
  [
    for i in 1 .. 10 ->
      i * n
  ]

let l =
  [
    for i in 1 .. 20 do
      if i % 2 = 0 then
        yield -i
      else
        yield i
  ]


let primesUnder max =
//the function tells you all the prime numbers up to max
//by dividing by everything
//if factorsOfN is empty then yield n
  [
    for n in 2 .. max do
      let factorsOfN =
        [
          for i in 2 .. (n - 1) do
            if n % i = 0 then
              yield i
        ]
      if factorsOfN = [] then
        yield n
  ]
```

the results are on the next page.

COMP302
2016.01.15 topics: set,list, fold, reduce, set/list comprehension, set/list/map collections
midterm is scheduled.

```
(* Results:

val choose : n:int -> m:int -> int
val pascalRow : n:int -> int list

> pascalRow 9;;
val it : int list = [1; 9; 36; 84; 126; 126; 84; 36; 9; 1]
> pascalRow 8;;
val it : int list = [1; 8; 28; 56; 70; 56; 28; 8; 1]
>
val primesUnder : max:int -> int list

> primesUnder 100;;
val it : int list =
  [2; 3; 5; 7; 11; 13; 17; 19; 23; 29; 31; 37; 41; 43; 47; 53; 59; 61;
67; 71;
   73; 79; 83; 89; 97]
>

*)
```

```
we just talked about set collection,
now lets talk about map collection:
//here we have a list of pair
//map.ofList converts from list to map
//each pair consists a string and a integer
// remember map = (key, value)
let profs = Map.ofList [("Simon, 19);("Doug", 20);("Alex", 20)]

let result = Map.find "Simon" profs

//you can declare a new type
type AssocList = Map<string, int>
//this is useful later on for look up and it will appear on the
assignments.

let profs2:AssocList = Map.ofList [("Simon, 19);("Doug", 20);("Alex",
20)]
```

SO,

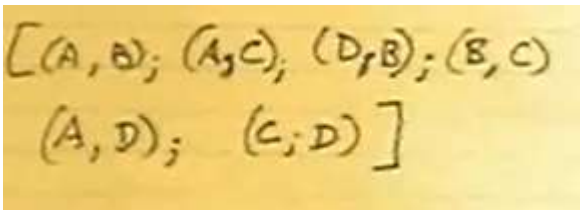
sets must contain elements for which an order is defined.

```
otherwise the compile error.

let candidate = set of ["chump";"lump";"trump"]
max -> "trump"
min -> "chump"
```

preview for last question on assignment 2: map coloring

```
given a list of countries,
given a list of pairs of countries
```

if you have 4 countries in a map then the list is
A column will be a set of countries.



comp302
2016.01.18 binary tree

Tree: they are like lists, they are inductively defined

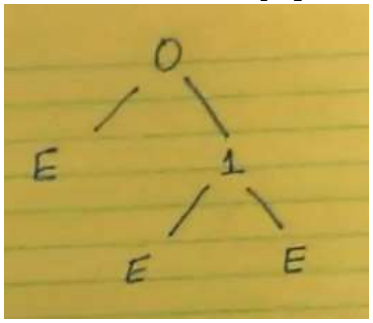
inductive definition of list = [ ] or item::lists
start with an empty list 0, list 1= { [] }, list 2= {[]}∪{item ::[]} …etc

Tree is built in in F#
Tree = empty + node(tree, item, tree)

```
type 'a tree = Empty | Node of 'a tree * 'a * 'a tree
```

*1= type variable, it allows you to define polymorphic trees
*2=made up word
*3=constructor function

```
let t1 = Node(Empty, 0, (Node(Empty, 1, Empty)))
```



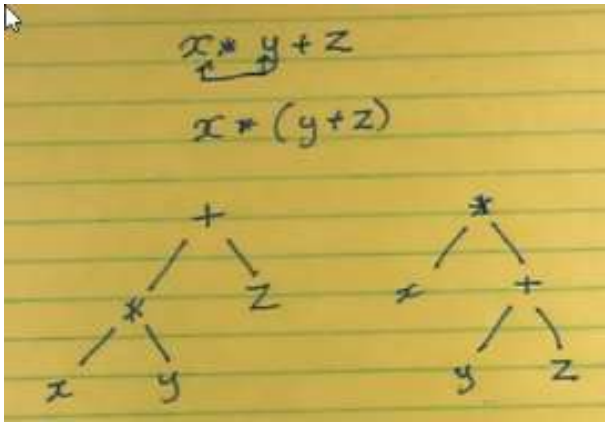

```
let rec height (t: 'a tree) =
  match t with
    |  Empty -> 0
    |  Node (l,_,r) -> 1 + max(height l, height r)

let rec sumNodes (t : int tree) =
  match t with
    | Empty -> 0
    | Node(l,n,r) -> n + sumNodes(l) + sumNodes(r)

let showInt n = printf "%i\n" n

let rec inOrder (t: int tree) =
  match t with
    | Empty -> printf " "
    | Node(l,n,r) -> inorder(l); showInt n; inorder(r)
```

(inorder traversal, empty=print blank, print left first then print right after)

```
let rec flatten (t: 'a tree) =
  match t with
    | Empty -> []
    | Node(lft,v,rt) -> v::((flatten(lft))@(flatten(rt)))
                  MATCH
      v                          17                    v|->17
L            R            t1            t2       L|-> t1      R|->t2
```

user-defined type are in component into type checker, pattern matching

comp302
2016.01.18 binary tree



expressions: (assignment 3)
type exp trees = | const of int
| var of char
| plus of exp tree * exp trees
| times of exp tress * exp trees

env: keeps track of values for variables
type Env = Map<char, int>

```
let rec eval(e : Exptree, rho: Env) =
  match e with
  | Const n -> n
  | Var v -> Map.find v rho
  | Plus (e1, e2) ->
      let v1 = eval(e1,rho)
      let v2 = eval(e2,rho)
      v1 + v2
  | Times (e1, e2) ->
      let v1 = eval(e1,rho)
      let v2 = eval(e2,rho)
      v1 * v2
```

```
Binary Search Trees (new type)
(BST: the root has to be bigger than everything on the right subtree
and bigger then left subtree, the same thing has to apply recursively)
type 'key bstree = Empty | Node of 'key * 'key bstree * 'key bstree

let rec find less x (t : 'key bstree) =    (*1: less is a boolean which
takes 2 arguments, x and t)
  match t with
    | Empty -> false  (if it is an empty tree)
    | Node(y,left,right) ->
        if (less(x,y)) then find less x left
        elif (less(y,x)) then find less x right
        else (* x = y *) true
```

deletermin: 'key bstree -> ('key * 'bstree)

```
let rec deletemin (t: 'key bstree) =
  match t with
    | Empty -> raise EmptyTree
    | Node(y,Empty,right) -> (y,right)
    | Node(y,left,right) ->
        let (z,L) = deletemin(left)
        (z,Node(y,L,right))
```

the tree looks like:

y

empty right

(in this case, y is the smallest, since there is no left subtree)

2016.01.20
comp 302 (mathematical part of the course)

syntax (tells you the structure, dictates it)
semantics => exploit structure to understand meaning structurally
evaluation of expressions
special expressions: values (the end point of evaluation)
example: 1,2,3,4… values
roman numerals
expressions example: 7+5
the process of evaluation converts the expression 7+5 to 12.
expressions can be nested arbitrarily: (7+5)*(11+4)
define expressions by induction
e ↓ v (when evaluation of e terminates the value v results
(↓=evaluates)
nano ML(only have expressions):this is a languag
syntax
e::==n|true|false|e1+e2| e1=e2|e1-e2
if e then e1 else 2|e1 and e2| not e

e-> if e then e1 else e2 -> if true then e3+e4 else e2
-> if true then 3+5 else 1729
semantics
e↓v by induction on e
base case: n↓n same with true↓true, false↓false
inductive cases: suppose e1↓n1 and suppose e2↓n2 then **e1+e2 ↓ n1+n2**
(they are different *1 with programming language, *2 in your understanding of arithmetic.)
suppose e↓true and e1↓v1 then (if e then e1 else e2)↓v1
suppose e↓false and e2↓v2 then (if e then e1 else e2)↓v2
Theorem the evaluation of every expressions in nano ML terminates.
(this is wrong because you cant say if 17 then something else)
the evaluation of every **well-typed** expressions in nano ML terminates. (this is correct)
Proof by induction on the structure of experiences.

micro ML: e::==n|true|false|e1+e2| e1=e2|e1-e2 **|x| let x=e1 in e2**
(the 2 new features, everything else is the same as nano ML)
take a look at let **x=e1** in **e2 (x is only defined with body of e2. *1 binding, *2 scope)**
if you look at e2 in isolation, x appears like an undefined variable
x appears free in e2
on the other hand , x appears bound in the whole let

let x= 1729 in }x and y both bound
let y = 4104 in } x free y bound }
{ x+y } x and y are both free } }

let **x=e** in e2 (**binder**)

2016.01.20
comp 302 (mathematical part of the course)

FV: exp -> set of free variables
FV(x)=0
FV(free)=0
FV(e1+e2)=FV(e1)∪FV(e2)
…
FV(if bexp then e1 else e2) =FV(bexp) ∪FV(e1) ∪FV(e2) (bexp=boolean expression)
FV(let x=e1 in e2) =FV(e1)∪(FV(e2)\{x}) (\=remove)

[we need a more detailed understanding to deconstruct this, next lecture perhaps]

```
let x=1 in
        let y=x in
                let x=2 in
                        x+y
```

COMP302
2016.01.22 some inductive proofs and stuff from the lecture before (jan.18th)

FV(let x=17
    let y=23
        x+Y) =0
if we break it up,
=(FV(let y =23 in x+Y) \ {x}) U FV(17) (*this is a constant which is 0)
=(FV(x+Y)\{y})\{x}
=((FV(x)U FV(y))) \ {x}=0
<u>definition:</u>
an expression with no free variables = closed term
the point of variables = show flow of information
bound variabl show connections
the actual name of a BV does not matter
    let x=1 in x+x ≡let y =1 in y+Y
the ability to change name in conversion is called <u>renaming or conversion</u>

$$\int sinxdx = integral\ sinydy, dy\ is\ binder, sin'y'is\ bound\ variable$$

let x = 2 in
    let y =x in
        let x=1 in
            y+x
this gives 3

<u>substitution</u>
let x=2 in
    let y =x+1 in -> let y =2+1 in
        y=x
this goes from x=2 -> y=3 in y*2 -> 6
this is dangerous, there's a trap

go back to the previous eg:
let x=2 in
    let y=x in
        let x=1 in -> 3
            y+x (naïve substitution will mess up binding strucute)
in substitution, only replace <u>FREE VARIABLES</u>

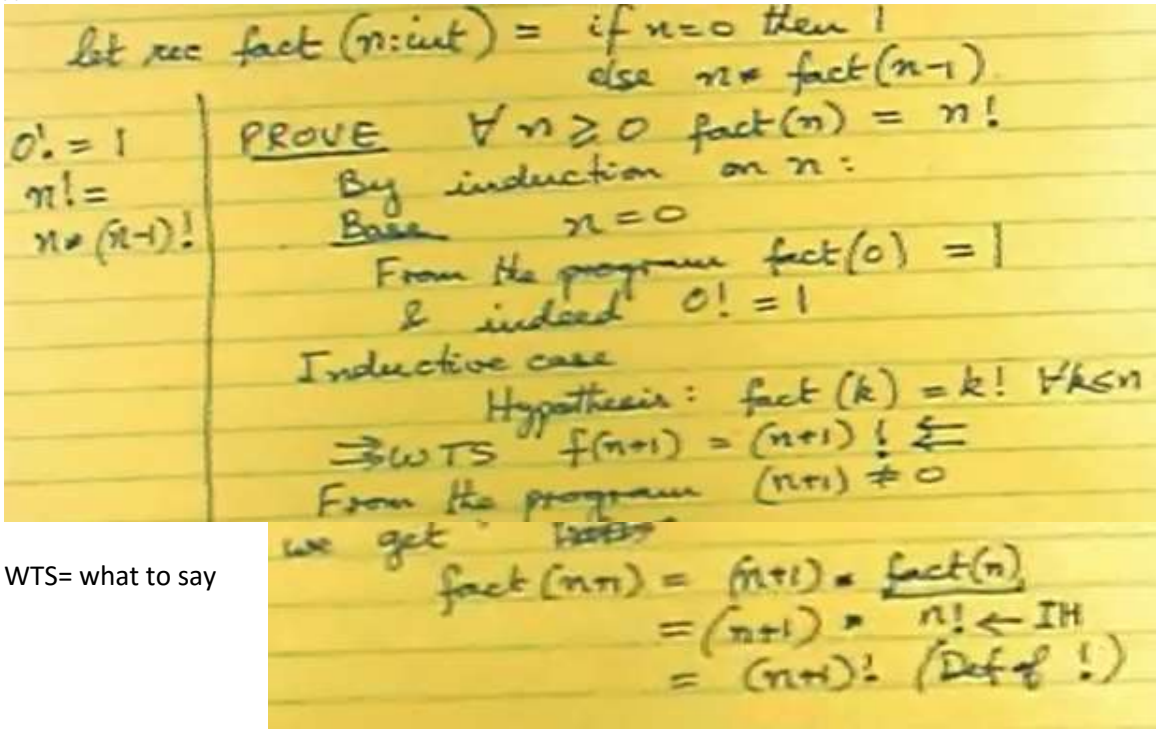
let x=2 in -> FV
caputure let y = x in
        let x=1 in
            y+x -> 3
whenever there's danger of capture
use renaming to avoid capture

COMP302
2016.01.22 some inductive proofs and stuff from the lecture before (jan.18th)

[e/x]e' replace FREE occurance of x in e' with e, if necessary rename variable to avoid capture.
[e/x]n ~> n
[e/x] true ~> true
[e/x] e1+e2~>([e/x]e1)+([e/x]e2)
[e/x]y~>{e x=y
{y x=/ y
replace e for x
[e/x] let u=e1 in e2 ~>
if u is in FV(e) item
let z be a fresh variable
let (z=[e/x]e1) in [e/x]([z/u]e2)
else let u=[e/x]e1 in [e/x]e2
operational semantics uML(micro ML)
e1↓v1 [v1/x]e2↓v2 (↓=evalutates)

(let x=e1 in e2) ↓ v2

thm: all well-typed expressions in uML terminates.
//INDUCTIVE PROOF



WTS= what to say

you assume the recursive call works, then check the base case, inductive case, inductive hypothesis.

Inductive Proofs of Program Behaviour (lecture on Jan 25, 2016)

NOTE: all code is NOT necessarily F# code, but the meaning should hopefully still be clear - if it isn't, please contact Tricia at patricia.olson@mcgill.ca with questions

```
let reverse l =
  match l with
  | [] -> []
  | x::xs -> append (reverse xs, [x])

let append (l1, l2) =
  match l1 with
  | [] -> l2
  | x::xs -> x::(append (xs, l2))
```

proof for append:

want: append $(l_1, l_2) = [x_1, \ldots, x_n, y_1, \ldots y_m]$
when $l_1 = [x_1, \ldots, x_n]$ and $l_2 = [y_1, \ldots, y_m]$

base case: $n = 0$

$l_1 = []$ append $(l_1, l_2) = l_2 = [y_1, \ldots, y_m]$ (match to 1st case) →

$l_2 = [y_1, \ldots, y_m]$

in all proofs we are examining the code to determine which case will be matched to

inductive step: assume append works for $|l_1| \leq n$, prove for $|l_1| = n+1$

$l_1 = [x_0, \ldots, x_n]$ → so we match to case 2:

$l_2 = [y_1, \ldots y_m]$ $x = x_0$ $xs = [x_1, \ldots x_n]$

and append $(l_1, l_2) = x_0 :: (\text{append}(xs, l_2))$

by I.H. append $(xs, l_2) = [x_1, \ldots x_n, y_1, \ldots y_m]$

so append $(l_1, l_2) = x_0 :: [x_1, \ldots x_n, y_1, \ldots y_m]$

$= [x_0, x_1, \ldots, x_n, y_1, \ldots, y_m]$ □

proof for reverse:

want: reverse $l = [x_n, \ldots x_1]$ if $l = [x_1, \ldots x_n]$

base case: $n = 0$

$l = []$ reverse $l = []$

induction step: assume reverse works for $|l| \leq n$, prove for $|l| = n+1$

$l = [x_0, x_1, \ldots, x_n]$ reverse l = append (reverse $[x_1, \ldots x_n]$, $[x_0]$) $x = x_0$, $xs = [x_1, \ldots x_n]$

by induction hypothesis (I.H.) reverse $[x_1, \ldots x_n] = [x_n, \ldots x_1]$

by proof of append, append $([x_n, \ldots x_1], [x_0]) = [x_n, \ldots x_1, x_0]$ □

```
let rev l = helper (l, [])

let helper (l, acc) =
  match l with
  | [] -> acc
  | x :: xs -> helper (xs, x :: acc)
```

proof for rev l :

want : rev l = $[x_n, \ldots x_1]$ when $[x_1, \ldots x_n] = l$

to show this, we need to show that helper $([x_1, \ldots x_n], [y_1, \ldots y_m]) = [x_n, \ldots x_1, y_1, \ldots y_m]$

base case: $n = 0$

helper $([], [y_1, \ldots y_m]) \rightarrow [y_1, \ldots y_m]$

inductive step: I.H: rev l works correctly for $|l| \leq n$. w.t.s rev l works for $|l| = n+1$

helper $([x_0, x_1, \ldots x_n], [y_1, \ldots y_m]) \rightarrow$ helper $([x_1, \ldots x_n], [x_0, y_1, \ldots y_m])$

by I.H., this $= [x_n, \ldots x_1, x_0, y_1, \ldots y_m]$ □

Russian peasant exponentiation

recursive method to compute b^e

```
rpe (b, e) =
  if (e = 0) then 1
  elif (b = 0) then 0   // catches special case
  elif (e is odd) then (b * rpe (b, e-1))
  else   // e is even
    let a = rpe (b, e/2) in a * a
```

proof for rpe : want to show rpe $(b, e) = b^e$ for all $e \geq 0$

base case: $e = 0$

rpe $(b, 0) = 1$

induction step: Assume rpe (b, e) works correctly for $e \leq n$

If e is odd, rpe $(b, e) = (b *$ rpe $(b, e-1))$

by I.H. rpe $(b, e-1) = b^{e-1}$

then $b \cdot b^{e-1} = b^e$

If e is even, rpe $(b, e) =$ (rpe $(b, \frac{e}{2}) \cdot$ rpe $(b, \frac{e}{2})$)

by I.H. rpe $(b, \frac{e}{2}) = b^{e/2}$

then $b^{e/2} \cdot b^{e/2} = b^e$ □

Another way to compute exponents:

```
fastexp(b,e) =
  let helper(b,e,a) =
    if (e=0) then a
    elif (e is odd) then helper(b,e-1, b*a)
    else helper(b*b, e/2, a)
  if b=0 then 0 else helper(b,e,1)
```

How can we reason about this program? We are no longer making recursive calls using strictly decreasing arguments, so we can't use the same technique we applied previously.

New idea: use an <u>invariant</u>. This is a quantity that remains constant after each step.

For fastexp, we will define our invariant to be $I = b^e \cdot a$

proof: No matter what branch we take in helper, I remains constant.

e is odd:

$$\text{helper}(b_0, e_0, a_0) \rightarrow \text{helper}(\underbrace{b_0}_{b_n}, \underbrace{e_0 - 1}_{e_n}, \underbrace{b_0 * a}_{a_n})$$

w.t.s. $b_0^{e_0} \cdot a_0 = b_n^{e_n} \cdot a_n$

$= b_0^{(e_0 - 1)} \cdot (b_0 \cdot a_0)$

$= b_0^{e_0} \cdot a_0$ ✓

e is even:

$$\text{helper}(b_0, e_0, a_0) \rightarrow \text{helper}(\underbrace{b_0 * b_0}_{b_n}, \underbrace{e_0/2}_{e_n}, \underbrace{a_0}_{a_n})$$

w.t.s. $b_0^{e_0} \cdot a_0 = b_n^{e_n} \cdot a_n$

$= (b_0 \cdot b_0)^{\frac{e_0}{2}} \cdot a_0$

$= b_0^{e_0} \cdot a_0$ ✓ □ I is an invariant

proof that fastexp is correct: w.t.s. $\text{fastexp}(b,e) = b^e$

if $b = 0$, $\text{fastexp}(0,e) = 0$ ✓

if $b > 0$, $\text{fastexp}(b,e) = \text{helper}(b,e,1)$

so in the initial call, $I = b^e$

by examining the code, we see helper ends when $e = 0$, outputting a_f. Does $a_f = b^e$ at this point?

We showed I remains constant, so

$b^e = b_f^0 \cdot a_f$ (where b_f denotes "b final")

$= 1 \cdot a_f = a_f$ ✓ □ $\text{fastexp}(b,e) = b^e$

↳ we proved that the value of I remains constant at each step, so the value of I at the beginning must equal the value of I when helper terminates.

initially $I = b^e \cdot 1 = b^e$ at the end $I = b_f^0 \cdot a_f = 1 \cdot a_f$

a_f is our output and $a_f = b^e$ ✓

2016.01.25
comp 302

inductive proofs of program behaviour

```
let reverse l =
        match l with
        | []->[]
        | x::xs -> append(reverse xs, [x])

let append( l1, l2) =
        match l1 with
        | [] -> l2
        | x::xs -> x:: (append(xs,l2))
```

proof for append:
base case: n=0 l1=[]
        append([], l2) -> l2
inductive step:
        i.h. append(l1,l2) -> l1l2 if |l1|s n
        if |l1|=n+1: append(l1,l2)
        (this mean x1:: (append([x1,…,xn], l2))
proof for reverse:
b.c: |l|=0
l=[] reverse []->[]
i.s: assume reverse l is correctly if |l| s n, prove for |l|=m+1
reverse l -> append(reverse xs, [x]) l=[x0,x1,…xn]
        append (reverse [x1, … ,xn], [x0]
        which means [xn,…x1] @[x0] =[xn….,x0]

```
let rec l = helper ( l, [])
let helper(l, acc)=
        match l with
        | []-> acc
        |x::xs -> helper(xs, x::acc)
```

inductive proof:
        base case: |l| =0 l=[] rev l = []
        rev l = helper ([], [])= acc = []
induvtive step: I.H. rev [x1,…xn] =[xn,…x1] works correctly
|l|=n+1 l=[x0,x1…,xn]
rev [x0,…xn] = helper([x0,….xn,[]] helper ([x0,…xn],[y1…,ym])
x=x0 w helper([x1,…xn], x0::[y1,…,ym])=d
xs=[x1,….xn]
helper([x1m…xn],[y1,…ym])=[xn,….x1,y1,…ym]

2016.01.25
comp 302

Russain peasent exponentiation
recursive method to compute b^e

```
rpe(b,e)=
        if (b=0) then 0
        elif (e=0) then 1
        elif (e is odd) then (b= rpe(b,e-1))
        else //e is even
                let a = rpe(l,e/2) in a*a
```

we want to show that rpe(b,e) = b^e for all $e \geq 0$

base case: e=0 $b^0=1$

rpe(b,0) = 1 works correctly

induction step: e=n+1 assume rpe(b,e)=b^e when $e \leq n$

if e is odd, rpe(b,e) = (b*rpe(b,e-1))

by induction hypothesis, rpe(b,e-1)=b^{e-1} $b*b^{e-1}=b^e$ works correctly

if e is even, rpe(b,e)=rpe(b,e/2) rpe(b,e/2)*rpe(b,e/2)=$b^{e/2}*b^{e/2}=b^e$

by induction hypothesis, rpe(b,e/2)=$b^{e/2}$

another way to compute exponents (b^e):

```
fastexp(b,e)=
        let helper(b,e,a)=
        if(e=0) then a
        elif (b is odd) then helper (b,e-1,b*a)
        else helper(b*b, e/2, a)
        if b=0 then 0 else helper(b,e,1)
```

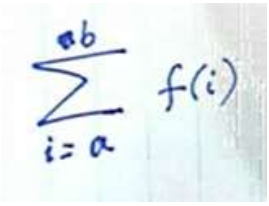
2016.01.27
topic : higher order functions (definition : takes a function as argument(dosnt affect the code))

Higher order functions

1. functions are first-class citizens. (they have all the rights and duties, they can be arguments to other functions; **cant** compare with other functions, **cant** print a function)
2. functions generated dynamically. (not always generated with 'let', the code creates a function not you, and perhaps returns it)
3. parametricity works at all levels.

code samples:

```
let rec sumInts(a,b) = if (a > b) then 0 else a + sumInts(a+1,b)

let rec sumSquares(a,b) = if (a > b) then 0 else (a*a) +
sumSquares(a+1,b)

let rec sumCubes(a,b) = if (a > b) then 0 else (a*a*a) +
sumCubes(a+1,b)

let rec sum(f,a,b) =
  if (a > b) then 0 else (f a) + sum(f,a+1,b)
//whatever it is, i will compute f of a and recursively do the rest of
it

let square n = n * n
let cube n = n * n * n

let rec sum_inc(f,a,b,inc) = //increment not just by 1
    if (a > b) then 0
    else (f a) + sum_inc(f, (inc a), b, inc)
//inc tells the value you are incrementing
//there are two function parameters, f and inc
//f is intentionally to be unspecified
```

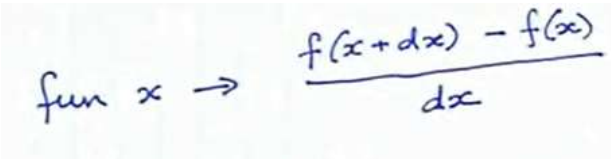



```
let byTwo n = n + 2

let rec product(f,a,b,inc) = //why 'sum'? i could've used 'product'
    if (a > b) then 1
    else (f a) * product(f, (inc a), b, inc)

let id x = x
let inc n = n + 1
product(id, 1, 5, inc) //this gives factorial 5 = 120

let acc(comb,f,a,b,inc,init) = //this is a higher order function
//this function is a general purpose accumulator
        //it has a 'f' telling how are you computing each value
        //it has an 'inc' telling how are you incrementing values
//'comb' how am i combining these terms
//'a and b' for limits
//'init' is the unit for operation,
//    for multiplication unit is 1, addition it is 0

  let rec helper(a, res) =     //this is what we had before
    if (a > b) then res
    else helper (inc(a), comb(res, f(a)))
  helper(a,init)
```

2016.01.27
topic : higher order functions (definition : takes a function as argument(dosnt affect the code))

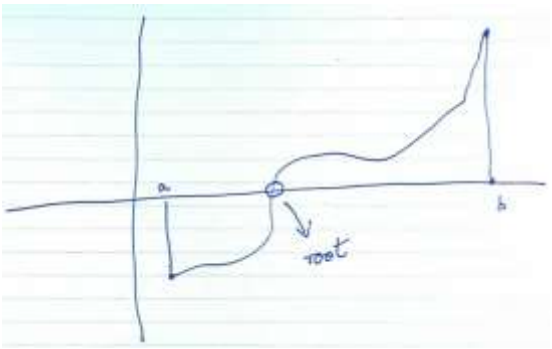
```
   -  - - -  - - - - generate dynamic function on the fly
now take a look at:
let test = acc((fun (x,y) -> x + y),(fun n -> n * n), 1,5,(fun m -> m +
1),0);;

fun n -> n+1 (this creates a function dynamically.)
proper way:
fun arg(can be structured) -> body (what happens to arg)
symbol: lamda
```

```
(* Doubler and self-application. *)
let twice f = fun x -> f (f x) //taking a function as an arg and apply
it twice

let inc n = n + 1

let fourtimes f = (twice twice) f //apply twice to itself

let compose(f,g) = fun x -> g(f(x)) //composition
//takes two functions as arguments and returns a new function
//the new function first apply f then apply g

(* Some examples from calculus. *)
let deriv (f, dx:float) = fun x -> ((f(x + dx) - f(x))/dx)
(x could be x^3)
//deriv takes a function as an argument, float to float
//it needs dx as well
//the output on the right hand side is
```

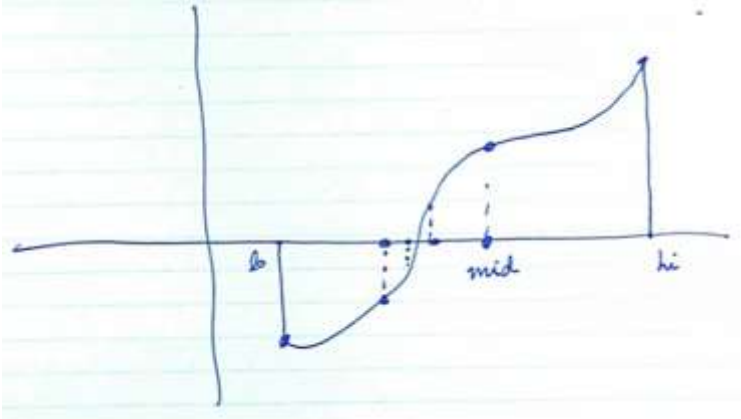



```
how do you know it is true?
for a programmer you know because you wrote the code
for others you know because you look at the type
derive:((float->float)*float)->(float->float) output

let abs(x:float) = if (x < 0.0) then ~x else x
let close(x:float,y:float,tol:float) = (abs(x-y) < tol)
let square(x:float) = x*x

every time you are shinking the interval you are converging the
segments very fast (this is calculus 1, differenciation)
let rec halfint(f,pos_value:float,neg_value:float,epsilon:float) =
  //the idea of this program is ONP
  let mid = (pos_value + neg_value)/2.0
  if (abs(f(mid)) < epsilon) //check if it is close
  then
    mid
  elif (f(mid) < 0.0)
  then
    halfint(f,pos_value,mid,epsilon)
  else
    halfint(f,mid,neg_value,epsilon);;
//every time you are shrinking the interval
```

2016.01.27
topic : higher order functions (definition : takes a function as argument(dosnt affect the code))



fundamental theorem of calc 2: in order to go from a to b, you must go through 0.



you are looking for root,but you don't know where it is, so you take the mid val between a and b, you know it is not 0 so you continue, this time you take mid between a and mid because you know mid is greater than 0, and you get a new mid value called m2, m2 is not quite close to 0, so you do the same thing except this time you take the mid val between m2 and b because m2 is negative, you continue to do this process until you get to root.
(now look back at the program halfint)

```
(this is calc 2, integration) (: is accumulating parameter)

let rec iter_sum(f, lo:float, hi:float, inc) =
  let rec helper(x:float, result:float) =
    if (x > hi) then result
    else helper(inc(x), f(x) + result)
  helper(lo,0.0);;

let integral(f,lo:float,hi:float,dx:float) =
  let delta (x:float) = x+dx //delta is your incrementor
  dx * iter_sum(f,(lo + (dx/2.0)), hi, delta)
//if you know how to compute integral, you can write out the formula
//like this and this is how easily you conquer calc 2 lmao

let r_sq (x:float):float = x * x

integral(r_sq,0.0,1.0,0.001)

integral(sin,0.0, 3.14159, 0.001)
```

2016.01.27
topic : higher order functions (definition : takes a function as argument(dosnt affect the code))

COMP302
topic: continue the discussion of higher order function

things to keep in mind when designing a program:

1. functions are first class citizens
2. functions generated dynamically
3. parametricity works at all levels.

when you are writing a piece of code, it just sits there and do nothing unless you run it and apply a function to it.

high order programming
(fun x -> body) is a value (fun x -> body) $\equiv \lambda x.body$
(fun x -> 1+2)
(fun x -> body)↓(fun x -> body)

evaluation rule for applications: ↓=evaluate

$$\frac{e1\downarrow(\text{fun } x \text{ -> body}) \qquad e2\downarrow v \quad [v/x]\ \text{body}\downarrow v'}{e1e2\downarrow v'}$$

//above the line is assumption
//e1e1=e1 applies e2 //below the line is what we conclude

this thing is called the **substitution model**

- e2 is evaluated first then passed to the function aka. **call by value**
  (the alternative is call by name)
- CBV is more efficient than call by name limit more possibilities of diverging

[v/x] body means substitutes v for x wherever x occurs in the body
[v/x] body ↓ v' evaluates the result and obtain v'

(fun x-> x +1) (3+2) ->
(fun x -> x+1)(5) -> 5+1 -> 6 **//reduction semantics**

eg:
if you write (fun x -> x+1) y **//what is y? it is an variable not a value**
let y = 99 in (fun x -> x+1) y means (fun x -> x + 1) 99 -> 99 +1 -> 100

church numerals:

```
let zero = fun f -> (fun x -> x)

let one = fun f -> (fun x -> (f x))

let two = fun f -> (fun x -> (f (f x)))

let showcn cn = (cn (fun n -> n + 1)) 0
```

I can try run these code,

```
let r1 = showcn one

let r2 = showcn two
```

```
//successor function, give me a cn i will output another cn function
// apply this f function returns a function x
// (cn f) apply f as many times as encoded in cn
// (f (( means one more f
let succ cn =  (fun f -> (fun x -> f ((cn f) x)))
```

COMP302
topic: continue the discussion of higher order function

```
let r3 = showcn (succ two)

let add n m = fun f -> (fun x -> ((n f) ((m f) x)))

let times n m = fun f -> (fun x -> (n (m f) x))
//(m f) is applied n times

let exp n m = fun f -> (fun x -> (m n) f x)
```

eg. smash **[[1;2;3],[4;5],[6]] to [1;2;3;4;5;6]**

```
let smash ll = List.fold (@) [] ll
//accumulates all the elements in ll (list of lists)
//using (@) append, as the accumulater
//and [] empty list as the starting point
//fold is the built-in higher order function
```

you have a list of lists, it gives you a list by using append and fold, starting from an empty list

eg. inter list 0, **[1;2;3] to [[0;1;2;3];[1;0;2;3];[1;2;-;3];[1;2;3;0]]**

// 0 is stucked everywhere in the list

//inter takes an item and a list, and produce a list of list of the item inserted in the list of lists

```
let rec inter item lst =
  match lst with
    | [] -> [[item]]
    | x :: xs -> (item :: lst) :: (List.map (fun u -> (x:: u)) (inter
item xs))
//if i have x in front of xs, then i put item in front of lst
//which gives my very first instance ([0;1;2;3])
//now i ripped out the x and just look at xs,
//i recursively inter for the rest of them
//but i need to put back element x into every one of them
//List.map is used, it says to each guy in the inter item xs
//stick x in front of each guy (this each guy increments every time)

eg. permutation [1;2;3] to
[[[1;2;3];[2;1;3];[2;3;1]];[[1;3;2];[3;1;2];[;3;2;1]]
//how many are them? n factorial many
//perms returns a list of lists
//the idea behind this program is
//i got [1;2;3] want to pull out 1, left with [2;3]
//i magically perms [2;3] to become [[2;3];[3;2]]
//now i have to inter 1 with every list in the lists
//[[[1;2;3];[2;1;3];[2;3;1]];[[1;3;2];[3;1;2];[;3;2;1]]
//but now i have too many levels of nesting so i need to smash it
let rec perms l =
  match l with
    | [] -> [[]]
    | x::xs -> smash (List.map (fun u -> (inter x u)) (perms xs))
```

COMP302
2016.02.01

```
let x = 1 in    (the 'in' word helps to not to indent  properly)
  let y = 2 in
    x + y;;     this is 3

let x = 1 in
  let x = 2 in
    x + x;;    this is 4
binding: x=1, x is bind to 1, inside a frame
frame: {} inside a environment
environment: includes everything
so you look at x+x, and you look for x and found out it is 2+2, so x=1
is shadowed.

let x = 1 in
  let x = 2 in
  printf "x is %i\n" x;;

let x = 1 in
  let y = x in
    let x = 2 in
      x + y;; this is 3 because x+y=(x=2)+(y=1)=3 (static)
this will output 4 in dynamic binding.
dynamic binding: the binding for the variable alters as the program
executes.
normally we use static binding, as the example above, but in object
oriented programming it is dynamic.
example:
let x = 1 in
        let y = x in
                let x=y in
                        x+y
dynamic binding: x=y and y=x will go into loops
static binding: x+y -> 2

let result =
  let x = 1
  let y = x
  let x = 2
  y

(* The first interesting example.  The function remembers the
definition of x!  *)
let result2 =
let x = 1 in
  let f = fun u -> (printf "Inside f, x is %i\n" x);(u + x) in
          let x = 2 in
                  (printf "x is %i\n" x);f x
(the printf is just for debugging and see what the output is)
so from the top of the function, x is bind to 1 inside, f is bind to a
function u which has an expression of u+x, the x is bind to 1 on the
top(this is called a closure: it remembers the environment that exists
where the function is defined), and x is bind to 2 at inside f.
f x is f 2 which means f has a parameter u and u is 2 in this call,
that being said, u+x -> u is 2 and x is 1 -> u+x=3
```

COMP302
2016.02.01

```
talk about recursion:
let rec fact n = if n = 0 then 1 else n*fact(n-1)
fact 1 which is 1
(the rec declaration makes the closure aware of environment being
created.)

(* Lightweight syntax. *)
let result3 =
  let x = 1
  let f = fun u -> (u + x)
  let x = 2
  f x

let result4 =
  let x = 1
  let y = let x = 3 in x
  (printf "x is %i\n" x); let x = 2  in (printf "Now x is %i\n" x);y;;
x is 1, now x is 2, y is 3

let x = 1 in
  let y = (let u = 3 in u + x) in
    let x = 2 in
    x + y;;

let x = 1 in
  let f = (let u = 3 in (fun y -> u + y + x)) in
    let x = 2 in
    f x;;
```

comp302
2016.02.03
these stuff are important, bindings and environment.
the meaning of expression should not change with later bindings.
(this dicipline is called STATIC BINDING OR LEXICAL SCOPING)
it is enforced by layers in the environment.

CLOSURES ensure that a function uses the envrionment that existed when it is creaed.

eg:

```
let x=1 in
        ley y = (let u=3 in u+x ) in
                let x = 2 in
                        x+y;
```

when a let is exited the binding is removed.
so we have x bond to 1, y bond to ?, u bond to 3, and you have an expression of u+x
therefore y bond to 4, then u is removed from the environment.
we have x bond to 2, and a expression x+y, which will return 2+4=6.

Trapping

```
let x = 1 in
        let f = (let u = 3 in
                fun y-> u+y+x))
        let x=2 in
                fx;;
```

we have
x=1
f=? there's a function inside, so we create a closure, y=u+y+x which points to u=3
u=3 that points to x=1 (temporary, will removed later on)
now you want to remove u=3 because we exited the scope but you cant because the frame is trapped inside the closure of f.
then we have x bond to 2, and f apply to x, (f x) in the end.
(f 2) points up to u=3(whatever the closure points to) so u + y + x = 3+2+1=6
u is private to f.

The environment model
Notes for COMP 302 Winter 2016

Prakash Panangaden
based on notes by Brigitte Pientka

February 22, 2016

Introduction

Until now we have understood function application through the substitution model. The intuitive idea is that when a function f with a single parameter, say x, is applied to an *expression* e the evaluation proceeds as follows. First, the expression e is evaluated to produce a value v. Then all *free* occurrences of x in the body of f are replaced by v. The resulting expression is then evaluated.

This intuitive view is what is captured by the operational semantics. Our evaluation rules are based on substitutions, which were formalized by an inductive definition. An advantage of the substitution model is that it is simple and easy to use in proving properties about programs. It provides a high-level abstract view of how programs are evaluated. Recall the rule for evaluating functions:

$$\begin{array}{lcl} \texttt{(fun } x \texttt{ -> } e\texttt{)} \ v & \longrightarrow & [v/x]e \\ \texttt{let } x = v \texttt{ in } e & \longrightarrow & [v/x]e \end{array}$$

Although this high-level view is convenient, because it abstracts over many implementation details and allows us to easily reason about programs and their behaviour, it has also some drawbacks. The main drawback is that *this is not what actually happens*! If one were to implement the language literally using the substitution model then one must copy the value of v multiple times, if x occurred multiple times in the expression e. It would be nicer and more efficient, if we could just remember this correspondence between x and the value v, and if we need it during evaluation of the expression e, we just look it up. The other drawback of the substitution model is that it does not easily extend to references and assignment. We will worry about that later.

We will introduce a different evaluation model, the *environment model*. It will provide a different view of evaluation where a particular structure called *the environment* keeps track of the correspondence between a name and a value. It provides a lower level view of the operational semantics, which is one step closer to an implementation, and gives a good explanation for references.

The `let` construct allows us to discuss the structure of the environment in detail. In F# one normally uses the lightweight syntax without the keyword `in`, but here I will not have room to show the whitespace necessary so I will use `in` in all my examples.

Terminology

1. A *binding* is an association between a name and a value. A name can name a value of any type such as an integer, a list a function or a memory location (a reference)[1]. For instance, in the expression `let x = 10 in x + 3`, we encounter the binding between the name `x` and the value `10`. Similarly, in the expression `let square = (fun x -> x * x)`, we have the binding between the name of the function `square`, and its input argument `x` and the function body `x * x`.
2. A *frame* is a collection of zero or more bindings, as well as a pointer to another frame, which is called its enclosing environment. An environment is a structured collection of frames, starting from a particular frame and going back through each frame's enclosing environment until the global environment is reached.

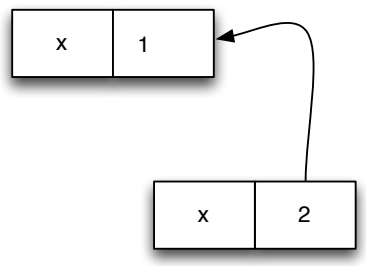
In the environment model, an expression is always evaluated in the context of a particular environment. The environment determines what values correspond to the names occurring in the expression. The purpose of an environment is to provide a way to associate a value with a particular name. The way this works is that the first frame in the environment is searched to see if it contains a binding for that name. If so, the associated value is used. If not, the first frame of the enclosing environment is searched, and so on up to the global environment. If the name is not found there, an error is reported. The evaluator **never** follows a pointer backwards!

There are three different possible bindings. We may bind a name to an integer, float, etc, or we may bind a name to a function, or we may bind it to a location in memory. A function is a complicated entity, it is not just the body of the function. It is a more complex entity called a *closure*. We will discuss closures in some detail below.

Let us look at some examples. A binding is typically represented by a box with two parts. The left part has the name of the binding while the right part either contains the value if the value is an integer, boolean, etc. For more complex entities like functions we will introduce some more notational conventions later.

To look up a binding for `x`, we follow the pointers (or arrows) until we find the first binding for it. Consider the first example.

[1] I will avoid the word "variable" because it suggests that things "vary"; sometimes they do, as with ref types, and at other times they do not.



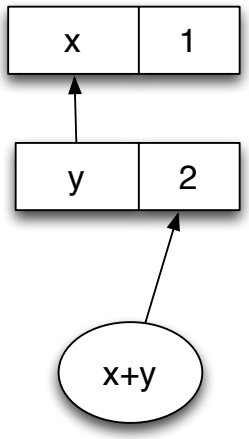
let x = 1
let x = 2

Here first `x` is declared with `let x =1` and then *inside the scope of this binding* we have another declaration for `x`. The structure is shown in the picture above. It is important to realize that we are **not changing the value of `x`** with the second declaration, we are creating a **new `x`** with a new binding in **its own frame**. *Both bindings exist at the same time.*

Now consider

```
let x = 1 in
  let y = 2 in
    x + y;;
```

The environment and the expression being evaluated is shown below.



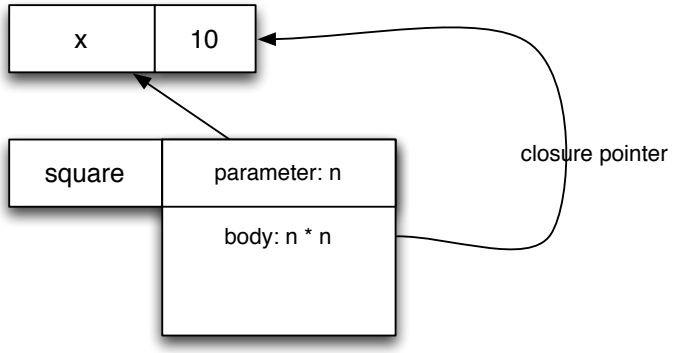
The arrows show the pointers from a frame to the enclosing frame. There is no problem with this example. We wish to evaluate the expression `x + y`; we need bindings for `x` and `y`. We follow the arrows looking for the binding in each frame. Incidentally, we also need a definition for the plus symbol: this is in the top-level frame which is loaded when F# starts up, it contains all the pre-defined names. We will look at more subtle cases later. Let us turn to function evaluation for now.

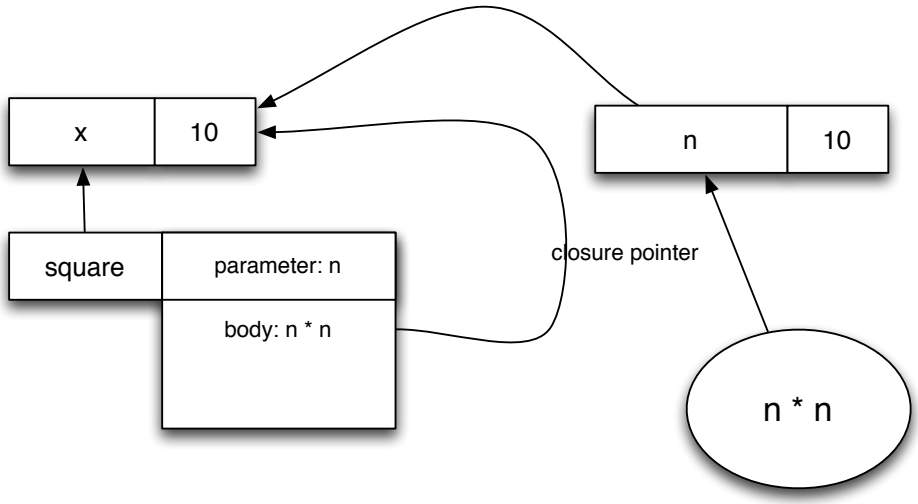
Next, we show what happens when evaluating `square x` in the environment where we have de-

fined

```
let x = 10
let square n = n * n
square x
```

This gives the environment shown in the first picture below. What happens if we try to evaluate `square x`? This expression is inside the inner `let` so it sees both the frames above. The first thing the evaluator looks for is the name "square"; it finds it and sees that it is indeed a function. It then looks for `x`, which is does not find in this frame, so it follows the pointer up and finds another frame where it does find the definition of `x`. Now it looks for the parameter name in the function definition and sees that it is `n`; **it creates a new frame** which is shown in the picture below, and binds `n` to the result it got from evaluating `x`. **This frame is temporary; it will be removed when evaluation is over**. Now the body is evaluated in the environment shown in the lower part of the picture.



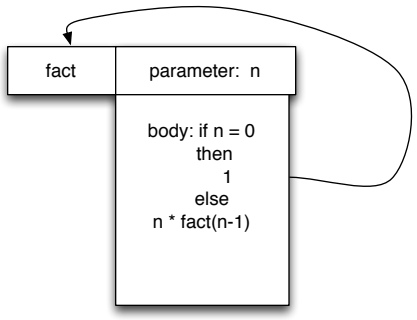


Why did the arrow from the (temporary) frame for `n` point to the frame for `x` rather than the frame for `square`? The new frame points to **the same place as the pointer from the function**. You see now that if inside the body of `square` there were another reference to `square` there would be an error. In short, functions defined this way cannot be called recursively. One has to do something special for recursive functions.

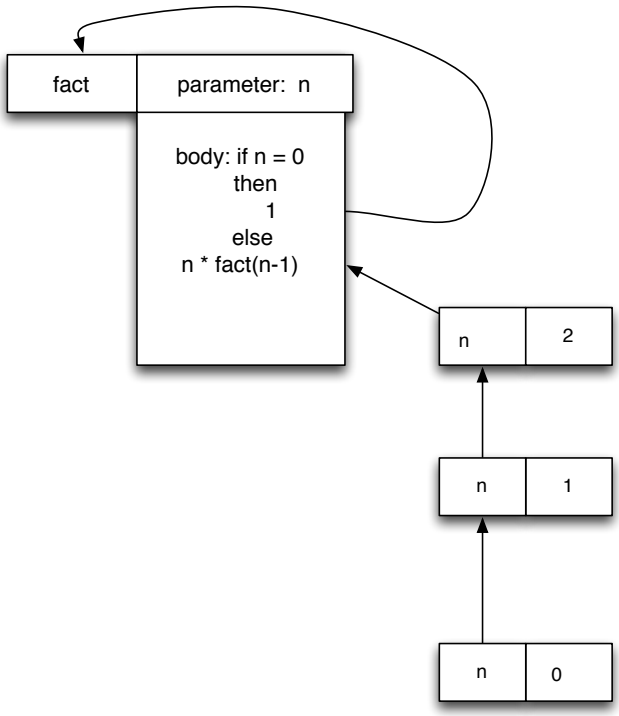
To illustrate what happens when we have recursion, consider evaluating the factorial function `fact`. Here is the code:

```
let rec fact n = if n = 0 then 1 else n * fact(n-1)
```

Notice we have sais `let rec`; this is a signal to send the arrow back to frame being created. The picture below shows the result.



Now consider what happens when we compute `fact 2`. The resulting environment is shown below:



We need to bind the input argument `n` of `fact` to `2`. This is the top most binding in this frame. When executing the body of `fact`, namely `if 2 = 0 then 1 else 2 * fact(1)`, we call factorial recursively. Now the input argument is bound to `1`. Therefore we establish another binding for `n` which will point to the previous binding where `n` was `2`. So in every recursion step, we will keep track of the binding between the input argument and the current value it is bound to, until we have reached the final value. It is worth mentioning that in our discussion of the environment model we are only talking about keeping track of bindings. The computation still to be done in each recursion is typically tracked by a run-time stack which we do not model in these notes.

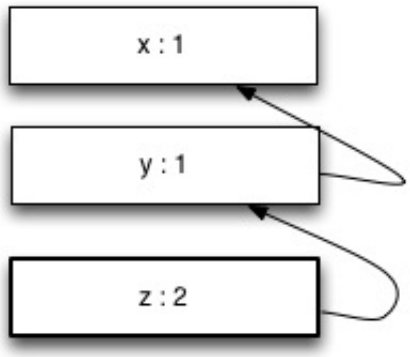
Local scopes

We can create local scopes by using `let`. The scope is delimited by the `let` keyword. Furthermore, when the expression evaluation is complete the *let bindings are removed*. Thus they are *local* and *temporary*. Every language has this feature, with whatever notation[2]. In Java, C and $C++$ one uses nested curly braces for exactly this purpose.

Consider the following:

```
let  x = 1 in
  let  y = x in
    let  z = 2 in
         y + z
```

Each nested `let` creates a new layer as shown in the figure.



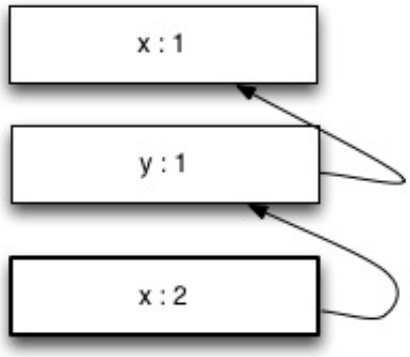
One *crucial* point: we do **not** store the binding as `y:x`. When the binding for `y` is established, the system evaluates `x` and finds the value 1. Thus, if later frames define new bindings for `x` the binding for `y` does **not** change. This discipline is called *static binding*.

Let me illustrate this point with a more subtle example.

[2]This is **not** a class about F#!

```
let  x = 1 in
  let  y = x in
    let  x = 2 in
         x + y
```

Does this give $2, 3$ or 4? The environment looks like this at the end

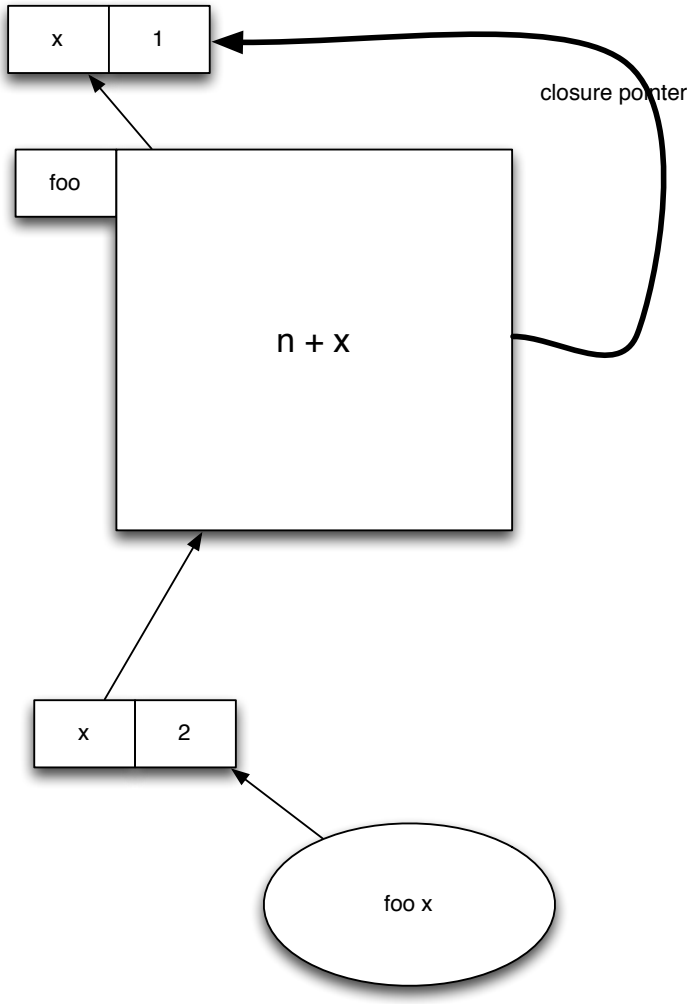


When we evaluate `x + y`, the `x` is looked up and the latest value 2 is seen but the `y` is bound to the same value as it was when it was set up: the bindings do not change, that is why it is called static binding.

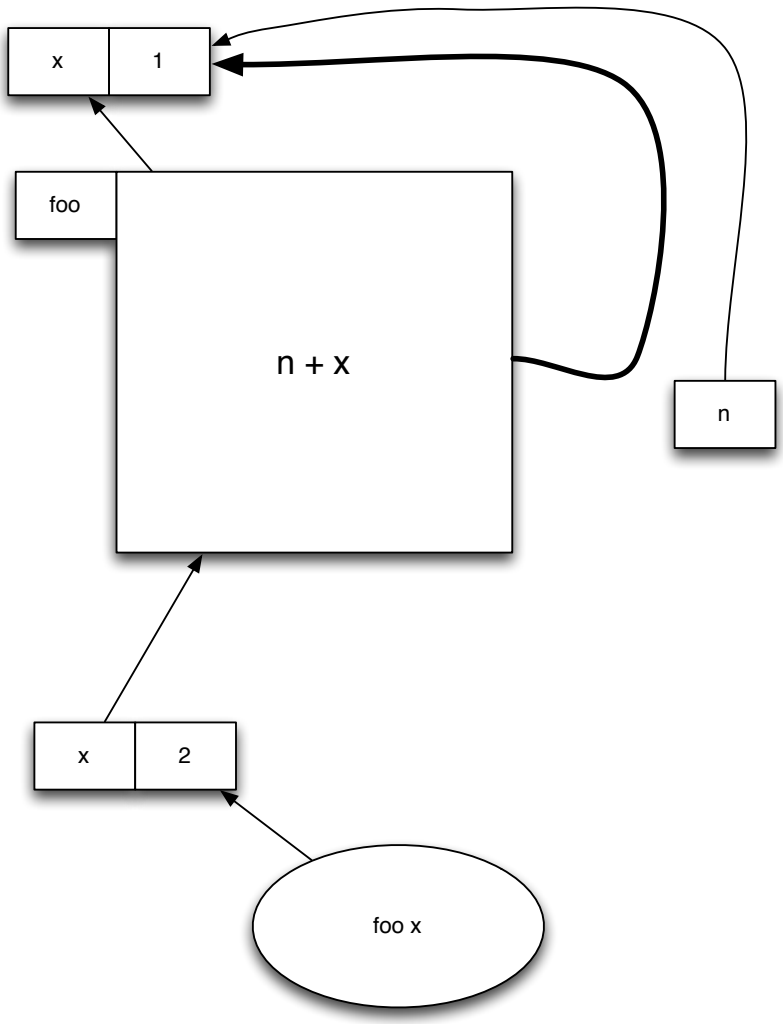
The same phenomenon can be illustrated with combinations of `let` and function definitions. Consider

```
let  x = 1 in
  let foo n = n + x in
    let  x = 2 in
         foo x
```

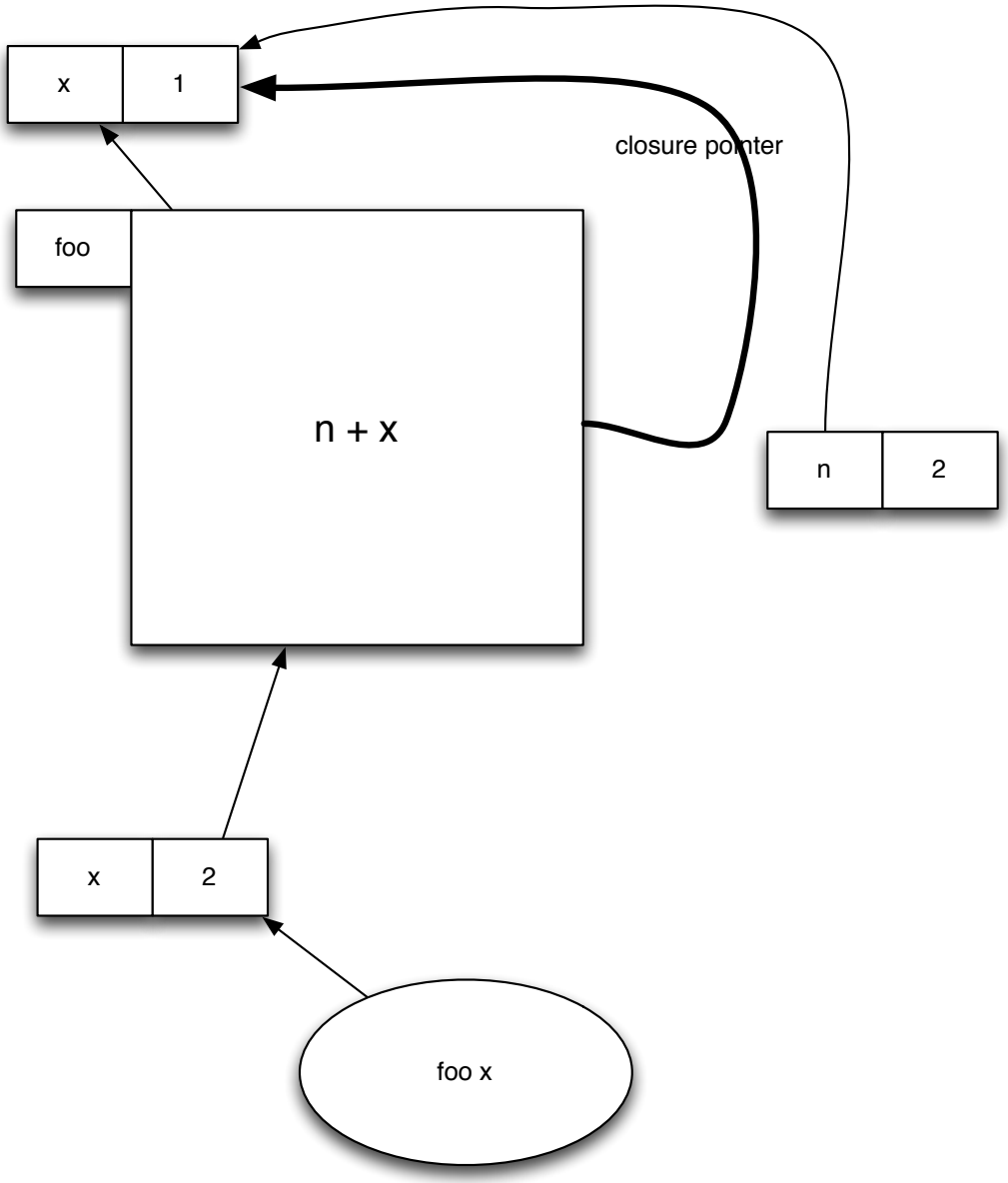
This gives the value 3 for exactly the same reason. When the function `foo` is defined it contains a pointed to the environment that exists *at the time that it is created.* The only binding for `x` is $(x, 1)$ at that point. Even though when `foo` is called there is a new binding for `x`, the environment inside the closure for `foo` will not have this new binding. This picture shows what the environment looks like just after the three bindings are set up and the evaluation of `foo x` is about to start.



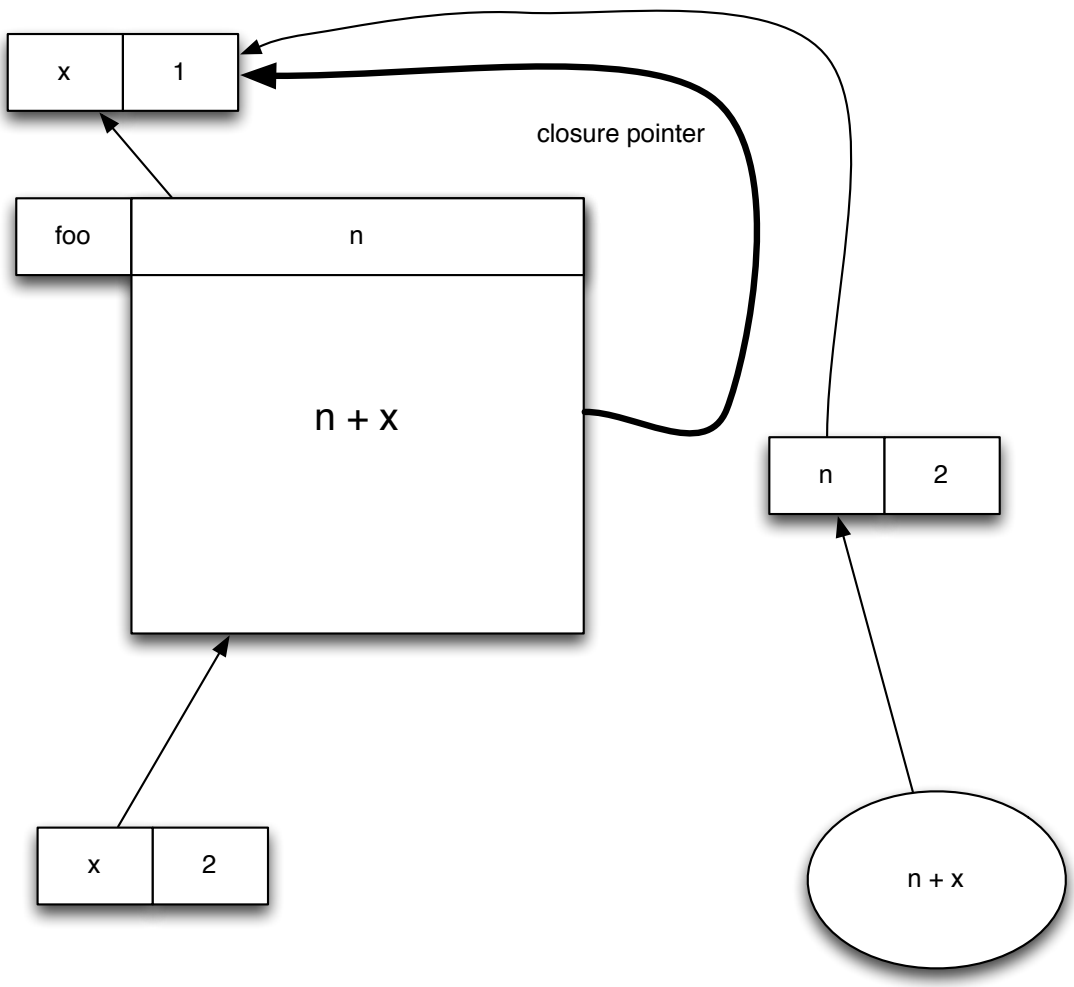
When `foo x` is evaluated, the name `foo` is looked up first and it is found to be a function with one parameter. Accordingly, a frame is set up for the parameter; **the pointer from this frame goes to wherever the pointer from the definition of foo goes.** The partially created environment now looks like this:



Functions are called by value so before the body can be executed *the argument is evaluated first*, thus `x` is looked up and the value 2 is returned. Then the 2 is bound to n and a new frame is created (this frame will be removed when the function evaluation is complete) and *placed on top of a copy of the environment in the closure as we indicated above*. The picture now looks like this:



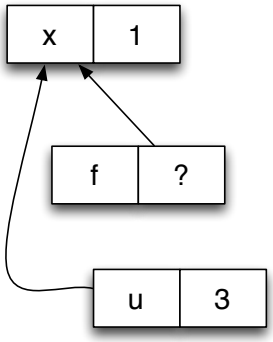
Now in evaluating the body `n + x`, two names have to be looked up. The `n` is found to be bound to 2 and the `x` is found to be bound to 1; the binding $(x, 2)$ will not be seen. The final picture looks like this:



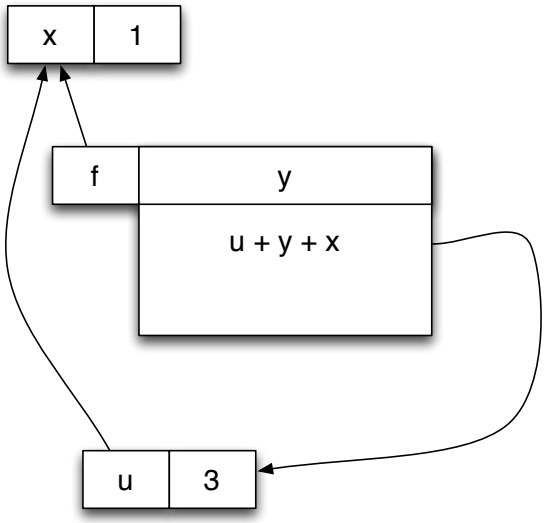
A final example combines the ideas we have seen so far. It is subtle so look at it carefully.

```
let x = 1 in
  let f =
    (let u = 3 in (fun y -> u + y + x) ) in
  let x = 2 in
     f(x)
```

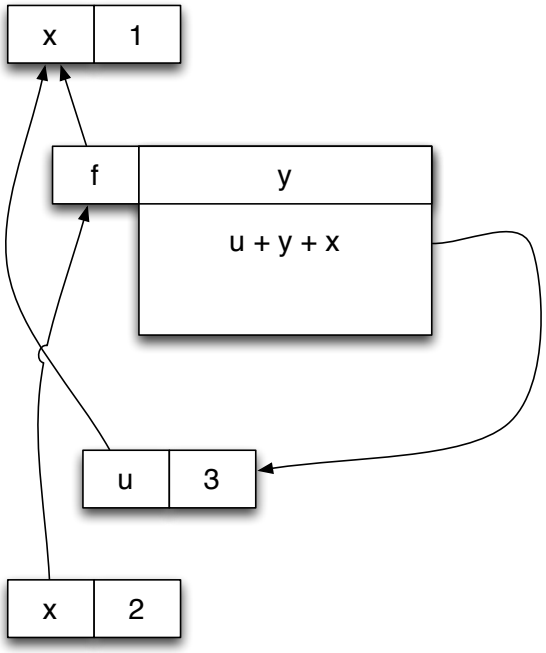
In the first picture below we see that the frame for the binding $(x, 1)$ is set up. The definition of f is not yet complete. We have to evaluate the expression on the right-hand-side of `let f = ...`, which starts with a `let u = ...`. Thus the frame for the binding $(u, 3)$ is set up. This points to the frame that exists at this time, namely to the top frame; the binding for f is not yet set up at this stage.



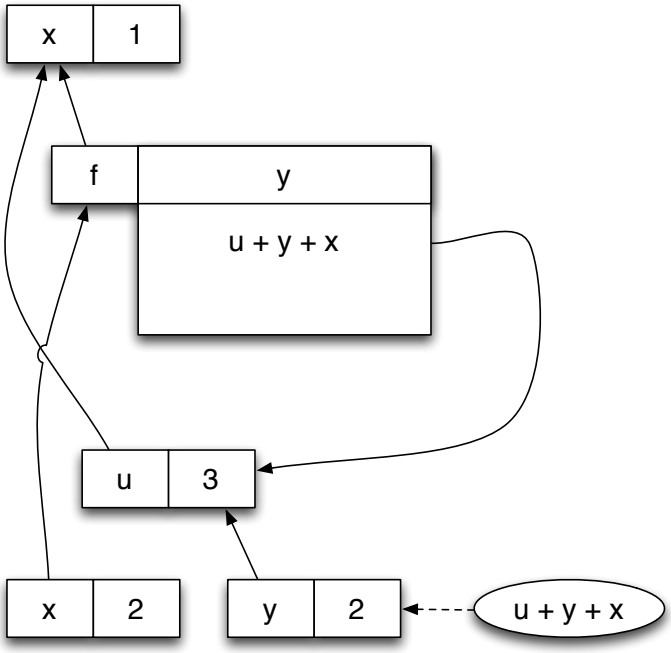
Then the closure for f can be constructed. It is shown below. Note carefully that frame $(u, 3)$ is alive and well and is the latest frame so the closure of f contains a pointer to this frame. *This traps the frame*; as long as f is alive, the u frame is also alive.



Lastly the frame for the inner x binding is set up; the `let` binding for u is closed (but it is still alive and trapped inside the closure for f) so the pointer from this frame goes to the frame for f.



Now we make the function call $f(x)$. The evaluation of x gives 2 so a new temporary frame binding $(y, 2)$ is set up. Where does it point? It points to whatever the closure of f points to f. The resulting diagram is shown below.



The final result of this computation is 6.

Summary

The environment model can be summarized as follows:

An environment is a structured collection of *frames*. Each frame is a box (possibly empty) of bindings, which associate names with their corresponding values. (A single frame may contain at most one binding for any variable.) Each frame also has a pointer to its enclosing environment. The value associated with a name with respect to an environment is the value given by the binding of the name in the first frame in the environment that contains a binding for that name. If no frame in the collection specifies a binding for the name, then the name is said to be unbound in the environment.

These notes were written originally for SML, then edited for OCaml and edited again for F#. This just shows that the concepts are all the same; there are only minor differences in syntax. These concepts are **not** valid for Python as far as I know[3].

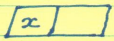
[3]Which is not very far.

Rules for drawing environment diagrams:

1. Every "let" opens a new scope for bindings: so when you see

   let x = ...

   you should immediately draw

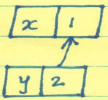


   with the box on the right empty at first. We will fill it in presently.

2. There should be an arrow from the new box to the NEAREST ENCLOSING box that is <u>still open</u>. So, if we have

   let x = 1 in
   let y = 2 in

   We draw



   The arrow goes from one <u>whole box</u> to another; in this picture the arrow is pointing at [x|1], <u>NOT</u> at [1]!!

3. To fill the value in the right-hand-side of a box you must produce a value: (a) an integer, (b) a boolean (c) a float (d) a string (e) a data structure OR (f) a function (together with some additional data). (g) a memory address [LATER]

(4) We need to evaluate the expression exp in

let x = exp

in order to get a value. This expression is to be evaluated in the environment that exists WHEN you enter the let. It cannot include the value you are defining. Thus you cannot implement recursion with a let.

(5) To implement a recursive definition you have to use the special keyword rec

let rec fact n = - - - - -

This causes the new environment pointer to point to the frame being created.

(6) The evaluation of a function definition produces a CLOSURE. Thus

let foo n = n+1729

produces a closure, as does

let foo = fun n → n+1729

(7) What is a closure?: A closure has 3 pieces:

(a) a (list of) parameter(s)

(b) a body, i.e. code which may mention names

(c) a pointer to the environment that exists when the function is defined. You never follow this pointer when searching for bindings.

How a function is called:

(i) First, evaluate the argument(s) in the current environment.

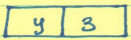
(ii) Create a new frame (binding) matching the parameter with the value produced by evaluating the argument.

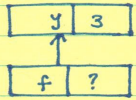
(iii) The pointer from this frame goes to the same place as the pointer in the closure.

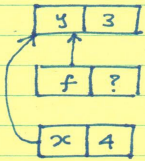
(iv) Now evaluate the body. In doing so you follow pointers until you find the name you need. You NEVER go inside closures and follow pointers there. The closure is used to set up the environment for evaluating the function body.

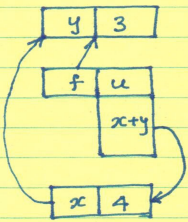
———— x ————

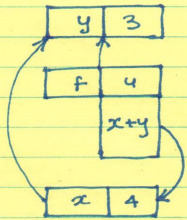
```
let y = 3 in
  let f = let x=4 in fun u -> x+y in
  f(1)
```

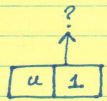






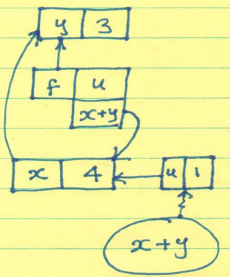






WHERE DOES THIS GO?

Ans: To the same place as the arrows in the closure



⇝ 7

2016.02.05
COMP302

COMMAND
expressions—evaluation→ values
commands ~> ():unit
(they are evaluated for side-effect.)
eg. printf

update
x=1729
x<-1729

mutable variables
let mutable x=1
what x means now is not an integer but a memory cell that stores an integer.
if x was pointed to 1 ie. x=1 and now let x<-2
then x now is pointing at 2 in the same cell now.

new type of value: location
env
name~> value
x~>a box that stores 1
x~environment part~>location~store part~>1

x<-**x**+1
this name is used in 2 different ways
*name of a location
**the value uside the location

referential transparency:
William Rufus was so called because of his red hair.
he is known as william the conquerer
William the conquere was so called because of his red hair.
(this sentence now doesn't make sense)
(missed the last part of what he was saying, might need to go back to see the recording)

let mutable x=1

~~let change (mutable n:int) =~~
~~n<-n+1;; (this is forbidden)~~

let munge(n:int)=
let mutable m=n
m<-m+1
this function is retarded, no effect persists after execution.

let mes_with() =x<-x+1
updates a global variable

2016.02.05
COMP302

write a while loop: this is a function that takes an int parameter n, print all the integer from 1 to 10

```
let foo n=
        let mutable x =n in
                while (x<10) do
                (printf fn "x is %i" x);(x<-x+1)
```

(*semi colan ; is the command combiner.)

$(exp1);(exp2);\ldots;(exp_n)$

evalutated is from L-> R

each value is discarded except the last.

if we want side-effects to persist we need records.

```
(* Basic record syntax . *)

type Person = { name : string ; birthday : int * int; title : string }

let prakash = { name = "Prakash"; birthday = (3,11); title = "Lord of
all he surveys"}

type intRec = { mutable count : int }
```

let r1={count =0}

```
let increment (counter: intRec) =
  counter.count <- counter.count + 1
  counter.count;;
```

this is a legitimate F# program.

F# provides some macros:

```
(* Using ref, :=, ! *  Run these and explain what happens. *)
let x = ref 1 (automatically creates a record with content:1)
let y = x
let incr n = (n := !n + 1) (take n and updated like n+1)
(!n (bang n) is n.content, ! is dereferencing operator, because n is a
record, you cant do n+1 on mutable record without dereferencing)'
incr x (increment int x by 1)
!x
!y
let z = ref (!y)  (int ref = {contents=2;}
incr z
!z
!y
```

using ref always creates a record with 1 mutable field called contents.

result if we run the code:

```
let x = ref 1   (x -> a record that has content of 1)
let y = x       (y -> whatever x is pointing)
let incr n = (n := !n + 1)
incr x
!x = 2
!y = 2
let z = ref (!y) (z is  now pointing to whatever y is pointing, 2)
incr z
!z = 3
!y = 2 (it did not get affected)
```

(**the bolded parts are two different things, very important)**

COMP302
MIDTERM ON 25TH

45 min midterm on 25th, 7-8pm, MC204
1 cheat sheet double sided
contents:
- higher order funcntion
- list and recursion (understand the tricky list programs)
- ENV Diagram

tricky list programs:

```
let psums lst =   (*partial sum*)
  let rec helper l a =
    match l with
      | [] -> [a]
      | x ::xs -> a :: (helper xs (x + a))
  match lst with
    | [] -> [0]
    | x :: xs -> helper xs x
```

```
let smash ll = List.fold (@) [] ll (*it takes a list of lists, grabs each of the sublists and appends it to a new list so that the output is simply all the elements within all the sublists. so [[1,2,3],[4,5,6]] -> [1,2,3,4,5,6]*)
```

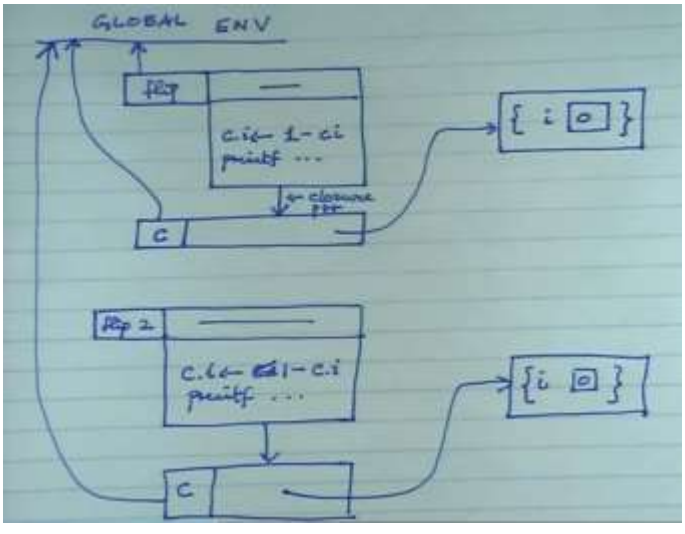
```
let rec inter item lst =
  match lst with
    | [] -> [[item]]
    | x :: xs -> (item :: lst) :: (List.map (fun u -> (x:: u)) (inter item xs))

let rec perms l =
  match l with
    | [] -> [[]]
    | x::xs -> smash (List.map (fun u -> (inter x u)) (perms xs))
```

today's topic: mutation and closures
examples:

```
type counter = { mutable i: int};;
let mutable a = 0;;
let flip0 () = (a <- (1 - a)); (printf "%i\n" a);;
```

(if a is zero it is 1, if a is 1 it is zero.)

```
let flip =
  let c:counter = { i = 0 }
  fun () -> (c.i <- 1 - c.i); (printf "%i\n" c.i);;

let flip2 =
  let c:counter = { i = 0 }
  fun () -> (c.i <- 1 - c.i); (printf "%i\n" c.i);;
```



COMP302

MIDTERM ON 25TH

```
C is an object, it's able to be manipulated independently and
privately. It's trapped inside the closure.

let make_flipper = fun () ->
  let c:counter = { i = 0 }
  fun () -> (c.i <- 1 - c.i); (printf "%i\n" c.i)
(* Datatype of bank account transactions.*)
type transaction = Withdraw of int | Deposit of int | Checkbalance

(* Bank account generator. *)
let make_account(opening_balance: int) = (return a function)
(happen) let balance = ref opening_balance
    fun (t: transaction) ->
      match t with
        | Withdraw(m) ->  if (!balance > m)
                          then
                            balance := !balance - m
                            printfn "Balance is %i" !balance
                          else
                            printfn "Insufficient funds."
        | Deposit(m) -> (balance := !balance + m; (printf "Balance is
%i\n" !balance))
        | Checkbalance -> (printf "Balance is %i\n" !balance);;
```

mutation & closures

wrapper: pieces of code wrapped around some other code.

typical example: monitoring(keep an eye on someone and make sure you know what they are doing)

```
let morgoth = make_account(1000)
let sauron = make_account(500)

(* Monitoring *)
type 'a tagged = Query | Normal of 'a
type 'b answers = Numcalls of int | Ans of 'b

let makeMonitored f =
    let c = ref 0
    fun x ->
      match x with
        | Query -> (Numcalls !c)
        | (Normal y) -> ( c := !c + 1; (Ans (f y)));;

(* To avoid the value restriction we have to use a non-polymorphic
version. *)
let monLength = makeMonitored (fun (l : int list) -> (List.length l));;

let inc n = n + 1

let moninc = makeMonitored(inc)

let taglist l = List.map (fun x -> Normal(x)) l
```

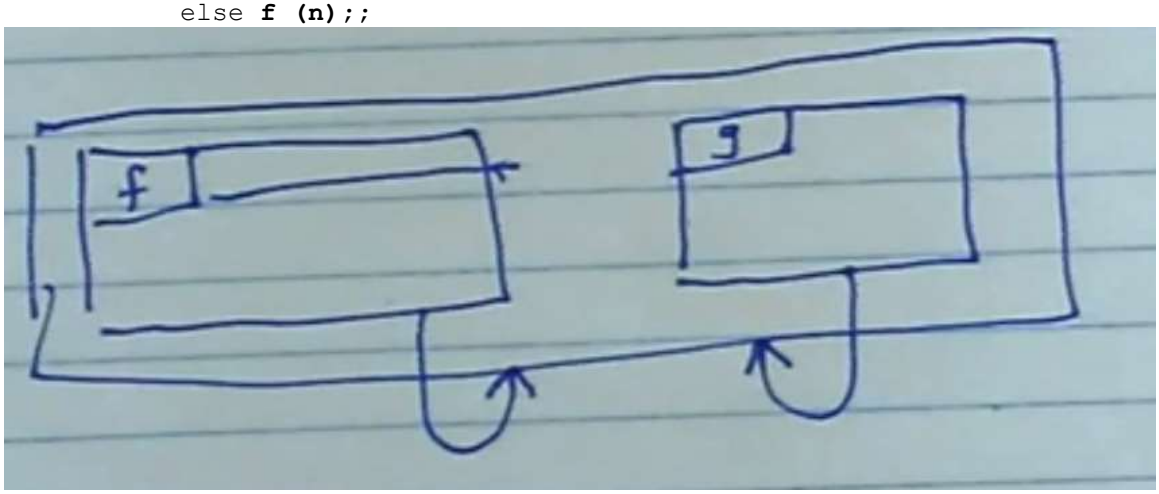
COMP302
MIDTERM ON 25TH

Mutual recursion:

```
let rec f =
  let c = ref 0 (local frame)
  fun n -> if (n%2 = 1) then (c := !c + 1);(printfn "Inside f c is:
%i" !c)
           else g(n)
```

this function does: if it's a odd number then increments it and print it

```
and g =
  let c = ref 0(local frame
  fun n -> if (n%2 = 0) then (c := !c - 1); (printfn "Entered g c is:
%i" !c)
           else f (n);;
```




```
they both point at the env frame
(*
val f : (int -> unit)
val g : (int -> unit)

> > f 4;;
Entered g c is: -1
val it : unit = ()
> g 3;;
Inside f c is: 1
val it : unit = ()
> g 2;;
Entered g c is: -2
val it : unit = ()
> f 3;;
Inside f c is: 2
val it : unit = ()

*)
```

two copies that are each trapped each is private but f and g can see each other.

Look up seb.fold for assignment 2
set.iter may be useful but may be misleading
set.forall is usefull
set.add c1(set.add c2 s)

last lecture on imperative programming

Stack:
local buildings
-let
-function calls

data structures: records, lists, objects,etc… are allocated on the heap.
they persists until they become unavalible.

```
let l1=[1;2;3]
let l2=l1@[4;5]
```

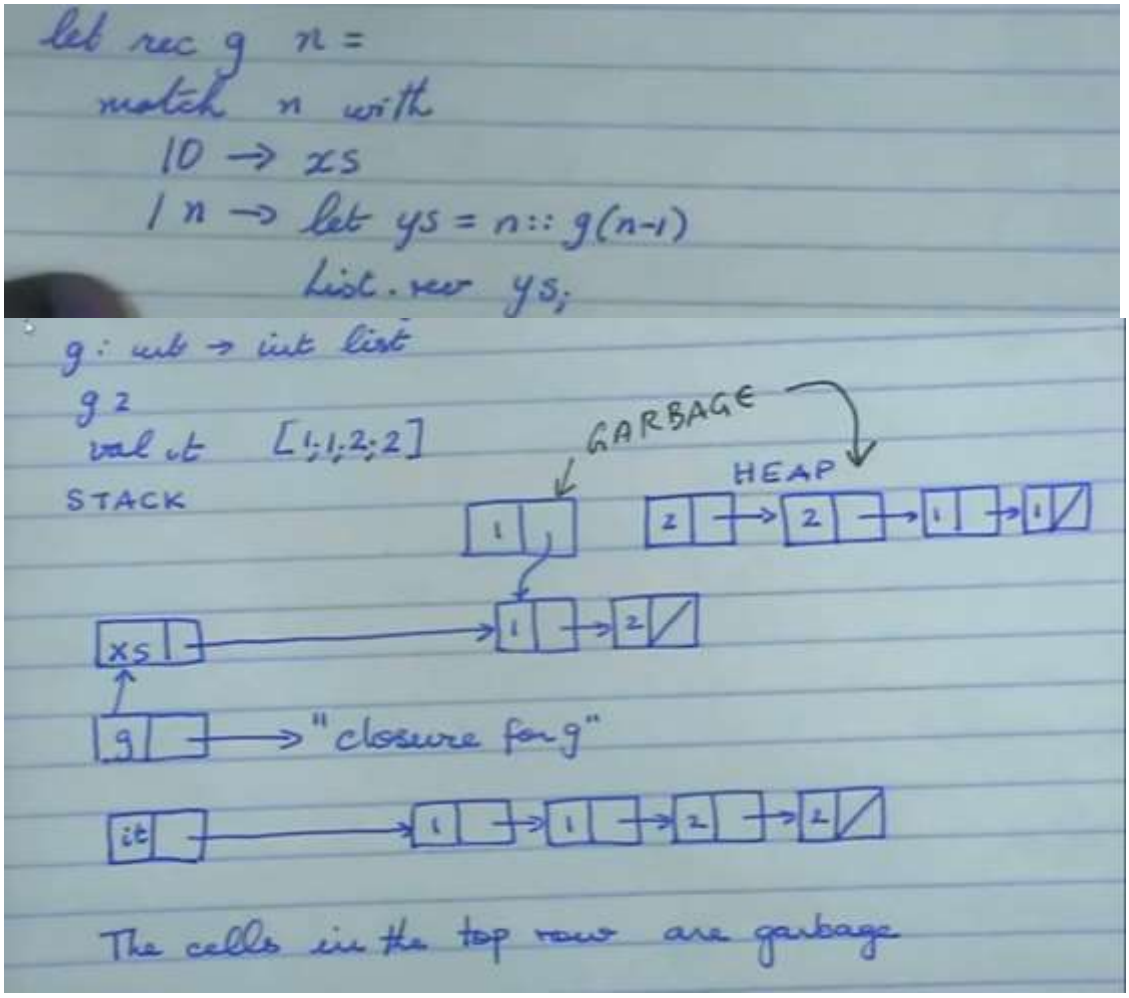
stack heap
l1 -> 1 -> 2 -> 3 '\0'
l2 -> 1->2->3->4->5
l1 is copied but only if necessary.
STRUCTURES ARE IMMUTABLE.
once you created l1, it ends at 3.

l3=0::l2 (let l3 be zero and stick before l2)
l3 -> 0 -> l2
which means
l3 -> 0->1->2->3->4->5

```
let l4= l1@l3;;
```
l4-> 1->2->3->l3

```
let zs= let xs=[1;2]
        let ys=[3;4]
        xs@ys;;
```
stack heap
zs
xs points to zs
xs has 1->2
ys points to ys
ys has 3->4
there will be a temp cell stores 1->2
so zs has 1->2->3->4
and 1->2 which was stored at temp, now these are inaccessibble (GARBAGE)

last lecture on imperative programming



lifetime of data
stack: automatically allocated and deallocated
heap: allocated explicitly; dea;;ocated by the garbage collectior
programmer does not dispose of stack data -> schewe, SML, CAML, haskshell, clogure, scala, java, c#, ruby
prgrammer given explicit control: c, c++, python, pascal
malloc; free
garbage collector avalibable freely
heap problem in real time systems
verification of code to ensure closure of memory leaks.

```
(* This is how you turn on timing

> #time;;

--> Timing now on

*)

let lst = [1;2]
```

last lecture on imperative programming

```
let rec wastetime n =
  match n with
    | 0 -> lst
    | _ -> let ys = n :: wastetime(n-1)
           List.rev ys;;

let rec fib n =
  match n with
    | 0 -> 1
    | 1 -> 1
    | _ -> fib(n-1) + fib(n-2);;

let tailfib n =
  let rec helper (n,a,b) =
    match n with
      | 0 -> a
      | 1 -> b
      | _ -> helper(n-1,b,a+b)
  helper(n,1,1);;

let rec naive_reverse l =
  match l with
    | [] -> []
    | x :: xs -> naive_reverse(xs) @ [x];;

let tfact n =
  let rec helper(n,m) =
    match n with
      | 0 -> m
      | _ -> helper(n-1,n*m)
  helper(n,1);;

let factW n =
  let ni = ref n
  let r = ref 1
  while (!ni > 0) do
    (r := !r * !ni); (ni := !ni - 1)
  !r;;

(* Timing examples

> wastetime 10000;;
Real: 00:00:01.131, CPU: 00:00:01.131, GC gen0: 187, gen1: 5
val it : int list =
  [9999; 9997; 9995; 9993; 9991; 9989; 9987; 9985; 9983; 9981; 9979;
9977;
   9975; 9973; 9971; 9969; 9967; 9965; 9963; 9961; 9959; 9957; 9955;
9953;
   9951; 9949; 9947; 9945; 9943; 9941; 9939; 9937; 9935; 9933; 9931;
9929;
   9927; 9925; 9923; 9921; 9919; 9917; 9915; 9913; 9911; 9909; 9907;
9905;
   9903; 9901; 9899; 9897; 9895; 9893; 9891; 9889; 9887; 9885; 9883;
9881;
   9879; 9877; 9875; 9873; 9871; 9869; 9867; 9865; 9863; 9861; 9859;
9857;
```

```
   9855; 9853; 9851; 9849; 9847; 9845; 9843; 9841; 9839; 9837; 9835;
9833;
   9831; 9829; 9827; 9825; 9823; 9821; 9819; 9817; 9815; 9813; 9811;
9809;
   9807; 9805; 9803; 9801; ...]
> Real: 00:00:00.000, CPU: 00:00:00.000, GC gen0: 0, gen1: 0

Pretty fast!

> > fib 20;;
Real: 00:00:00.000, CPU: 00:00:00.000, GC gen0: 0, gen1: 0
val it : int = 10946
> fib 30;;
Real: 00:00:00.008, CPU: 00:00:00.008, GC gen0: 0, gen1: 0
val it : int = 1346269
> fib 40;;
Real: 00:00:00.996, CPU: 00:00:00.996, GC gen0: 0, gen1: 0
val it : int = 165580141
> fib 50;;
  ^C ^C
- Interrupt I couldn't wait!
> fib 45;;
Real: 00:00:10.997, CPU: 00:00:10.997, GC gen0: 0, gen1: 0
val it : int = 1836311903

> > tailfib 40;;
Real: 00:00:00.000, CPU: 00:00:00.000, GC gen0: 0, gen1: 0
val it : int = 165580141
> tailfib 45;;
Real: 00:00:00.000, CPU: 00:00:00.000, GC gen0: 0, gen1: 0
val it : int = 1836311903
> tailfib 50;;
Real: 00:00:00.000, CPU: 00:00:00.000, GC gen0: 0, gen1: 0
val it : int = -1109825406

Fast but cannot handle big numbers with ordinary integers.

> > let result = naive_reverse [1 .. 10000];;
Real: 00:00:01.483, CPU: 00:00:01.483, GC gen0: 190, gen1: 6

val result : int list =
  [10000; 9999; 9998; 9997; 9996; 9995; 9994; 9993; 9992; 9991; 9990;
9989;
   9988; 9987; 9986; 9985; 9984; 9983; 9982; 9981; 9980; 9979; 9978;
9977;
   9976; 9975; 9974; 9973; 9972; 9971; 9970; 9969; 9968; 9967; 9966;
9965;
   9964; 9963; 9962; 9961; 9960; 9959; 9958; 9957; 9956; 9955; 9954;
9953;
   9952; 9951; 9950; 9949; 9948; 9947; 9946; 9945; 9944; 9943; 9942;
9941;
   9940; 9939; 9938; 9937; 9936; 9935; 9934; 9933; 9932; 9931; 9930;
9929;
   9928; 9927; 9926; 9925; 9924; 9923; 9922; 9921; 9920; 9919; 9918;
9917;
   9916; 9915; 9914; 9913; 9912; 9911; 9910; 9909; 9908; 9907; 9906;
9905;
```

```
   9904; 9903; 9902; 9901; ...]

> let r2d2 = List.rev [1 .. 10000];;
Real: 00:00:00.000, CPU: 00:00:00.000, GC gen0: 0, gen1: 0

val r2d2 : int list =
  [10000; 9999; 9998; 9997; 9996; 9995; 9994; 9993; 9992; 9991; 9990;
9989;
   9988; 9987; 9986; 9985; 9984; 9983; 9982; 9981; 9980; 9979; 9978;
9977;
   9976; 9975; 9974; 9973; 9972; 9971; 9970; 9969; 9968; 9967; 9966;
9965;
   9964; 9963; 9962; 9961; 9960; 9959; 9958; 9957; 9956; 9955; 9954;
9953;
   9952; 9951; 9950; 9949; 9948; 9947; 9946; 9945; 9944; 9943; 9942;
9941;
   9940; 9939; 9938; 9937; 9936; 9935; 9934; 9933; 9932; 9931; 9930;
9929;
   9928; 9927; 9926; 9925; 9924; 9923; 9922; 9921; 9920; 9919; 9918;
9917;
   9916; 9915; 9914; 9913; 9912; 9911; 9910; 9909; 9908; 9907; 9906;
9905;
   9904; 9903; 9902; 9901; ...]

Built in list reversal is very fast.

I computed 16! a million times and threw away the answer.  I used a
tailrecursive
version and a familiar imperative version with a while loop.  The tail
recursive
version was actually faster.

> > for i in 1 .. 1000000 do let _ = tfact(16) in ();;
Real: 00:00:00.022, CPU: 00:00:00.022, GC gen0: 0, gen1: 0
val it : unit = ()
> for i in 1 .. 1000000 do let _ = factW(16) in ();;
Real: 00:00:00.054, CPU: 00:00:00.054, GC gen0: 8, gen1: 0
val it : unit = ()

*)
```

The language of Types

In modern programming languages the types are rich enough to require a little language of their own. Here we describe such a type language for an ML-like language.

Basic Types: int / bool / string / char / real

Now we can build compound types; in order to do this we use type constructors: these are constructs to create new types from old ones:

* : product → : function space

list | : sum

I am ignoring array, record and other things for now.

Let us consider * : this is the counterpart of the term constructor (,).

Given two types say int & bool we define a new type int * bool. How do we connect terms of the language & types? Through typing rules. I will use letters like f, e, t etc for terms & greek letters α, β for types.

A typing rule has the form

$$\frac{term_1 : typ_1, \quad \cdots \quad term_k : typ_k}{term : typ}$$

We use type variables to have generic rules Here is the rule for * :

$$\frac{t_1 : \tau_1, \quad t_2 : \tau_2}{(t_1, t_2) : \tau_1 * \tau_2}$$

How do we read such a rule? In words: *if* you have *shown* that t_1 has type τ_1 & that t_2 has type τ_2 *then* you can conclude that the pair (t_1, t_2) has type (τ_1, τ_2).

How do we start off? We need some basic assumptions for the basic types. Here, for example, are the rules for *int*:

$$\frac{}{0 : int} \qquad \frac{}{1 : int} \qquad \frac{}{2 : int} \qquad \frac{}{\sim 1 : int} \quad \ldots$$

Why draw a line with nothing on top? To indicate that we need no further assumptions:

0 has type int, no assumptions are needed to conclude this. We can now type more complicated expressions with more rules:

$$\frac{x : int,\ y : int}{x + y : int} \qquad \frac{x : int,\ y : int}{x * y : int} \quad \ldots$$

This is not the complete set of rules for int, but it gives the idea. Here are some rules for bool

$$\frac{}{true : bool} \qquad \frac{}{false : bool} \qquad \frac{b : bool}{not\ b : bool} \quad \ldots$$

$$\frac{x : int,\ y : int}{x = y : bool}$$

Here is the rule for conditionals

$$\frac{e_1 : \tau,\ e_2 : \tau,\ b : bool}{\text{if } b \text{ then } e_1 \text{ else } e_2 : \tau}$$

This says that both branches must have the same type τ, the condition must be a boolean and then the overall conditional will have type τ.

Now some rules for tuples and projections as I showed before.

$$\frac{t_1 : \tau_1 \quad , \quad t_2 : \tau_2}{(t_1, t_2) : \tau_1 * \tau_2}$$

We can use this with any types, e.g.

$$\frac{17 : \text{int}, \quad \text{false} : \text{bool}}{(17, \text{false}) : \text{int} * \text{bool}}$$

We can make nested pairs too:

$$\frac{(17, \text{false}) : \text{int} * \text{bool} \qquad 2+3 : \text{int}}{((17, \text{false}), 2+3) : (\text{int} * \text{bool}) * \text{int}}$$

Note we are assigning types to expressions not just to values. We introduce <u>destructors</u> fst and snd:

$$\frac{e : \tau_1 * \tau_2}{\text{fst}(e) : \tau_1} \qquad \frac{e : \tau_1 * \tau_2}{\text{snd}(e) : \tau_2}$$

Now we can type a complete expression

$$\frac{\dfrac{17 : \text{int} \quad \text{true} : \text{bool}}{(17, \text{true}) : \text{int} * \text{bool}} \qquad \dfrac{2 : \text{int} \quad , \quad 3 : \text{int}}{(2+3) : \text{int}}}{((17, \text{true}), (2+3)) : (\text{int} * \text{bool}) * \text{int}}$$

$$\text{snd}((17, \text{true}), (2+3)) : \text{int}$$

This entire picture represents a little <u>proof</u>. A proof is just a tree structure with axioms at the leaves & a rule at every node. Conceptually, the type checker builds this tree to verify the typing of an expression. Actually type checking is done more eagerly but you should think of type checking as the construction of such a proof tree.

Lists : list is a type constructor, :: is a term constructor, nil or [] is a constant and hd, tl are the destructors. Here are the rules

$$\frac{e : \tau \qquad l : \tau\text{-list}}{e :: l : \tau\text{-list}}$$

$$\frac{l : \tau\text{-list}}{hd(l) : \tau} \qquad \frac{l : \tau\text{-list}}{tl(l) : \tau\text{-list}}$$

Note the type system will not tell you what happens if you apply hd to nil. The type rule says this is OK but in reality an exception is raised.

Now for the most important type constructor: $\rightarrow$ However we need to deal with variables first because a parameter is a crucial part of fn. How does a variable get a type? For now, we assume that variables are typed by declarations. Later we will see that polymorphic types can be inferred. We will need to keep track of these declarations when making typing judgments. We have a set of assumptions (or declarations) called a context. For example we might have

$$n : int,\ x : bool,\ y : int * int, \ldots$$

We will use Γ for a context. Our judgement will look like $\Gamma \vdash e : \tau$

Using the type assumptions (or declarations) in Γ we conclude that the expression e has the type τ. We can add these contexts to our previous rules & use them otherwise unchanged.

$$\frac{\Gamma \vdash e : \text{bool} \qquad \Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 : \tau}$$

Before we deal with functions let us consider let expression which introduce new bindings. We need to manipulate Γ

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}$$

This says the obvious thing: if $x : \tau$ is one of the assumptions in Γ then one can conclude $x : \tau$.

Now for let

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 \text{ end} : \tau_2} \quad x \text{ must be fresh in } \Gamma$$

Here is an example:

$$\dfrac{\dfrac{x : \text{int} \vdash x : \text{int} \qquad x : \text{int} \vdash 2 : \text{int}}{x : \text{int} \vdash x + 2 : \text{int}} \qquad \dfrac{\overbrace{x : \text{int}, y : \text{int}}^{\Gamma} \vdash x : \text{int} \qquad \Gamma \vdash y : \text{int}}{x : \text{int}, y : \text{int} \vdash x + y : \text{int}}}{\dfrac{\vdash 5 : \text{int} \qquad x : \text{int} \vdash \text{let } y = x+2 \text{ in } x+y \text{ end} : \text{int}}{\vdash \text{let } x = 5 \text{ in } (\text{let } y = x+2 \text{ in } x+y \text{ end}) \text{ end} : \text{int}}}$$

Note order of assumptions does not matter, we can rearrange the order at will. We can reuse assumptions as often as we want. We may have unused assumptions. For example, in let $x = 5$ in 3 end : int we do not need any assumption about x to type check the body $3 : \text{int}$.

Now the all important → constructor. If τ_1, τ_2 are types then $\tau_1 \to \tau_2$ is a type. We have a term constructor fn $x \Rightarrow \cdots$ and a "destructor" i.e. function application. Here are the rules:

$$\frac{\Gamma, x:\tau_1 \vdash e:\tau_2}{\Gamma \vdash \text{fn } x \Rightarrow e : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}$$

Let us derive some types

$$\frac{x:\text{int} \vdash x:\text{int}}{\vdash \text{fn } x \Rightarrow x : \text{int} \to \text{int}} \qquad \frac{x:\text{bool} \vdash x:\text{bool}}{\vdash \text{fn } x \Rightarrow x : \text{bool} \to \text{bool}}$$

Hmm! The same expression has multiple types. Later we will discuss polymorphism. For now we will think monomorphically.

Some more examples

```
let x = 1 in
    let f = fn u => u+x in
        let y = 2 in
            f y
        end
    end
end
```

This will lead to a large tree that will not fit on the paper. I will show it on the next sheet sideways.

Γ_1

$x:int, f:int\to int, y:int \vdash f:int\to int \quad \Gamma_1 \vdash y:int$

$x:int, u:int \vdash x:int \quad x:int, x:int, u:int \vdash u:int$ $\quad \Gamma_1 \vdash 2:int$ $\qquad \Gamma_1 \vdash f\ y : int$

$x:int, u:int \vdash u+x:int$ $\qquad x:int,\ f:int\to int \vdash \text{let } y=2 \text{ in } f\ y \text{ end} : int$

$x:int \vdash \text{fn } u \Rightarrow u+x : int \to int$

$x : int \vdash \text{let } f = \text{fn } u \Rightarrow u+x \text{ in } : int$

$\text{let } y=2 \text{ in}$

$f\ y$

end

$\vdash \text{let } x=1 \text{ in}$

$\text{let } f = \text{fn } u \Rightarrow u+x \text{ in}$

$\text{let } y=2 \text{ in}$

$f\ y : int$

Types for references

We will pretend that the F# shortcuts are part of the core language

expressions ::= ... | $e_1 := e_2$ | $!e$ | ref e | () | ...

types ::= ... | τ ref | unit | ...

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \text{ref } e : \tau \text{ ref}} \qquad \frac{}{\Gamma \vdash () : \text{unit}}$$

$$\frac{\Gamma \vdash e : \tau \text{ ref}}{\Gamma \vdash !e : \tau} \qquad \frac{\Gamma \vdash e_1 : \tau \text{ ref} \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 := e_2 : \text{unit}}$$

let increment $(x : \text{int ref}) : \text{unit} = x := !x + 1$

$\Gamma = x : \text{int ref}$

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash 1 : \text{int}} \quad \dfrac{\Gamma \vdash x : \text{int ref}}{\Gamma \vdash !x : \text{int}}}{\Gamma \vdash !x + 1 : \text{int}} \quad \dfrac{\Gamma \vdash x : \text{int ref}}{\Gamma \vdash x := !x + 1 : \text{unit}}}{\vdash \text{fun } (x : \text{int ref}) \to x := !x + 1 : \text{int ref} \to \text{unit}}$$

type are there to inhibit behaviour.

simple type system:
basic types,
type constructors: *, -> (builds function type), list
polymorphic type system: means many shaped
same expression can be view as multiple types
'a list -> type variable
typing rules has a form
e1:t1……en:tn } assumptions

---

e:t }conclude

e1:t1 e2:t2

(E1,e2):t1*t2

1729:int "foo":string

(1729,"foo"):int*string

x:int y:int

---

x=y:bool

e1:t1 e2:t1 b:bool

---

if b then e1 else e2:t1 (t1 can stand for any type but it must be used consistently)

17: int false:bool	2:int 3:int
(17, false): int*bool	2+3:int
e1:int	e2:int

e1+e2:int
this is a type derivation

Polymorphism The major new ingredient is type variables within the type system. We use letters like $\alpha, \beta, \gamma \ldots$ as type variables. Our language of types is now

$$\tau ::= int \mid bool \ldots \mid \tau_1 * \tau_2 \mid \tau_1 \to \tau_2 \mid \alpha$$

What does it mean to say an expression e has type $\alpha * \alpha$? It means that *any* type expression *substituted consistently* for α gives a possible type of e. Thus it means that e belongs to any one of a *family* of *types*.

What are these substitutions? We define by induction the notation $[\tau/\alpha]\tau'$: replace occurrences of α in τ' by τ.

(1) $[\tau/\alpha]\alpha = \tau$

(2) $[\tau/\alpha]\beta = \beta \quad (\alpha \neq \beta)$

(3) $[\tau/\alpha]int = int$

(4) $[\tau/\alpha]bool = bool$

(5) $[\tau/\alpha]\,\tau_1 * \tau_2 = ([\tau/\alpha]\tau_1) * ([\tau/\alpha]\tau_2)$

(6) $[\tau/\alpha]\,\tau_1 \to \tau_2 = ([\tau/\alpha]\tau_1) \to ([\tau/\alpha]\tau_2)$.

Here is the crucial rule that captures polymorphism

$$\text{If } \Gamma \vdash e : \tau \text{ then } [\tau'/\alpha]\Gamma \vdash e : [\tau'/\alpha]\tau.$$

In our definition of substitution there is nothing that says τ cannot have its own type variables. Consider `fun x -> x`

It can be given many monotypes

$$\frac{x : int \vdash x : int}{\vdash \text{fun } x \to x : int \to int}$$

or

$$\frac{x : int \to int \vdash x : int \to int}{\vdash \text{fun } x \to x : (int \to int) \to (int \to int)}$$

These types for fun $x \to x$ are all obtained by making appropriate substitutions to $\alpha \to \alpha$.

We call them substitution instances of $\alpha \to \alpha$.

Thm For every expression e, there is a unique type τ, possibly containing type variables such that every valid type for e is obtained by an appropriate substitution of τ. We say that τ is a (or the) principal type for e.

TYPE INFERENCE

The strategy is to introduce fresh type variables whenever we don't know the type of a sub-expression. Then we look at how the expressions are used to infer constraints on the type variables. In the final phase, we try to solve the constraints. We will look for the most general solution: this means that we are looking for a solution such that all other possible solutions are substitution instances of the most general one.

NOTATION $\Gamma \vdash e : \tau / C$

in the context Γ, the expression e will have type τ if the constraints in C are satisfied. The constraints are of the form $\tau_1 = \tau_2$.

For constants we do not generate constraints.

$$\frac{}{\Gamma \vdash n : int \,/\, \emptyset} \qquad \frac{x:\tau \in \Gamma}{\Gamma \vdash x:\tau \,/\, \emptyset}$$

$$\frac{\Gamma \vdash e : bool \,/\, C_0 \qquad \Gamma \vdash e_1 : \tau_1 \,/\, C_1 \qquad \Gamma \vdash e_2 : \tau_2 \,/\, C_2}{\Gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 : \tau \,/\, C_0 \cup C_1 \cup C_2 \cup \{\tau_1 = \tau_2\}}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \,/\, C_1 \qquad \Gamma \vdash e_2 : \tau_2 \,/\, C_2}{\Gamma \vdash e_1 + e_2 : int \,/\, C_1 \cup C_2 \cup \{\tau_1 = int,\ \tau_2 = int\}}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \,/\, C_1 \qquad \Gamma \vdash e_2 : \tau_2 \,/\, C_2}{\Gamma \vdash e_1 = e_2 : bool \,/\, C_1 \cup C_2 \cup \{\tau_1 = \tau_2\}}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \,/\, C_1 \qquad \Gamma, x:\tau_1 \vdash e_2 : \tau_2 \,/\, C_2}{\Gamma \vdash let\ x = e_1\ in\ e_2 : \tau_2 \,/\, C_1 \cup C_2}$$

Functions We don't have declarations so how can we know the type of x in fun $x \to \ldots$? We don't know, so we introduce a fresh type variable say α:

$$\frac{\Gamma, x:\alpha \vdash e : \tau \,/\, C}{\Gamma \vdash fun\ x \to e : \alpha \to \tau \,/\, C}$$

Example

$$\frac{\dfrac{\dfrac{}{x:\alpha \vdash x:\alpha \,/\, \emptyset} \qquad \dfrac{}{x:\alpha \vdash 1 : int \,/\, \emptyset}}{x:\alpha \vdash x+1 : int \,/\, \{\alpha = int\}}}{\vdash fun\ x \to x+1 : \alpha \to int \,/\, \{\alpha = int\}}$$

solution $\alpha = int$ so we get

$$\frac{}{\vdash fun\ x \to x+1 : int \to int}$$

Applications: We have to guess the return type of $e_1\ e_2$ by introducing a fresh type variable:

$$\frac{\Gamma \vdash e_1 : \tau_1 / C_1 \qquad \Gamma \vdash e_2 : \tau_2 / C_2}{\Gamma \vdash e_1\ e_2 : \alpha \,/\, C_1 \cup C_2 \cup \{\tau_1 = \tau_2 \to \alpha\}}$$

Lists

$$\frac{\Gamma \vdash e_1 : \tau_1 / C_1 \qquad \Gamma \vdash e_2 : \tau_2 / C_2}{\Gamma \vdash (e_1 :: e_2) : \tau_2 / C_1 \cup C_2 \cup \{\tau_2 = \tau_1\text{-list}\}}$$

$$\frac{}{\vdash [\,] : \alpha\text{-list}} \qquad \frac{\Gamma \vdash l : \tau\text{-list} / C}{\Gamma \vdash head(l) : \tau / C} \qquad \frac{\Gamma \vdash l : \tau\text{-list} / C}{\Gamma \vdash tail(l) : \tau\text{-list} / C}$$

EXAMPLES OF INFORMAL DERIVATIONS

let rec map = fun f → fun x → if (x = [ ]) then [ ]
else f(head(x)) :: (map f (tail(x))).

We introduce variables for the types that we do not see know and then look for constraints:

$f : \alpha$, $x : \beta$

From $x = [\,]$ we see $\beta = \gamma\text{-list}$

From $f(head(x))$ we see f is a function and $head(x) : \gamma$ so $\alpha = \gamma \to \delta$ δ is fresh.

$f(head(x)) = \delta$ so $f(head(x)) :: \cdots : \delta\text{-list}$.

So type of map is

$$(\gamma \to \delta) \to \gamma\text{-list} \to \delta\text{-list}.$$

```
let rec append (l1, l2) =
    match l1 with
    | [] -> l2
    | x::xs -> x:: (append (xs, l2)).
```

$l_1 : \alpha \quad l_2 : \beta$

from the match : $\alpha = \gamma$-list γ-fresh

$x : \gamma$ so return type is γ-list

so if $l_2 : \gamma$-list

Thus γ-list $*$ γ-list $\rightarrow \gamma$-list.

```
let double = fun f -> fun x -> f (f x)
```

$f : \alpha, \quad x : \beta \qquad f\ x : \gamma$

so $f : \alpha \rightarrow \beta \rightarrow \gamma = \alpha$

$f\ (f\ x)$ says input type for f is γ

so $\beta = \gamma$

double : $(\beta \rightarrow \beta) \rightarrow \beta \rightarrow \beta$

~~let~~ fun $x \rightarrow$ fun $y \rightarrow (x, y)$

$\alpha \rightarrow \beta \rightarrow \alpha * \beta$

fun $f \rightarrow f\ f$

$f : \alpha \qquad f\ f$ says $f : \alpha \rightarrow \beta$

so $\boxed{\alpha = \alpha \rightarrow \beta}$

This equation cannot be solved!

comp302
2016.02.17

3 kinds of polymorphism
- parametric
- selectype polymorphism
- ad-hoc polymorphism or overloading

+ int->int->int
+ float->float->float
the machine code is totally different

parametric polymorphism: the same code works uniformly at many types.

list manipulation programs don't care what is in the list.

type inference - refers to the automatic deduction of the data **type** of an expression in a programming language.

we will have type variables; a,b, c,…. type vars
l: a- list, l:'a list
any substitution of a type for a yields a valid type for l.

[t / a] t' : replace any occurrence of a in t' with t

[int / a] a-list = int – list

[b-list / a] a-list = b-list-list

[int->int / a] a->b = (int->int)->b (replace a with b)
1) [t/a] a = t
2) [t/a] b = b (a=/=b)
3) [t/a] int = int
4) [t/a] bool = bool
5) [t/a] t1*t2 = ([t/a]t1)*([t/a]t2)
6) [t/a]t1->t2 = ([t/a]t1)->([t/a]t2)

fun x -> x
x:int |- x:int

|- fun x->x: int -> int

x:float list |- f: float list

|- fun x-> x: float list -> float list

this is where type variable come in,
x has a unknwon type alpha |- x: alpha

|- fun x->x: alpha->alpha
FACT: any possible valid type for fun x->x is obtained by some substitution instance of x->x

comp302
2016.02.17

what is the time for this output? started from bottom and go upwards
x: int |-x:int x:int|-x:int
---
x: |- x+x (we know this line exists)

|- fun x-> x+x: int->int

read the notes online, page 1-4 (polymorphic type notes)

lets look at some informal examples:
```
let rec map = fun f -> fun x->
    if (x=[]) then p[
    else f(head(x))::(map f (tail x))
```
INFORMAL
f:alpha x:beta
beta=gamma list (some list you don't know) (looking at if(x=[])
alpha = gamma -> delta (looking at f(head(x), head(x):gamma),
(delta=fresh(some new type we created)
f(head(x)): delta
  return type is delta list
map: (gamma->delta) -> gamma list -> delta list
in f sharp it will say ('a->'b) ->'a list ->'b list

```
let double = fun f->fun x -> f(f x)
```
f:alpha x:beta
when we see f x, we know the output is the same type of the input
alpha= some -> type, some is beta, it returns type=gamma(we don't know what it is)
f:b->g x:b
fx:gamma
f(f x) -> constraint
so we deduced beta = gamma
f:b->b
double: (b->b) ->b->b

fun f -> f **f** (this is rejected as badlytype
f:a = b->g
a=b
a=a->g ?

a=(a->g)->g
=(a->g)->g->g
….
never settles down to give a valid type expression

Solving typing constraints

$\{ \alpha = int \to \beta,\ \beta = \beta_1 * bool,\ \beta_1 = int \}$ can be solved by

$\alpha = int \to (int * bool)$

$\beta = int * bool \qquad \beta_1 = int$

$\{ \alpha_1 \to \alpha_2 = int \to \beta,\ \beta = int \to \alpha_1 \}$

$\alpha_1 = int \qquad \beta = int \to int = \alpha_2$

So we can use something like Gaussian elimination. The algorithm is called <u>unification</u>

We write σ for a substitution $[\tau/\alpha]$ where τ is a type (perhaps containing type variables) and α is a type variable. We write $[\sigma]\tau$ for the effect of carrying out σ. If τ_1 & τ_2 are type expressions & σ is a substitution on <u>all</u> the type variables - so σ could look like $[\tau_1/\alpha_1, \tau_2/\alpha_2, \dots]$ - such that

$[\sigma]\tau_1 = [\sigma]\tau_2$, where the equality sign now means identity, we say that τ_1 & τ_2 are <u>unifiable</u> & σ is the <u>unifier</u>.

How do we solve constraints? We transform a set of constraints using the following rules:

$\{ C_1, C_2, \dots C_n, int = int \} \Rightarrow \{ C_1, \dots C_n \}$

$\{ C_1, C_2, \dots C_n, bool = bool \} \Rightarrow \{ C_1, \dots, C_n \}$

$\{ C_1, \dots C_n, \alpha = \tau \} \Rightarrow \{ [\tau/\alpha]C_1, [\tau/\alpha]C_2, \dots, [\tau/\alpha]C_n \}$

$\{ C_1, \dots C_n, \tau = \alpha \} \Rightarrow \{ [\tau/\alpha]C_1, [\tau/\alpha]C_2, \dots, [\tau/\alpha]C_n \}$

$\{C_1, \cdots, C_n, \tau_1\text{-list} = \tau_2\text{-list}\} \Rightarrow$
$\{C_1 \cdots C_n, \tau_1 = \tau_2\}$

$\{C_1, \cdots, C_n, (\tau_1 \to \tau_2) = (\tau_1' \to \tau_2')\} \Rightarrow$
$\{C_1, \cdots, C_n, \tau_1 = \tau_1', \tau_2 = \tau_2'\}$

$\{C_1, \cdots, C_n, (\tau_1 * \tau_2) = (\tau_1' * \tau_2')\} \Rightarrow \{C_1, \cdots, C_n, \tau_1 = \tau_1', \tau_2 = \tau_2'\}$

One important caveat: we will not allow constraints of $\alpha = \tau$ where $\alpha \in FV(\tau)$. For example we will not allow

$$\alpha = int \to \alpha$$

This would lead to $\alpha = int \to (int \to \cdots)$ a never ending expression. (There is a way of making sense of these expressions but it is outside the scope of this class.)

Before we introduce a constraint of the form $\alpha = \tau$ we will check if α occurs in τ: this is poetically called an "occurs-check."

For example: $\{\alpha_1 \to \alpha_2 = int \to \beta, \beta = \alpha_2 \to \alpha_2\}$
$\Rightarrow \{\alpha_1 = int, \beta = \alpha_2, \beta = \alpha_2 \to \alpha_2\}$
$\Rightarrow \{[\alpha_2/\beta]\alpha_1 = [\alpha_2/\beta]int, [\alpha_2/\beta]\beta = [\alpha_2/\beta](\alpha_2 \to \alpha_2)\}$
$\Rightarrow \{\alpha_1 = int, \alpha_2 = \alpha_2 \to \alpha_2\}$ OCCURS-CHECK FAILS.
$\Rightarrow$ NOT UNIFIABLE

We can also fail if expressions do not match so $\tau\text{-list} = \tau_1 \to \tau_2$ will immediately fail for example. So will $(\tau_1 * \tau_2) = (\tau_3 \to \tau_4)$.

The unification algorithm continues until the set of constraints is empty or none of the rules can be applied.

COMP 302
Mar. 7, 16
Next week Wed, and Friday we will have 2 guest lectures of the topic Continuation.
A3 is due, A4 is put up, about unification, key inference.
A5 is about parsing and compiling.
Meta-circular interpreter
language with let and lambda(known as fun)
(the interpreter code/example is on the website)
(this whole week will be about how to deal with language processing)

(make a new F# type called lambda term)

```
type lterm =
  |  Num of int
  |  Var of string
  |  Apply of lterm *lterm
  |  Plus of lterm *lterm
  |  Lambda of string * lterm (take a variable name and a lterm, value)
        (eg: fun x -> body will become Lambda("x", <body>)
  |  Let of (( string * lterm) list) *lterm;;
        (eg: let x=1,y=2,z=3 in <foo>
              let(["x", Num1);*"y",Num2);("z",Num3)], <foo>)

type results  =
  | Int of int
  | Closure of (lterm * envs) (function closure: can be view as value)
        (the crucial thing of closure is the function has to remember
the environment so the value doesn't  change based on other thing)
  | Unbound
and (this is used for mutually recursive, envs defines layer, layer
defines results, results(closure) defines envs.)
  layer = Layer of (lterm * results) list
and
  envs = Env of layer list;;
(the environment is going to be a bunch of layers with bindings(a name
and a closure, aka results) inside stack on each other.)
```

Lambda(x, y) stands for fun x-> y

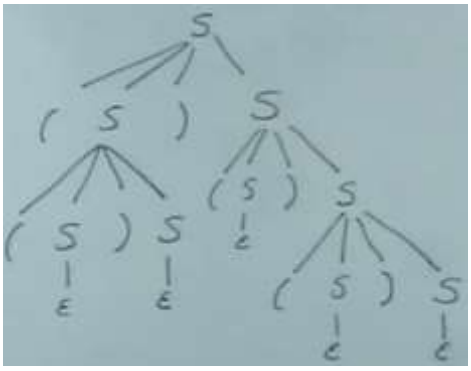
(fun x -> <body>, c-env) ARGUMENT
this is evaluated as making a layer of c-env and put a layer of package (argument x|arg) on top of it.

```
newenv((make_new_layer(binding_list, env),env)));;
(you make a new layer and put it on top of env layer)
```

this interpreter code is turing complete, no type system.

COMP302
2016.03.09, wed.
Studying syntax today
Syntax: the theory of grammar, it tells you what are correctly structured sentences.
think about: how does one construct sentences?
which in programming, equivalent to What are legal programs?
Chomsky invented Context-free grammars
the process of recongnition is called: parsing.
A5: write a parser, which is what today's lecture about.

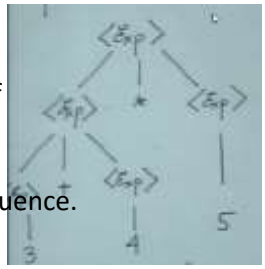
How do we describe grammer?
you can expend the ideas and form a tree.
root: sentence, children: subject, verb, object
object has 2 children: direct object, indirect object
direct o has 1 child: noun phrase… etc keep expending the tree
in the end you get a terminal symbol, which is you get a sentence.
the basic idea of the syntax is not about the meaning of the result, that's another subject: sematix.

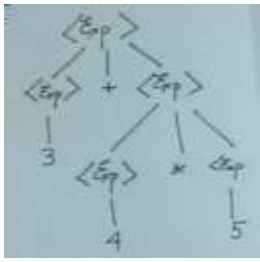


terminal symbols: frodo, the, ring, cast, into, fire
non-terminal symbols: <object>, <verb>, <noun>… Root: <sentence>

example: we have {(,) } 2 items, left and right bracket
how do we know if (), (()), ((((())))) is good, and)(is bad,
we need a rule.
S-> (s) s | ε (the symbol epsilon means empty.
rules for generating strings of parens.

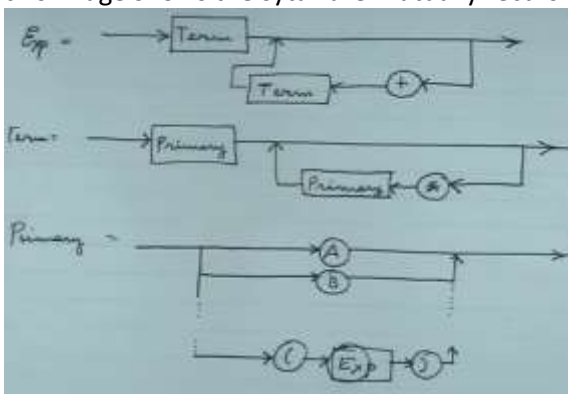
we need grammars that capture the connect grouping too.
example: 3+4*5 is not equalt to (3+4)*5
theres notes on the website:
http://www.cs.mcgill.ca/~prakash/Courses/Comp302/Notes/parsing.pdf
they both shares the same rule:
<exp> -> <exp> | <exp>+<exp>|<exp>*<exp>| none
it's ambiguous when a grammar produces 2 or more parse tree for a sequence.





goal of parsing: start out with a long string, you want to extract from it

this image shows the sytax are mutually recursive



COMP302
2016.03.11 Fri

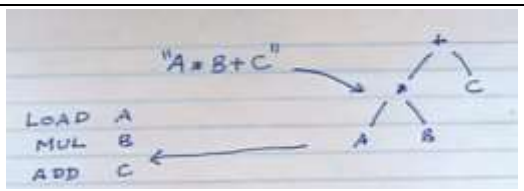
continue wednesday's lecture
stream of class goes into a box,
the box contains tokenizer (lexical analizer) which takes a stream of characters and group them into meaningful symbols.
stream of tokens comes out from the box.
It is then fed into a Parser
and a tree comes out from the parser.
The tree then get translated fed to code generator and machine code comes out.
A5:
q1: parser
q2: code generator

Machine language:
1 register machine (called accumulator which can perform various tasks)
we assume variable names get translated to memory locations
we also need some tempory storge which is written in integers.

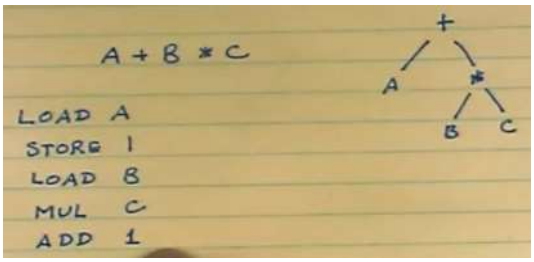
Machine instruction set:
LOAD X takes memory from x and puts in an accumulator.
or
LOAD N takes an integer and put into an accumulator.
opposite:
STORE X / STORE N takes a value from accumulator and puts it back to memory
(load wipes out whatever was in the accumulator and replace it)
ADD X / ADD N (take the value in x and add it into the accumulator which update it)
MUL X / MUL N
THER IS NO CONTROL FLOW.

---

example : a*b+c
load a
mul b
add c



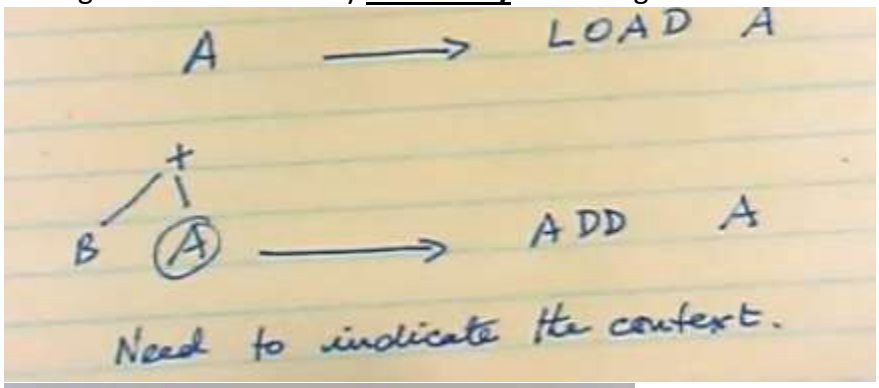
eg: A+B*C
load a
store 1
load b (which wipes out a in accumulator)
mul c
add 1 (which add a into b*c)

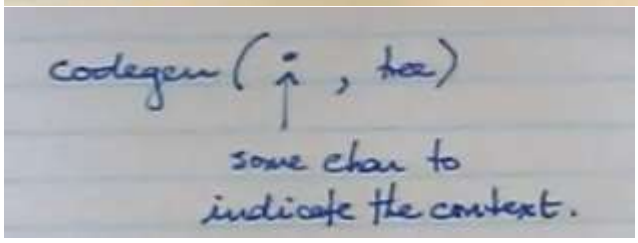


eg: a*b + c*d
load a
mul b
store 1
load c then mul b then add 1

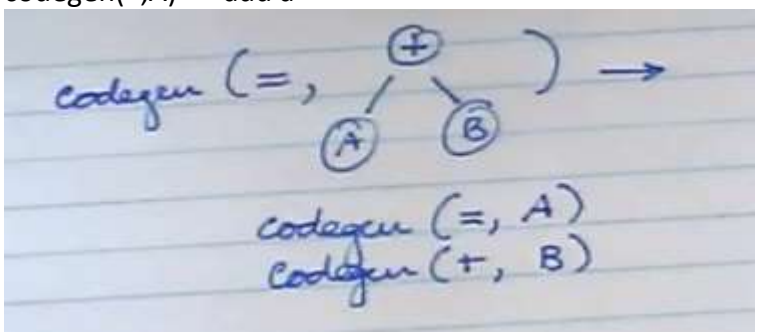
COMP302
2016.03.11 Fri

Code generation is done by **recursively** traversing the tree.

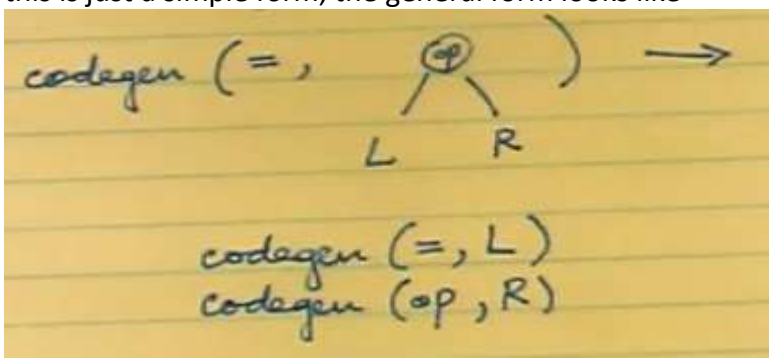




= is fresh computation
+ is something has been computed on the LHS, now we want to add the RHS
* same manner for multiplication
codegen(=,A) -> load a
codegen(*,A) –> mul a
codegen(+,A) –> add a

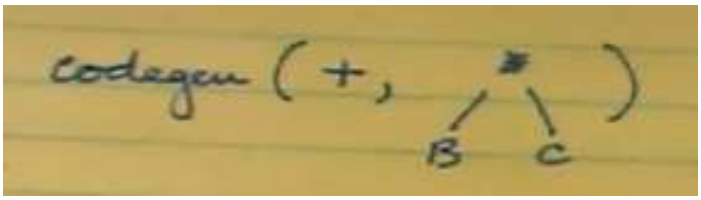


this is just a simple form, the general form looks like

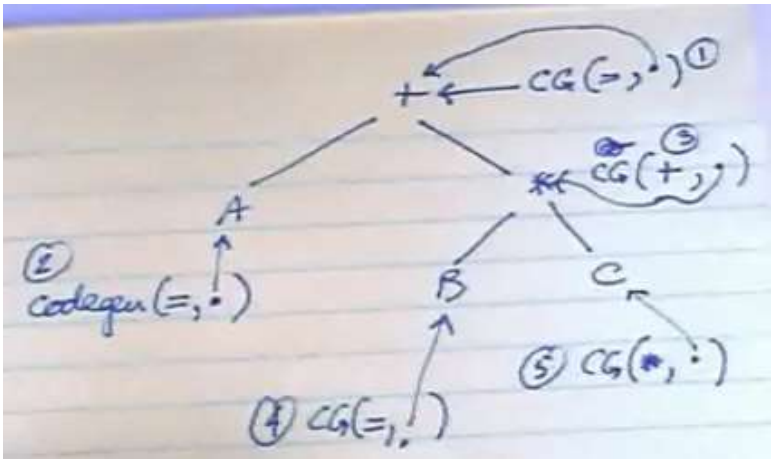


op stands for operator

COMP302
2016.03.11 Fri



Now we have this thing
temp. storage needed
memory := memory +!
output(STORE, memory)
then recursively call
codegen(=, L)
codege(op, R)
below is the only place where you need memory allocation
if op = '+' then
output(ADD, memory)
else if op='*' then
output(MUL, memory)
memory:=memory-1

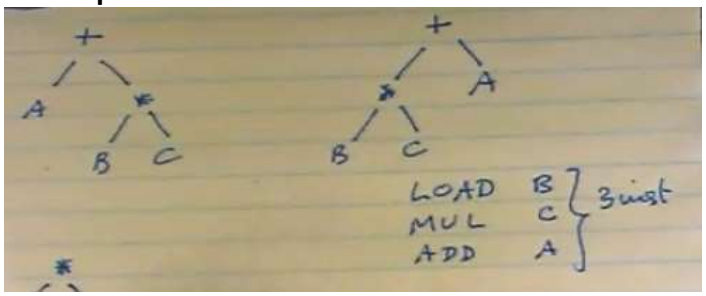


quickly run through an example: a+b*c
call numbers goes as:

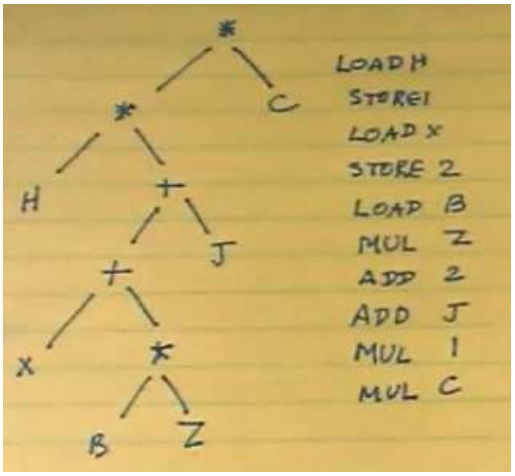
1. top level call begins at '+' : CG(=, .)
2. then CG(=, .) at 'A'
3. then CG(+, .) at '*'
4. CG(=, .) at 'B'
5. then CG(=, .) at 'C'

LOAD A
STORE 1
LOAD B
MUL C
ADD 1

Code optimization

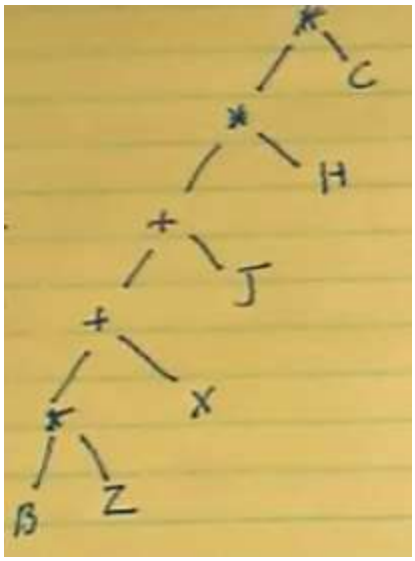


the RHS is more simpler than LHS



this takes 10 steps, 2 extra cells

COMP302
2016.03.11 Fri



this is the optimized version

LOAD B
MUL Z
ADD X
ADD J
MUL H
MUL C

6 steps, no extra memory cell

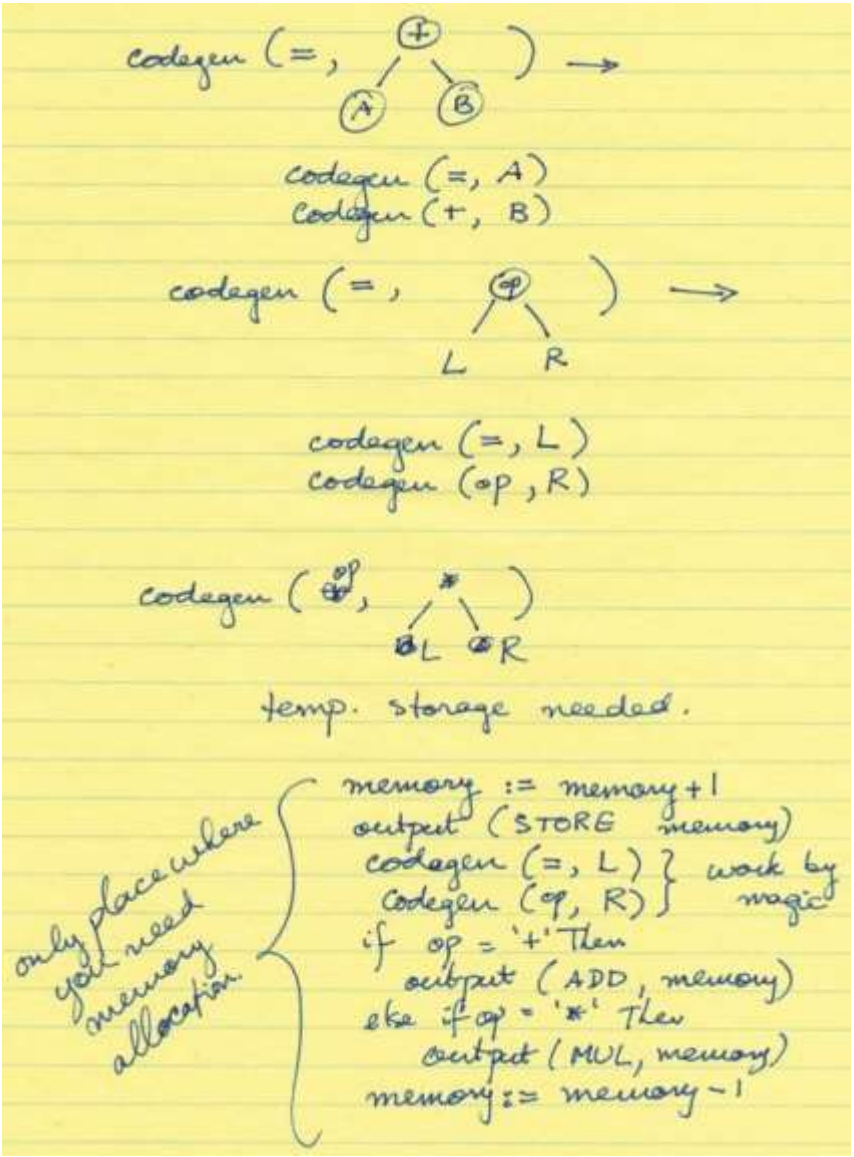
a5 q1 psudo code :

```
procedure Exp;
  begin
    Term;
    while sym = '+' do
      begin
        getsym; term
      end
  end
procedure Term;
  begin
    primary;
    while sym = '*' do
      begin getsym; primary end
  end

procedure primary;
  begin
    if sym in ['A'..'Z'] then getsym
    else if sym = '(' then
      begin getsym; expression;
        if sym <> ')' then error else getsym
      end
    else error
  end
```

In your code these should be functions that return trees. This code only says whether there is an error or not, but it gives the basic control flow. PLEASE produce expression trees not parse trees.

COMP302
2016.03.11 Fri

a5 q2 :



Continuations

by Fernando Serboncini (fserb@fserb.com)

```
let forbidden a b =
  let mutable sum = 0
  for i = a to b do
    sum <- sum + i
  sum

forbidden 3 9 // ==> 42
```

```
let rec myFold f v s = // already in the standard library.
  match s with
    | [] -> v
    | (x::xs) -> f x (myFold f v xs)

let answer a b = myFold (+) 0 [a..b]

answer 3 9 // ==> 42
```

```
let awesomestFunction f x = f (f x)

awesomestFunction (fun x -> x * x) 3 // ==> 81
```

```
let creator x =
  let creature y = x * y
  creature

let c1 = creator 1
let c2 = creator 2

c1 3 // ==> 3
c2 3 // ==> 6
```

```
let divide_failwith x y =
  if y = 0 then failwith "Divisor cannot be zero."
  else
    x / y

let do_something a b =
  if (a = 0) then 0 else (divide_failwith a b)

do_something 4 2  // ==> 2
do_something 4 0  // ==> ??
```

```
System.Exception: Divisor cannot be zero.
  at divide_failwith (Int32 x, Int32 y)
  at do_something (Int32 a, Int32 b)
  at main@ ()
Stopped due to error
```

```
let print_number x =
  printf "%.0f\n" x
  System.Environment.Exit 0

let add_and_print x =
  let y = 17.0 + x
  print_number y

let square_add_and_print x =
  let y = x * x
  add_and_print y

square_add_and_print 5.0  // prints 42 and exits.
```

```
let print_number x =
  printf "%.0f\n" x
  System.Environment.Exit 0

let calculate x cont =
  let y = x * x + 17.0
  cont y

calculate 5.0 print_number  // prints 42 and exits.
```

```
// Ends everything.
let finish () =
  System.Environment.Exit 0

// Checks that a @pass is valid for a @user and continues.
let check_password user pass cont =
  printfn "Password is OK."
  cont user

// Do login input, checks for password and continues.
let do_login cont =
  printf "login: "
  let u = System.Console.ReadLine()
  printf "password: "
  let p = System.Console.ReadLine()
  check_password u p cont

// Greets @user and finishes.
let greet_user user =
  printfn "Hey, %s, welcome!" user
  finish ()

do_login greet_user
// login: drumpf
// password: 1729
// Password is OK.
// Hey, drumpf, welcome!
```

```
// Checks that a @pass is valid for a @user.
// If it is @valid_cont, else @fail_cont.
let check_password user pass valid_cont fail_cont =
  if pass = "1729" then
    printfn "Password is OK."
    valid_cont user
  else
    printfn "Wrong password."
    fail_cont ()

// Do login input, checks for password and continues.
let rec do_login login_cont =
  printf "login: "
  let u = System.Console.ReadLine()
  printf "password: "
  let p = System.Console.ReadLine()
  let when_fail = fun () -> do_login login_cont
  check_password u p login_cont when_fail

// Greets @user and finishes.
let greet_user user =
  printfn "Hey, %s, welcome!" user
  finish ()

do_login greet_user
// login: drumpf
// password: 1
// Wrong password.
// login: drumpf
// password: 1729
// Password is OK.
// Hey, drumpf, welcome!
```

```
let rec stack_sum n =
  if n = 0 then 0
           else n + stack_sum (n - 1)
```

```
stack_sum 1000     // ==> 500500
stack_sum 1000000  // ==>
// Process is terminated due to StackOverflowException.
```

```
let rec cont_sum n cont =
  if n = 0 then cont 0
           else cont_sum (n - 1) (fun x -> cont (x + n))

cont_sum 1000 print_number   // prints 500500
```

```
cont_sum 1000000 print_number  // prints 1784293664
```

```
type 'a tree =
  | Empty
  | Branch of  'a * 'a tree * 'a tree

//     a
//    / \
//   b   c
//  / \   \
// d  e    f
//        /
//       g
let tree1 = Branch ('a',
                Branch ('b',
                    Branch ('d', Empty, Empty),
                    Branch ('e', Empty, Empty)),
                Branch ('c',
                    Empty,
                    Branch ('f',
                        Branch ('g', Empty, Empty),
                        Empty)))
```

```
let rec stack_find el t =
  match t with
    | Empty -> false
    | Branch (x, l, r) ->
      if x = el
        then true
        else if stack_find el l
               then true
               else stack_find el r

stack_find 'g' tree1  // ==> true
stack_find 'h' tree1  // ==> false
```

```
stack_find 'g' tree1 =
  find (A)
    -> find (C)
      -> find (F)
        -> find (G)
          -> true

```

```
let rec cont_find el t succ fail =
  match t with
    | Empty -> fail ()
    | Branch (x, l, r) ->
      if x = el
        then succ ()
        else cont_find el l succ (fun () -> cont_find el r succ fail)

let yes () = printfn "yes"
let no () = printfn "no"

cont_find 'g' tree1 yes no  // prints yes
cont_find 'h' tree1 yes no  // prints no
```

```
let isEven x = (x % 2) = 0

cut [ 2 ; 4 ; 6 ; 5 ; 2 ] isEven  // ==> [ 2 ; 4 ; 6 ]
cut [ 1 ; 3 ; 5 ] isEven          // ==> [ ]
```

```
let cut list pred =
  // we first create our internal CPS function.
  let rec cc list pred cont =
    match list with
      | [] -> cont []
      | x::xs ->
        if (pred x)
          then cc xs pred (fun a -> cont (x :: a))
          else cont []
  // then we call it. Notice that our continuation is
  // the identity function:
  // let id = fun x -> x
  cc list pred id
```

```
let ret =
  try
    failwith "fail"
    "hello"
  with
    | Failure msg -> "caught: " + msg

ret  // ==> "caught: fail"
```

```
let divide_failwith x y =
  if y = 0 then failwith "Divisor cannot be zero."
  else
    x / y

divide_failwith 4 2  // ==> 2
```

```
let divide_failwith x y cont error =
  if y = 0 then error "Divisor cannot be zero."
  else
    cont (x / y)
```

```
let default_error msg =
  printfn "Uncaught exception: %s" msg
  System.Environment.Exit 1
```

```
divide_failwith 4.0 2.0 print_number default_error  // 2.0
divide_failwith 4.0 0.0 print_number default_error
    // Uncaught exception: Divisor cannot be zero.
```

```
// we must call close_database. So instead of:
// divide_failwith x y print_number default_error
// we have:
let cont = fun x ->
  close_database ()
  print_number x
let fail = fun x ->
  close_database ()
  default_error x

divide_failwith 4.0 2.0 cont fail
```

```
void OnThumbnailAvailable(RequestContext* context,
                          const GURL& url,
                          const SkBitmap* bitmap) {
  if (bitmap) {
    // ... use bitmap as thumbnail of the page.
  }
}
thumbnail_manager_->GetImageForURL(url, &OnThumbnailAvailable);
```

```
function onKeyPress(event) {
  // some key has just been pressed.
}
document.addEventListener("keypress", onKeyPress);
```

```
let a = 4
let b = 7
let f x = x * x
let g x y = x + y

f (g a b) ==> 121
```

```
let a_mc cont     = cont 4
let b_mc cont     = cont 7
let f_mc cont x   = cont (x * x)
let g_mc cont x y = cont (x + y)

// that can be called as:
b_mc (a_mc (g_mc (f_mc print_number)))  // ==> 121
```

```
// Is @p1 a threat to @p2?
let is_threat n p1 p2 =
  let (x1, y1) = ( p1 % n, p1 / n)
  let (x2, y2) = ( p2 % n, p2 / n)
  x1 = x2 || y1 = y2 || x1 + y1 = x2 + y2 || x1 - y1 = x2 - y2
```

```
// Is @p in conflict with any of positions on @queens?
let rec is_conflict n p queens =
  match queens with
    | [] -> false
    | x::xs -> if is_threat n x p then true else is_conflict p xs
```

```
let rec search n clist moves back =
  match clist with
    | [] -> back ()
    | (p::ps) ->
      if is_conflict n p moves
        then (search n ps moves back)
        else
          let new_moves = moves @ [p]
          if List.length new_moves = n
            then new_moves :: (search n ps moves back)
            else search n ps new_moves
                   (fun () -> search n ps moves back)
```

```
let queens n = search n [0..n*n-1] [] (fun () -> [])
```

```
List.head (queens 8)
// ==> [0; 12; 23; 29; 34; 46; 49; 59]
```

```
List.map (fun n -> List.length (queens n)) [1..10]
// ==> [1; 0; 0; 2; 10; 4; 40; 92; 352; 724]
```

COMP302
continuations
the notes is online
code:
http://www.cs.mcgill.ca/~prakash/Courses/Comp302/Notes/continuation_code_examples.pdf

note:
tbp

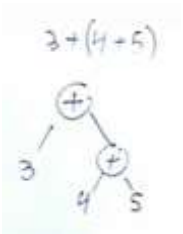
COMP302
Mar. 18, 2016 Friday

Parsing
Grammar
Exp= term+exp|term
term=primary *term | primary
primary =a|b|…|z| (exp) (this are constants
goal: write a CPS parser
(*will not be allowed to use continuations for your assignment.*)
so this lecture wont be giving away the solution for the assignment.

COntinuations control the flow of the program
if we look at primary = constants or (exp)
    success continuation
    failure continuation backtracking
        if it doesn't work, we need to do something else
let rec parsePrimary tokens succ fail =
(*what are tokens?
so there's a tokenizer which reads input strings and output a stream(don't need to worry about what it is, it's just some kind of list)
stream:
    next: provides the next token
    rest: gives the rest of the token stream.
if the input is finished, we receive token EOI(*end of input *)

Token = plus | times | Leftparenthesis | Rightparenthesis |A|B|…|Z|EOI
    '+' '*' '(' ')' 'a' 'b' 'z'
back to
let rec parsePrimary tokens succ fail =
    assume there's a function called parseConstant.
let fail' = fun () (*assume it takes the unit*) -> if next tokens =Lfparen then
                parseExp(rest tokens) succ' fail

parseConstant tokens succ
let succ' = fun e token' -> if next tokens' = RParen then succ
        (*rest tokens'*)
    else fail()
type exp = sum of exp*exp
    | prod of exp*exp
    | constant of char



let rec parseTerm tokens succ fail =
    let fail ' = fun () -> parsePrim tokens succ fail
    let succ' = fun e tok -> if next tok = times then parseTerm(rest tok)
                (fun e' tok -> succ (prod e e') tok)
            else fail' ()
parsePrim tokens succ fail' (*if the function success then we check if there's a term after and keep on checking, else we return primary*)

COMP302
2016.03.18
Friday
Token = plus|times|LParen|RParen|A|B|,,,|Z| EOI
'+' '*' '(' ')' 'A' …. '\0'
let rec parsePrim tokens succ fail =
second things: let fail' = fun () -> if next tokens = LParen
then call a function call parseExp(rest tokens) succ' fail

first thing: the idea is call a function called
parseConstant tokens succ

third thing: let succ' = fun e toekns' -> if next tokens' =RParen then succ e
(rest tokens')
else fail ()

fourth thing: type exp = sum of exp*exp
|product of exp*exp
| constant of char

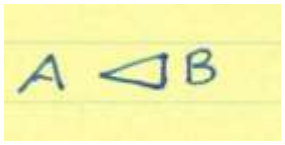
COMP302
2016.03.21 monday

last week's lectures are not testible.
subtyping is induced by various things from object oriented programming from polymorphism.
what is the point of OOP?
it focus on objects rather than functions.
the first OOP was called SIMULA 67 (Norway)
we can think of object as record with some "active" fields
OOP have two important features:

- code reuse -> inheritance.
- generality -> subtyping.

let's ilustrate the concept in JAVA, class A extends class B } **A subtype of B inherit code**

class A implements interface B -> pure subtyping

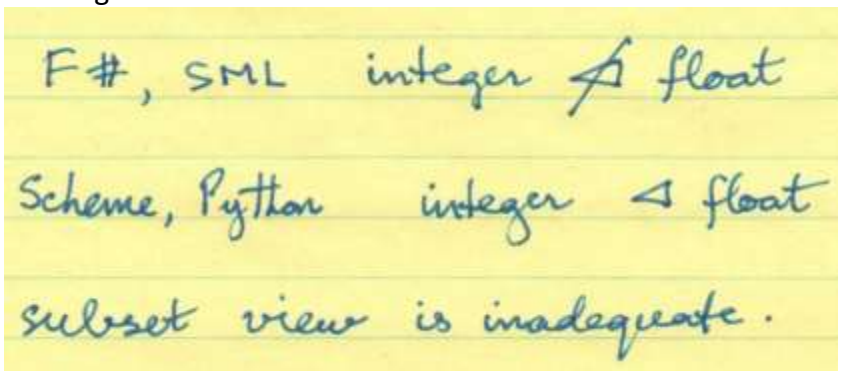


Thi is A is a subtype of B

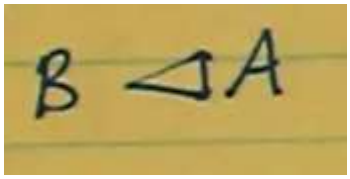
If my computation wants a B value it will be happy with given an A value. (esscence of this lecture)

CRUDE view: subtyping is the same as subset inclusion.

"isa"
a dog is a mammal
a BMW is a car
an integer is a real number

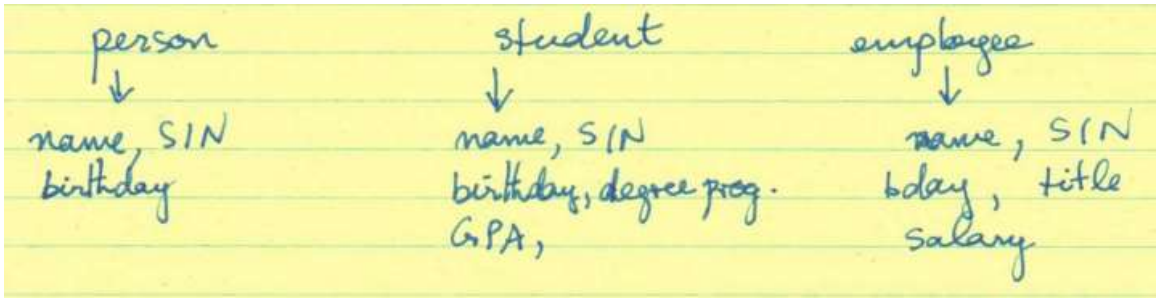


Contract : a type declaration signs a contract.
record have fields
A-> some fields
B-> all the same fields as A plus some more

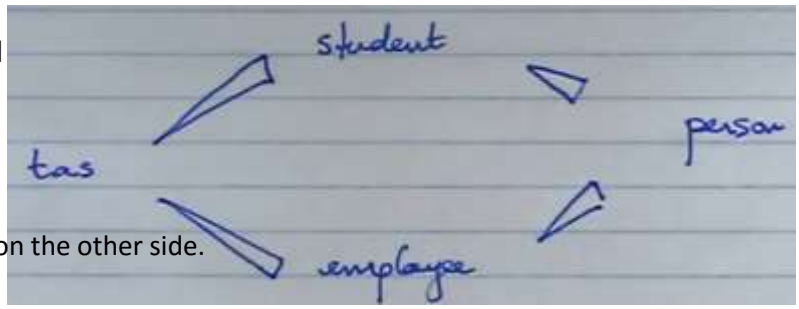


you should be happy to be given B value if you want A.

COMP302
2016.03.21 monday



student is a subtype of person
reading as a contract: i want a person record
student is a subtype of employee as well
TAs is a subtype of student and employee



this is called a subtyping diamond.
it is pure subtyping on one side and inheritance on the other side.
TAs is a subtype of person too.

Typing rules

$$\frac{\Gamma \vdash e : A \qquad A \lhd B}{\Gamma \vdash e : B}$$

$$\frac{}{T \lhd T} \qquad \frac{S \lhd R \qquad R \lhd T}{S \lhd T}$$

COVARIANT

$$\frac{S_1 \lhd T_1 \qquad S_2 \lhd T_2}{S_1 * S_2 \lhd T_1 * T_2} \qquad \frac{S_1 \lhd T_1 \quad T_2 \lhd T_2}{S_1 * T_2 \lhd T_1 * T_2}$$

$$\frac{S \lhd T}{S\ \text{list} \lhd T\ \text{list}}$$

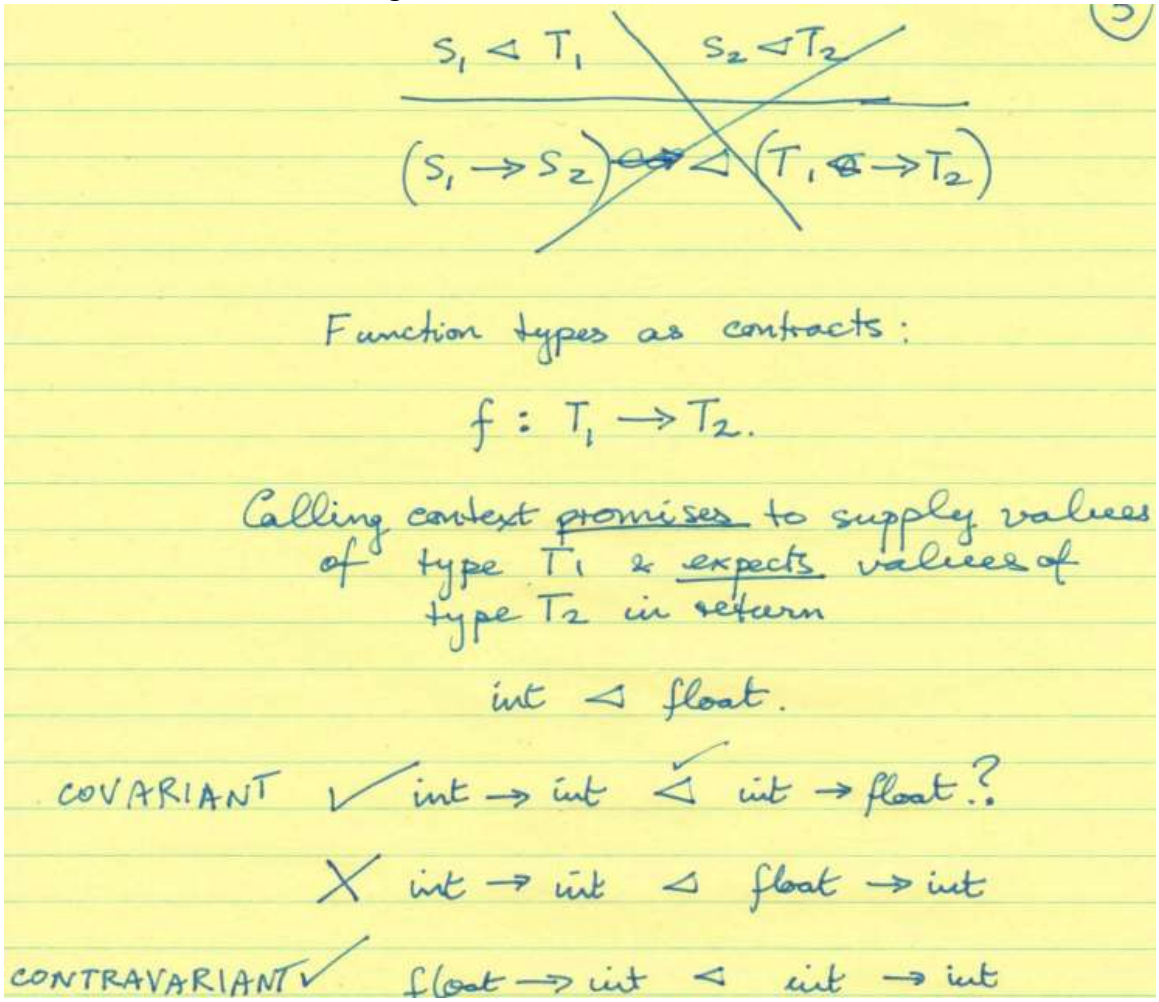
$$\frac{S_1 \lhd T_1 \qquad S_2 \lhd T_2 \quad \cdots \quad S_k \lhd T_k}{\{x_1 : S_1, \cdots x_k : S_k\} \lhd \{x_1 : T_1, \cdots x_k : T_k\}} \text{ depth}$$

$$\frac{k \leq n}{\{x_1 : S_1 \cdots x_n : S_n\} \lhd \{x_1 : S_1, \cdots x_k : S_k\}} \text{ width}$$

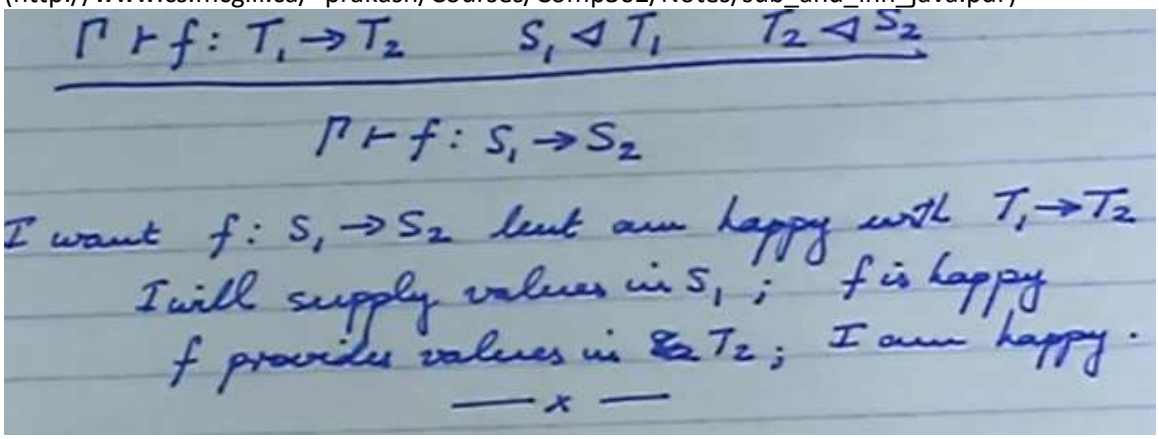
→ CONTRAVARIANT.

COMP302
2016.03.21 monday

the cross indicates that the logic of this relation isnt true.



COMP 302
2016.03.23
(http://www.cs.mcgill.ca/~prakash/Courses/Comp302/Notes/sub_and_inh_java.pdf)



if i have somehow figure f has type T1-> T2
then f is s1 -> s2

notes are typed in a LATEX file on the website.

Inheritance -> code reuse
subtyping -> code generality

myInt-> n:private
show, add, increment

gaussian integers a+ib (a and b are integers, i is sqrt(-1))

➔ extension of myINt

gaussInt is a subtype of myInt
gaussInt inherits methods from myInt
gaussian defines

- new add function -> different cignature (type)
- new show function -> OVERRIDING

both functions will co-exist which OVERLOADING (as opposed to overriding)

Rules for Method Lookup and Type Checking.

First the rules. There are two phases: compile time, which is when type checking is done and run time, which is when method lookup happens. Compile time is before run time.

- The type checker has to say that a method call is OK at compile time.
- All type checking is done based on what the declared type of a reference to an object is.
- Subtyping is an integral part of type checking. This means if B is a subtype of A and there is a context that gets a B where A was expected there will not be a type error.
- Method lookup is based on actual type of the object and not the declared type of the reference.
- When there is overloading (as opposed to overriding) this is resolved by type-checking.

there will be many questions related to this on the final.

COMP 302
2016.03.23
(http://www.cs.mcgill

3.1 Analysis of the gaussInt Example

Here is an example exhibiting both overriding and overloading.

```
class myInt {
  private int n;
  public myInt(int n){ this.n = n;}

  public int getval(){return n;}

  public void increment(int n){ this.n += n;}

  public myInt add(myInt N){ return new myInt(this.n + N.getval());}

  public void show(){
    System.out.println(n);}

}
```

hence of NEW

extend = inherit

```
class gaussInt extends myInt {
  private int m;  //represents the imaginary part

  public gaussInt(int x, int y){
    super(x);
    this.m = y;}
  public void show(){
    System.out.println(
    "realpart is: " + this.getval() +" imagpart is: " + m);}

  public int realpart() {return getval();}
  public int imagpart() {return m;}
```

hence of NEW

```
  public gaussInt add(gaussInt z){
    return new gaussInt(z.realpart() + realpart(),
                        z.imagpart() + imagpart());
  }

  public static void main(String[] args){
    gaussInt kreimhilde = new gaussInt(3,4);
    kreimhilde.show();
    kreimhilde.increment(2);
    kreimhilde.show();

    System.out.println("Now we watch the subtleties of overloading.");
    myInt a = new myInt(3);
    gaussInt z = new gaussInt(3,4);
->  gaussInt w;   //no object has been created
    myInt b = z;  //b and z are names for the same object
    //even though b and z refer to the SAME object they have
    //different types

    System.out.print("the value of z is: "); z.show();
    System.out.print("the value of b is: "); b.show();
    //which show method will be used?

    myInt d = b.add(b); //this does type check
    System.out.print("the value of d is: "); d.show();
    // w = z.add(b); will not type check
    // w = b.add(z); will not type check
    w = ( (gaussInt) b).add(z);//this does type check
    System.out.print("the value of w is: "); w.show();
    myInt c = z.add(a); //will this typecheck?
    System.out.print("the value of c is: "); c.show();

  }
}
```

Name	Declared Type	ActualType
a:	myInt	myInt
z	gaussInt	gaussInt
w	gaussInt	TBD
b	myInt	gaussInt
d	myInt	myInt
c	myInt	TBD

```
myInt a = new myInt(3);
gaussInt z = new gaussInt(3,4);
gaussInt w;
myInt b = z;

System.out.println("the value of z is"+ z.show());

> real part is 3 imag part is 4
```

this prints out the above line because z is declared to be of type `gaussInt`. It passes the type checker as there is a `show` method defined in the `gaussInt` class. At run time it uses the `show` method of `gaussInt` to display the above line.

```
System.out.println("the value of b is :" + b.show());

> real part is 3 and imag part is 4.
```

b is declared to be of type `myInt`. There is a method called `show` in the `myInt` class. The type checker sees that and because of that it passes the type checker, **but** the actual type of b is `gaussInt`. Method lookup is based on actual types of objects and therefor b uses the `show` method in the `gaussInt` class and displays what a `gaussInt` object would have shown.

```
myInt d = b.add(b)

System.out.println("the value of d is:"+ d.show());

> 6
```

b is declared to be of the type `myInt`, the type checker checks to see whether there is an `add` method in the `myInt` class. **Yes** there is one; it takes a `myInt` object and returns a `myInt` object as the result. At run time b's actual type is `gaussInt` the run-time system checks to see if there is an `add` method in the `gaussInt` class which matches the type that it was told by the type-checker. There are two add methods - one that takes a `myInt` and returns a `myInt` (This method has been inherited from the `myInt` class). The other takes a `gaussInt` and returns a `gaussInt`; this is the method that is explicitly defined in the `gaussInt` class. However the latter method does not match what the type-checker told the run-time system to expect.

NOW WHICH ADD METHOD DO WE USE?

since "When there is overloading, it is resolved by typechecking" the method which takes an object of the type `myInt` will be used. This is the method that has been inherited. It takes in a `myInt` and returns a `myInt`. Hence `b.add(b)` returns a `myInt` object and therefor NOW the actual type of d is `myInt`.

```
//w= z.add(b) -----(i)
//w = b.add(z)-----(ii)
```

These two will not type check

1. z is declared to be of the type `gaussInt`. There are two methods in the `gaussInt` class, the one that takes in a `myInt` object and returns a `myInt` object is used. Why? Once again overloading is resolved by typechecking. Since b is declared to be a `myInt` object it will pick the `add` method that it inherited.

   z is a `gaussInt` which is a subtype of `myInt` and hence is added to b and returns a `myInt`. w is declared to be a `guassInt`. Since `myInt` is not a subtype of `gaussInt` the assignment statement will not accept this for the right hand side, and hence would cause an error.

2. b is declared to be of the type `myInt`. The type checker checks if there is an `add` method in the `myInt` class there is one which expects a `myInt` object and returns a `myInt`object z is a `gaussInt` and since `gaussInt` is a subtype of `myInt`, b is added to z to produce a `myInt` object

   w is declared to be a `gaussInt` Since `myInt` is not a subtype of `gaussInt` it will not accept it and hence would cause an error.

```
w = ( (gaussInt) b).add(z)
```

This does type check as it is just a little modification to case 2 above. Now since w is a `gaussInt`, it better get a `gaussInt` on the right hand side. However, now, because of the cast, the type-checker knows that b is really a `gaussInt`. Thus, it now has to choose between two possible `add` methods. To resolve the overloading it uses the declared types; z has declared type `gaussInt`. Thus when it resolves the overloading of the `add` method it figures out to use the `gaussInt` to `gaussInt` version.

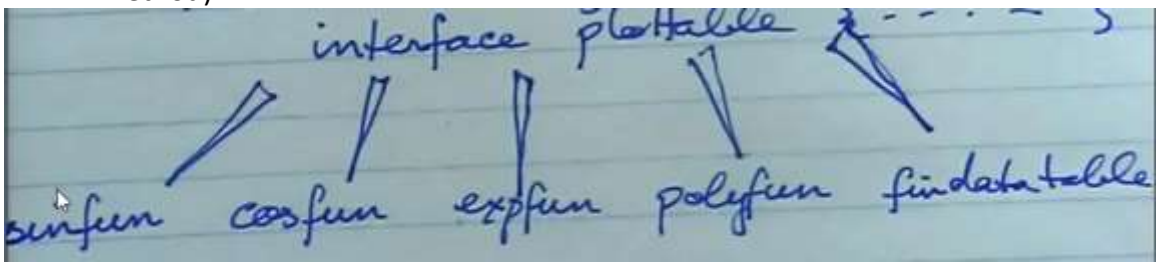
COMP 302
2016.03.23
(http://www.cs.mcgill.ca/~prakash/Courses/Comp302/Notes/sub_and_inh_java.pdf)

SUBTYPE POLYMORPHSM

this is more used then paremetric polymorphism.
if i got A is a subtype of B then
code fro B will work on A without modification.

Example: code to plot the graph of a function.
we don't want to write a program that only plot sin function or cos function or…etc.
the way to go is subtype polymorphsm,
in order to plot a graph of a function you want:

- function objects that have a method called "y" which accepts values of type float(/double) and returns float(/double).
- the actual plotting code does not care how y was obtained.
  so we define a **interface plottable** {…}, which is some type that takes a method y and returns a float and takes in a float. (no code here, just a declaration of the method)



plot method is declared to accept an object of type plottable.

COMP302
2016.03.30 topic: scheme

```
(define (fact n)
  (define (helper n m)
    (if (= n 0) m
        (helper (- n 1) (* n m))))
  (helper n 1))
;Value: fact

(fact 3)
;Value: 6
```

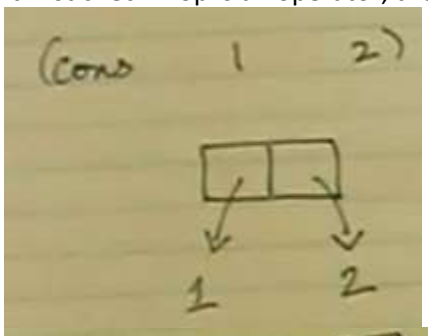
this is written in scheme, a language derived from LIPS
there's no type system, you can write nonsense.

scheme is ideal for writing program that manipulate other program's source code, this is called reflective programming.
;; the fact program is S expressions (syntactic).

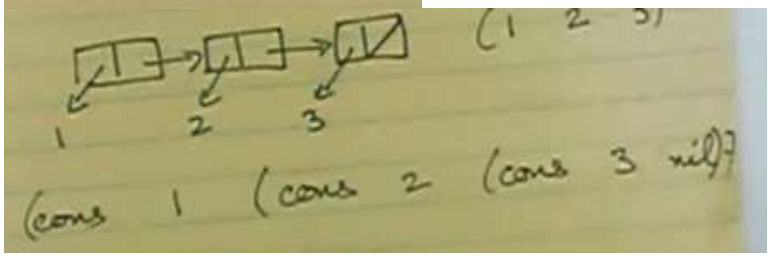
```
;; Making pairs

(cons 1 2)
;Value 11: (1 . 2)

(define foo (cons 1 2))
;Value: foo
```

cons stands for constructor
cons is a special form, in stead of creating computation, it creats data structure.

the fundamental rule is s-expression
which looks like (op arg1….argn)
it first check if op is an operator, then evaluate all the arguments (pass the values to the op)



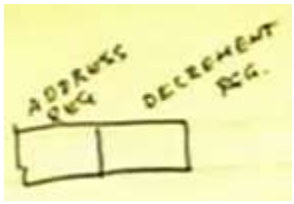
cons 1 2 creates con cell like this
display form: (1 . 2)



you write nil to avoid the dot.
writing (list 1 2 3) is the same as (cons 1 (cons 2 (cons 3 nil)))

COMP302
2016.03.30 topic: scheme

how do we access list structure?



you have a cell (contains address reg & decrement reg)
the way to access it is: car
you want to access the decrement reg: cdr
you write cdr “cudder” “mudda”

let’s see this in action: (in scheme)

```
(define foo (list 1 2 3 4))
;Value: foo

foo
;Value 14: (1 2 3 4)
```

```
(car foo)
;Value: 1

(cdr foo)
;Value 15: (2 3 4)
```

here’s a example of no type system:

```
(list 1 "foo" '(a b c))
;Value 16: (1 "foo" (a b c))
```

by writing ‘(1 2 3) instead of (1 2 3), you are saying to view it as a structure.

```
(define moo '(1 2 3))
;Value: moo

moo
;Value 17: (1 2 3)
```

Compare between 2 things:

```
; Equality for numbers
(= 1 2)
;Value: #f

(= 1.0 1.1)
;Value: #f

(= 1.0 1)
;Value: #t

; Equality for structures
(define a (cons 1 2))
;Value: a

(define b (cons 1 2))
;Value: b

(equal? a b)
;Value: #t

; But they are different cells in memory.  eq? tests pointer equality
(eq? a b)
;Value: #f
```

COMP302
2016.03.30 topic: scheme

Lots of examples to look at:

```
(cons 1 (cons 2 (cons 3 (cons 4 '()))))
;Value 13: (1 2 3 4)

;; Constructed form "cons" cells but if it is a list it must end in
nil.  A
;; dot is not displayed in that case.  If there is a dot at the end it
;; means that one does not have a proper list.

(cons 1 (cons 2 (cons 3 4)))
;Value 14: (1 2 3 . 4)

;; The function "list" makes a list.  It is NOT a free constructor like
cons.
(list 1 2 3 4)
;Value 15: (1 2 3 4)

;; There are many standard list operations: map, filter etc. just like
F#.

(map (lambda (n) (+ n 1)) '(1 2 3))
;Value 19: (2 3 4)

(filter odd? (list 1 2 3 4 5 6 7))
;Value 21: (1 3 5 7)   //select odds only

; Lists can be nested
'((1 2 3) (4 5 6) (7 8 9))
;Value 22: ((1 2 3) (4 5 6) (7 8 9))

; Unlike F# they can be heterogeneous.
(list '(1 2 3) '((4 5) (((6)))) 7)
;Value 23: ((1 2 3) ((4 5) (((6)))) 7)

(length '(1 2 (3 4 5)))
;Value: 3
; Did you understand why it got 3? instead of 5.
because the top level you see 3, and the fact that the 3rd level of the
nested list is not taking into a count.


; Here is a useful list function defined by recursion.
(define (nth n l)
  (if (= n 0)
      (if (null? l) //list empty or not
          (print "List too short")
          (car l))
      (nth (- n 1) (cdr l))))
;Value: nth

(nth 5 '(1 2 3 4 5 6 7 8 9))
;Value: 6
; Huh?  Oh, we are counting from zero!

(nth 5 '(1 2 3))
;you can trace the executions.(no type checking, but tracing in scheme)
```

COMP302
2016.03.30 topic: scheme

```
[Entering #[compound-procedure 24 nth]
    Args: 5
          (1 2 3)]
[Entering #[compound-procedure 24 nth]
    Args: 4
          (2 3)]
[Entering #[compound-procedure 24 nth]
    Args: 3
          (3)]
[Entering #[compound-procedure 24 nth]
    Args: 2
          ()]
;The object (), passed as the first argument to cdr, is not the correct
type.
;To continue, call RESTART with an option number:
; (RESTART 7) => Specify an argument to use in its place.
; (RESTART 6) => Return a value from the advised procedure.
; (RESTART 5) => Return a value from the advised procedure.
; (RESTART 4) => Return a value from the advised procedure.
; (RESTART 3) => Return a value from the advised procedure.
; (RESTART 2) => Return to read-eval-print level 2.
; (RESTART 1)

;Abort!

 => Return to read-eval-print level 1.

;; What went wrong?  No idea!

;; We will trace the execution.
(trace nth)
;Unspecified return value

(nth 3 (list 1 2))

[Entering #[compound-procedure 24 nth]
    Args: 3
          (1 2)]
[Entering #[compound-procedure 24 nth]
    Args: 2
          (2)]
[Entering #[compound-procedure 24 nth]
    Args: 1
          ()]
;The object (), passed as the first argument to cdr, is not the correct
type.
;To continue, call RESTART with an option number:
; (RESTART 5) => Specify an argument to use in its place.
; (RESTART 4) => Return a value from the advised procedure.
; (RESTART 3) => Return a value from the advised procedure.
; (RESTART 2) => Return a value from the advised procedure.
; (RESTART 1) => Return to read-eval-print level 1.

;; OK, I see. The test for empty list is in the wrong place.

(define (nth n l)
  (if (null? l)
```

```
      (pp "List too short.")
      (if (= n 0) (car l)
          (nth (- n 1) (cdr l)))))
;Value: nth

(nth 5 '(1 2 3))
"List too short."
;Unspecified return value



;; If you just type in (1 2 3 4) it will think that you want to apply
the
;; function "1" (remember there is no type system) to the following as
;; arguments.

(1 2 3 4)

;The object 1 is not applicable.
;To continue, call RESTART with an option number:
; (RESTART 2) => Specify a procedure to use in its place.
; (RESTART 1) => Return to read-eval-print level 1.

(restart 1)

;Abort!
```

```
; So how does one make a list without using "list"?
; We have a unique feature, the quote which inhibits computation.
'(1 2 3)
;Value 16: (1 2 3)

; Here are some examples of quote in action.
(quote a)
;Value: a

'quote
;Value: quote

; Note that scheme has SYMBOLS as a basic type.  These are not strings.
; They are atomic and cannot be taken apart.

'(The quick brown fox jumps over the lazy dog)
;Value 17: (the quick brown fox jumps over the lazy dog)

; Very handy for symbolic computation.  Here is a sentence generator.

 (define nouns '(rock-star professor student dog cat baby owl))
;Value: nouns

(define articles '(a the))
;Value: articles

(define verbs '((intrans runs) (intrans sleeps) (trans shoots) (intrans
sings) (intrans snores) (trans hugs)))
;Value: verbs
```

COMP302
2016.03.30 topic: scheme

```
(define adverbs '(quickly slowly loudly lazily))
;Value: adverbs

(define adjectives '(old sexy bald crazy cranky silly))
;Value: adjectives

(define (make-sentence)
  (append (make-noun) (make-verb)))
;Value: make-sentence

(define (make-noun)
  (let ((size (random 2)) (noun-chosen (modulo (random 100) 7)))
    (define (iter n)
      (if (= n 0)
          (list (nth noun-chosen nouns))
          (cons (nth (random 5) adjectives)
                (iter (- n 1)))))
    (cons (nth (random 2) articles) (iter size))))
;Value: make-noun

(define (make-verb)
  (let ((adv (nth (random 4) adverbs)) (verb-choice (nth (random 5)
verbs)))
    (let ((verb (cadr verb-choice)) (verb-type (car verb-choice)))
      (if (eq? verb-type 'trans)
          (cons verb (append (make-noun) (list adv)))
          (cons verb (list adv))))))
;Value: make-verb

(make-sentence)
;Value 25: (the cranky professor snores slowly)

(make-sentence)
;Value 26: (a cat snores quickly)

(make-sentence)
;Value 27: (the old rock-star snores loudly)

(make-sentence)
;Value 28: (a cat shoots the old cat loudly)

(make-sentence)
;Value 29: (the cranky rock-star sings lazily)
;LOL

; Can we construct a self reproducing function?


((lambda (x) (list x (list (quote quote) x))) (quote (lambda (x) (list
x (list (quote quote) x)))))
;Value 30: ((lambda (x) (list x (list (quote quote) x))) (quote (lambda
(x) (list x (list (quote quote) x)))))

;; Yes!

;; How did we come up with it?
```

COMP302
2016.03.30 topic: scheme

```
; First try
((lambda (x) (list x x))(quote (lambda (x) (list x x))))
;Value 31: ((lambda (x) (list x x)) (lambda (x) (list x x)))

; We don't have the quote.  Need to put it in.

((lambda (x) (list x 'quote x))(quote (lambda (x) (list x 'quote x))))
;Value 32: ((lambda (x) (list x (quote quote) x)) quote (lambda (x)
(list x (quote quote) x)))

;  This is right but the interpreter expand the abbreviation for quote,
so
;  we will put it in in the unabbreviated form.

((lambda (x) (list x (quote quote) x))
(quote (lambda (x) (list x (quote quote) x))))
;Value 33: ((lambda (x) (list x (quote quote) x)) quote (lambda (x)
(list x (quote quote) x)))

;; Some list programming
(define (flatten ll)
  (if (null? ll)
      '()
      (append (car ll) (flatten (cdr ll)))))
;Value: flatten

(flatten '((1 2 3) (4 5)))
;Value 34: (1 2 3 4 5)

(define (insert-all x l)
  (if (null? l)
      (list (list x))
      (cons (cons x l) (map (lambda (u) (cons (car l) u))
                            (insert-all x (cdr l))))))
;Value: insert-all

(insert-all 1 '(2 3 4))
;Value 36: ((1 2 3 4) (2 1 3 4) (2 3 1 4) (2 3 4 1))

(define (perms l)
  (if (null? l)
      (list '())
      (flatten (map (lambda (x) (insert-all (car l) x))
                    (perms (cdr l))))))

;Value: perms

(perms '(1 2 3))
;Value 37: ((1 2 3) (2 1 3) (2 3 1) (1 3 2) (3 1 2) (3 2 1))

(perms '(a b c d))
;Value 38: ((a b c d) (b a c d) (b c a d) (b c d a) (a c b d) (c a b d)
(c b a d) (c b d a) (a c d b) (c a d b) (c d a b) (c d b a) (a b d c)
(b a d c) (b d a c) (b d c a) (a d b c) (d a b c) (d b a c) (d b c a)
(a d c b) (d a c b) (d c a b) (d c b a))

(exit)
```

COMP302
2016.03.30 topic: scheme

Scheme -> s-expressions as the only compound type
basic types -> strings, ints, floats

mutation but also supports fully functional programs.
has higher order functions
has recursion
has nested local scopes
has closures

no type system

small challenge: write a program in scheme that outputs exactly its own source code. (announce answer on next friday)

next time: lazy computation and infinite structures.

COMP 302
2016.04.04
topic: stream (examable)
Today's lecture is examble, the one on friday and the following ones are not, try to attend the lectures because they wont be recorded.

Streams in F#
thunks: In computer programming, a thunk is a subroutine that is created, often automatically, to assist a call to another subroutine

fun() -> <code> //this is a thunks, when you evaluate this, it produces a closure

seq.delay is used to delay a computation, it can be wrapped together with thunks
you can delay a computation, but then you need to **force a computation**
sequence -> meant for lazy comp evaluation.

eg1: produce infinite sequence of ones.
given a function of type, let's say nat -> …..
this can be used in a stream
you write seq.initInfinite(fun i -> i*i) (this will produce infinte stream of the square of natural numbers)
in f#:
let nat = Seq.initInfinite(fun i -> i)

more primitives functions to look at:
Seq.append: takes 2 streams and append them together, both streams might be infinite. You may ask how to get to the second stream if the 1st is infinite, well, you wont.
But if the first is finite, then it will eventually finished and go to the second stream.

eg2: **//the idea here is x is an item and sigma is a stream, you want to stick x in front of sigma**
let cons x sigma = Seq.append
(Seq.signleton x) sigma
//this takes 1 item and makes a short 1 element stream of it
//then seq.append just glue them together.
eg3: **//i will give you the first nth element of stream**
let first sigma = Seq.nth 0 sigma

let rest sigma = Seq.skip 1 sigma **//skip the first few item and start from sigma**

let rec prefix **//on the assignment 6**
//it takes a number, a stream,
//then the constructor constructs a list with that many items taken from the stream

let rec numsFrom n = **//create a stream of all integer starting from n**
cons n **//stick n to the rest of the stream**
Seq.delay(fun () -> numsFrom (n+1))) **//if you didn't delay it, it will run into an infinite loop**
//seq.delay package whatever it is in the bracket, and make it a thunk
//fun () : the type of () is unit
if you do numsFrom 1729, it will start from 1729-> code
which is 1729 1730 1731 1732…etc.

COMP 302
2016.04.04
topic: stream (examable)
let idWithPrint n = (printfn "%i" n); n
//this prints n then return n
//this seems dumb, whats the point of it?
let natWithPrint = Seq.initInfinite idWithPrint

if you do prefix 5 natWithPrint
it will print on screen
1
2
3
4
5
[1;2;3;4;5]
//if i ask again the values get recompiled and returns the same result.

let natWPCached = Seq.cache natWithPrint
so now if i do prefix 5 natWPCached
it prints the same result as last one.
but if you run the program again, it will print out
6
7
[1;2;3;4;5;6;7] because the first 5 is cached.

//some F# version of examples which have been shown in Scheme already.
//things you need to do in F# not in Scheme: wrapping it in a thunk (fun ()-> …) and delaying it

```
let rec sieve sigma =
  Seq.delay (fun () ->
                let head = first sigma
                cons head (sieve (Seq.filter (fun n -> (n % head) <>
0) (rest sigma))))
let primes = sieve (numsFrom 2)

let rec addStreams s1 s2 = Seq.delay (fun () -> cons ((first s1) +
(first s2))
                                                    (addStreams (rest
s1) (rest s2)))

let rec psums sigma =  //partial sums
  Seq.delay (fun () -> cons (first sigma) (addStreams (psums sigma)
(rest sigma)))

let ones = Seq.initInfinite (fun i -> 1)

let yaNats = psums ones //another way of writing natural number

let rec fibs = Seq.delay (fun () -> cons 1 (cons 1 (addStreams fibs
(rest fibs))))

let rec pascal = Seq.delay (fun () -> cons ones (Seq.map psums pascal))

let triangular = Seq.cache (psums nat)
```

COMP 302
2016.04.04
topic: stream (examable)

```
//this gives you horizontal row
//a steam of streams that produces a list of structures
let showrow n ss =
  let rec helper n ss acc =
    if n = 0 then acc
    else
       helper (n - 1) (rest ss) ((Seq.nth (n - 1) (first ss))::acc)
  helper n ss []
```

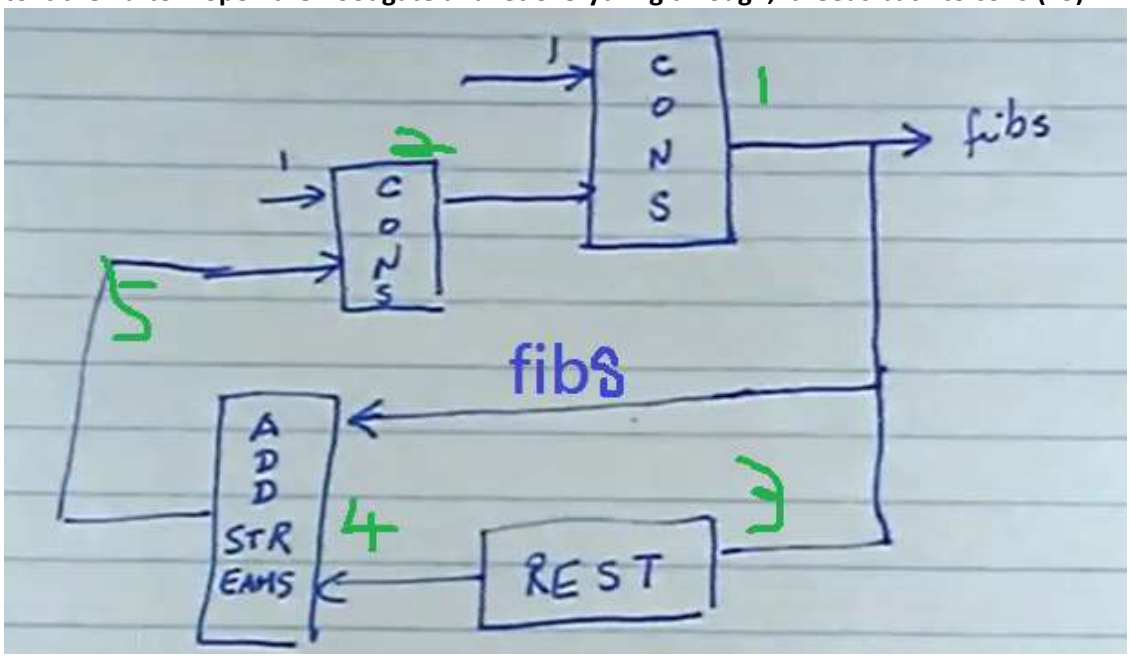
---

how to think about streams?

lets look at rhe fibs function

```
let rec fibs = Seq.delay (fun () -> cons 1 (cons 1 (addStreams fibs
(rest fibs))))
```

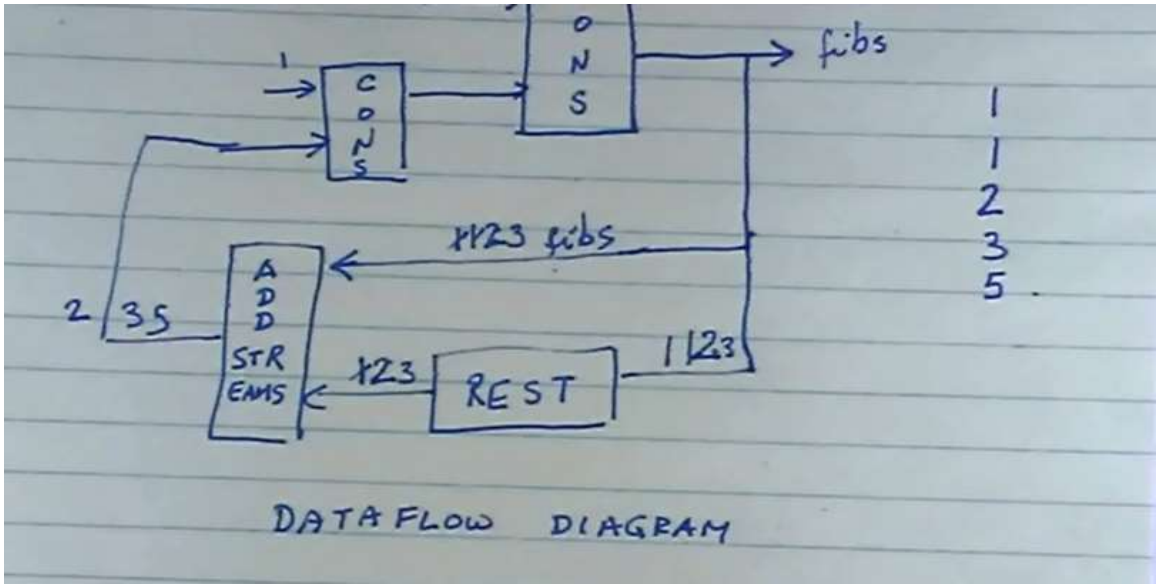
here is cons consing 1
then another cons for 1
and then addStreams of fibs(this comes out) and the rest of fibs
what rest is doing is, he says there is a stream coming in, i grab the first element and hold on to it then after i open the floodgate and let everything through, it feeds back to cons.(#5)



this is a dataflow diagram that shows you data flowing through the network.

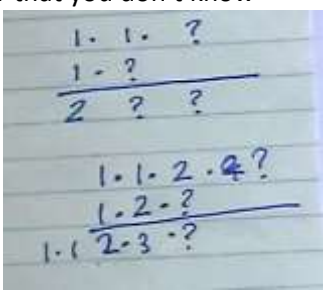
so at first it is
1
1
because of the 2 cons
fibs will have 11
and rest will have 11 at #3
REST grab the first element and hold on to it and let the rest through
1 on fib add 1 at rest gives your 2

COMP 302
2016.04.04
topic: stream (examable)
now you have
1
1
2
then 2 is pass into fibs, fibs now have 12
and rest has 12 as well, 1 gets grabbed and 2 passes through
1+2 = 3
now you have
1
1
2
3
then another cycle, you have
1
1
2
3
5



the interesting thing about this code is
the recursion becomes feedback.
this is a dataflow computation unlike what we are used to writing in controlflow computation.

another way to compute fibs
fibs0 = ?
fibs1 = 1•? //it has a one coming out and after that you don't know
fibs2=1•1•?
fibs3=1•1•2•?
fib4=1•1•2•3•?
fib = 1•1•addstream(rest(fib), fib)
fib_{n+1}=1•1•addstream(rest(fib_n), fib_n)



COMP 302
2016.04.04
topic: stream (examable)
there's a facinating subject here:
fixed point theory
you are looking for the fixed point of the equation fib = 1•1•addstream(rest(fib), fib)

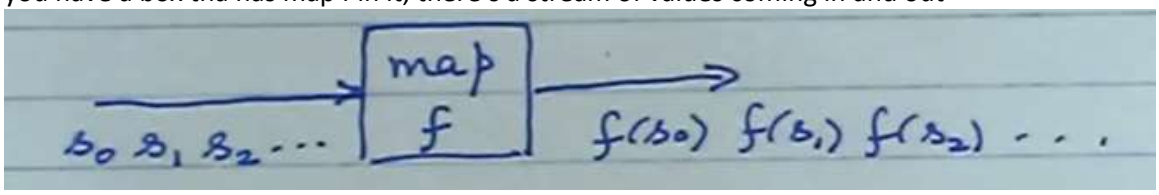
fib = F(fib)
fib is a fixed point of F
this concept is fundamental.

Suppose I am studying calculus, f:R->R
f(x)=x x is a fixed point of f.

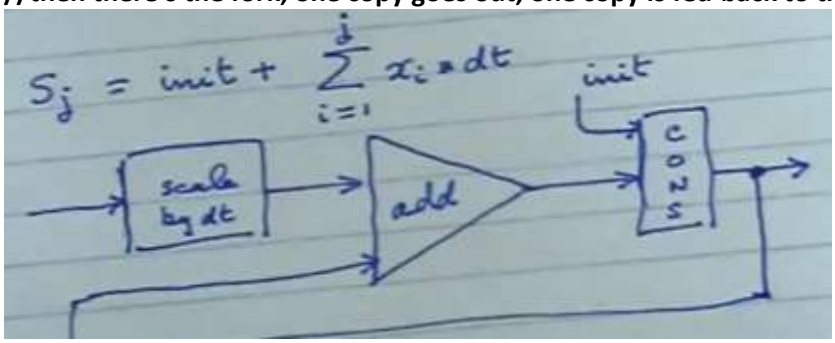
COMP 302
2016.04.06
topic: Dataflow model, stream signal paradim
In Scheme -> cons-stream delays its 2nd argument

F# -> eager need to supply function
(therefore you need to Seq.delay(fun () ->), wrapped it in a thunk)

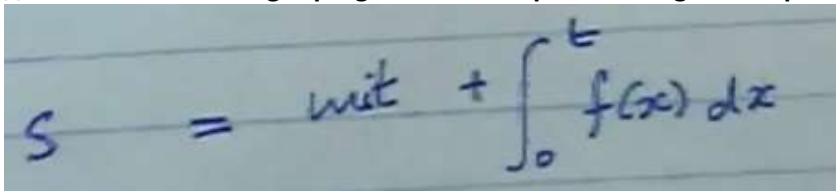
Dataflow model – invented by Jack Dennis in MIT
the idea here is think of **<u>streams as signals</u>**
//remember functional programming and streams
//it is the esscen of this course
you have a box tha has map f in it, there's a stream of values coming in and out



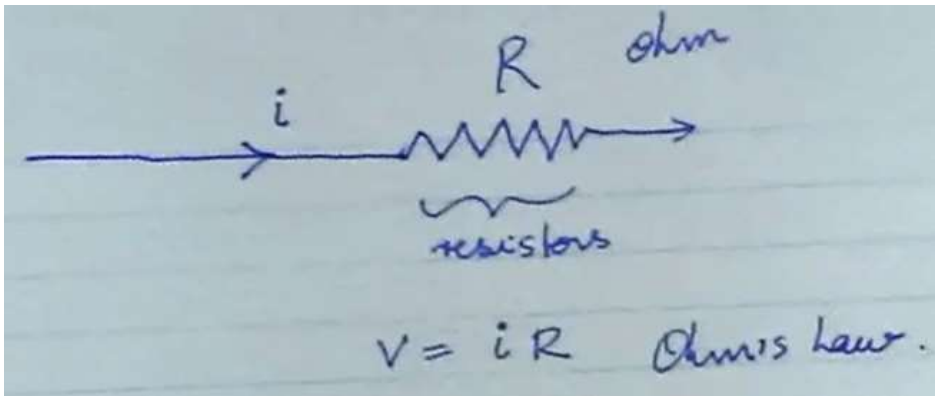
define terms:
x_i is input stream
dt -> fixed small real number
s_j = init + summation i going from i to j (X_i*dt)
//the graph below is a program that actually runs
//you have a box that has (scale by dt) in it
//the scale looks at a incoming stream of floating point values and multiply it by dt
//then theres a stream adder that takes 2 stream of values coming in and grab the first of each stream and output
//then there's the cons function we learned from the last lecture
//cons takes some value and stick it in front and then put it out
//then there's the fork, one copy goes out, one copy is fed back to the adder



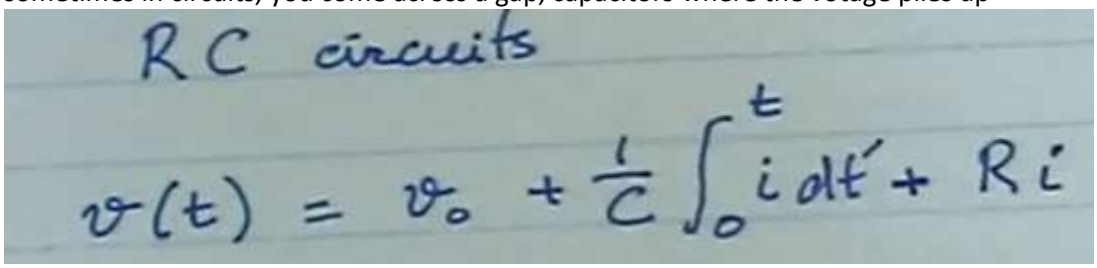
//now we have a integralprogram that computes the signal and produces an output signal



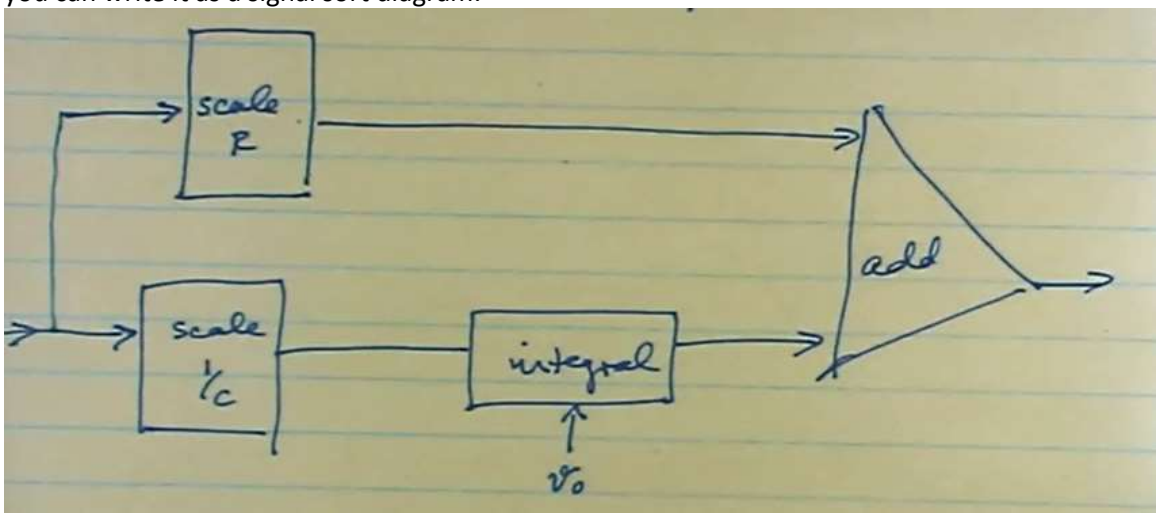
COMP 302
2016.04.06
topic: Dataflow model, stream signal paradim
Lets talk about circuits:
Ohm's law: V=iR (the votage across the resistor = current * resistance)
unit of R: ohm



sometimes in circuits, you come across a gap, capacitors where the votage piles up



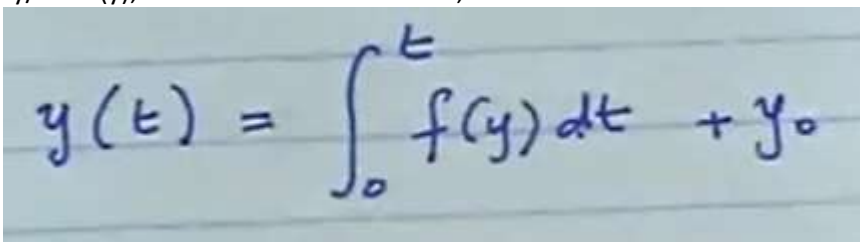
now if you look at the equation, there's a integral in it,
you can write it as a signal sort diagram:



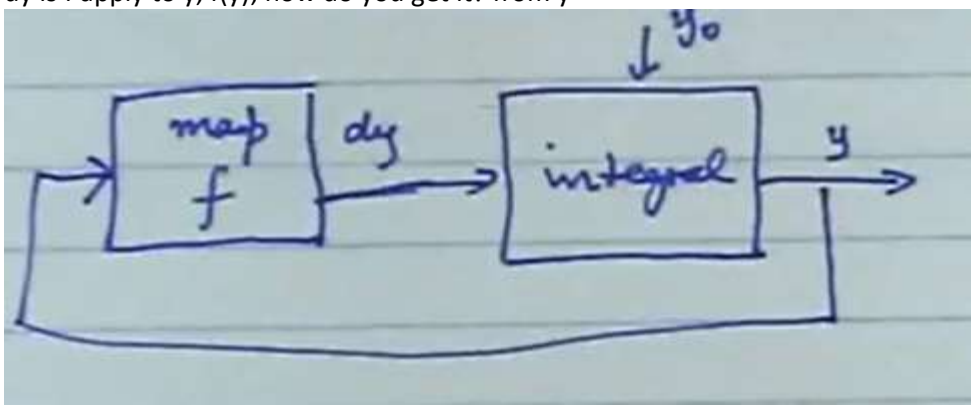
the program of this diagram looks like:

```
let RCmodel (R: float) (C: float) (dt: float) (inputSig : seq<float>)
(v0:float) =
  Seq.delay (fun () ->
       (addFloatStreams (scale inputSig R)
                        (integral (scale inputSig (1.0 / C)) v0 0.1)))
```

COMP 302
2016.04.06
topic: Dataflow model, stream signal paradim
y(t)
dy/dt = f(y), if f is a nice linear function, then this is a linear differentiation equation.



can be translated into the signal flow diagram
dy is f apply to y, f(y), how do you get it? from y



this is how you do differentiation :D
this particular program needs extra delays to make this works but that's not the point here
//the code for this progrm

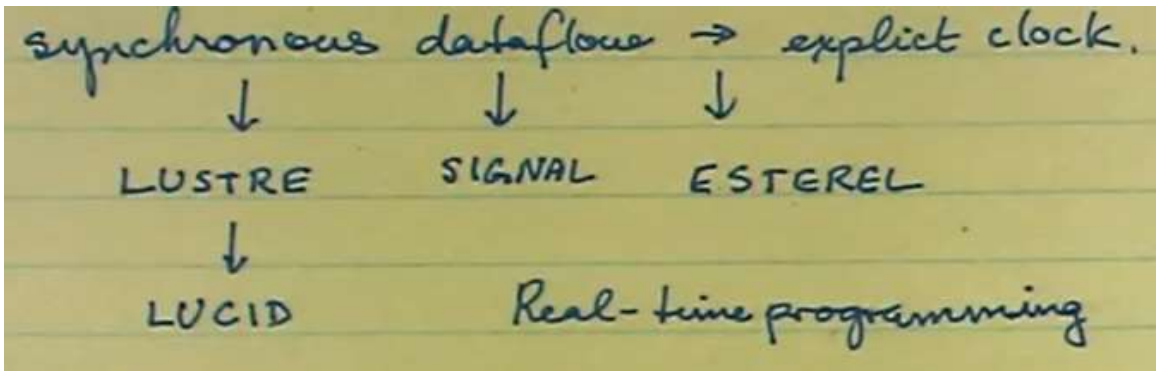
```
(define (integral2 delayed-integrand initial-value dt)
  (define int
    (cons-stream initial-value
                 (let ((integrand (force delayed-integrand)))
                   (add-streams (scale-stream dt integrand)
                                int))))
  int)
```

//the idea is essentially this kind of things can be done, and if you graph this you can see it is not difficult to do 2nd order, 3rd order functions.

---

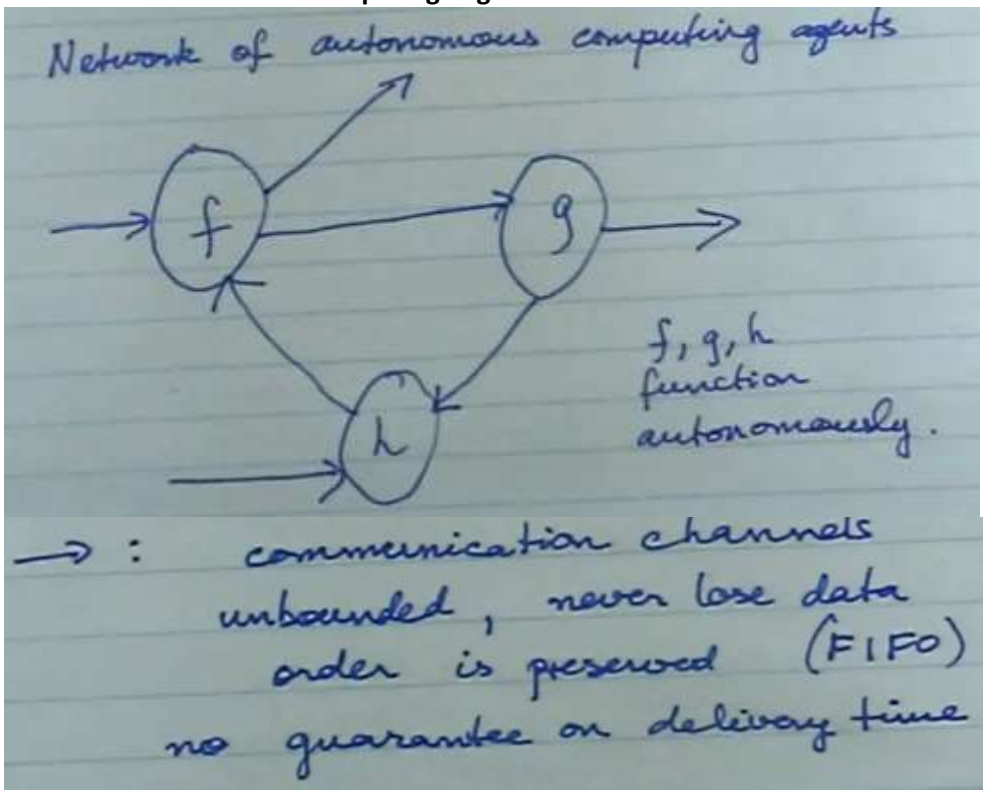
how does one think of this kind of dataflow programming?

1. **synchroness dataflow**-> explicitly clock. The way you write your code is you are concious of the beating of the clock, so the way of the code is written is it is clocked. **this is a powerful programing paradigm, they developed other languages.**

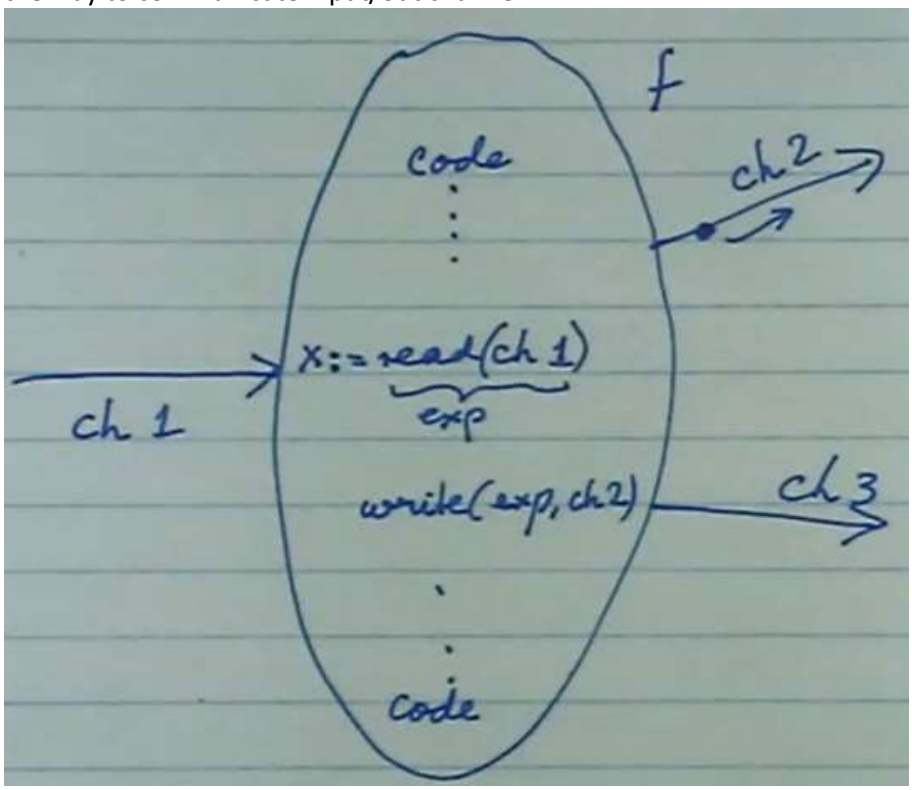


COMP 302
2016.04.06
topic: Dataflow model, stream signal paradim

2. **asynchroness dataflow:** no clock-> no global clock
   Network of autonomous computing angents



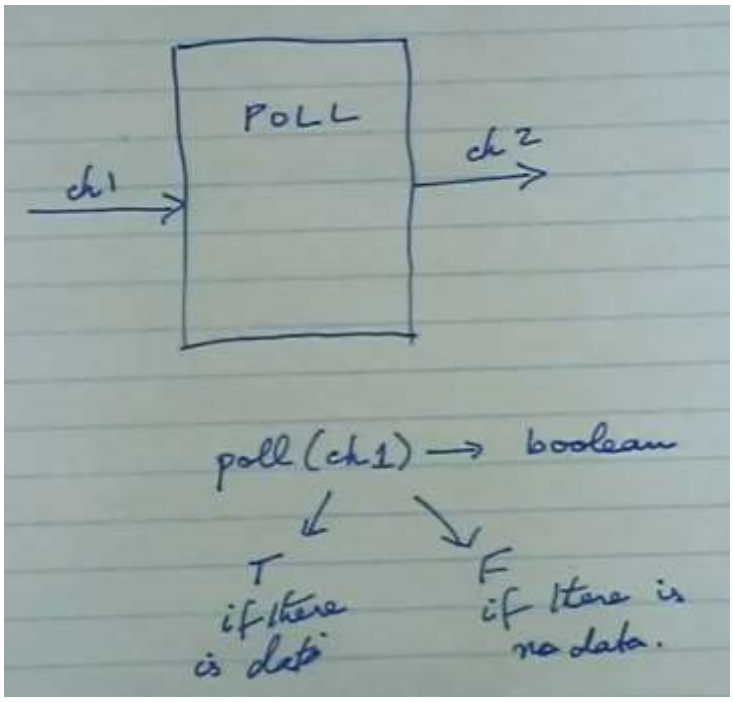
the way to communicate input/out channel



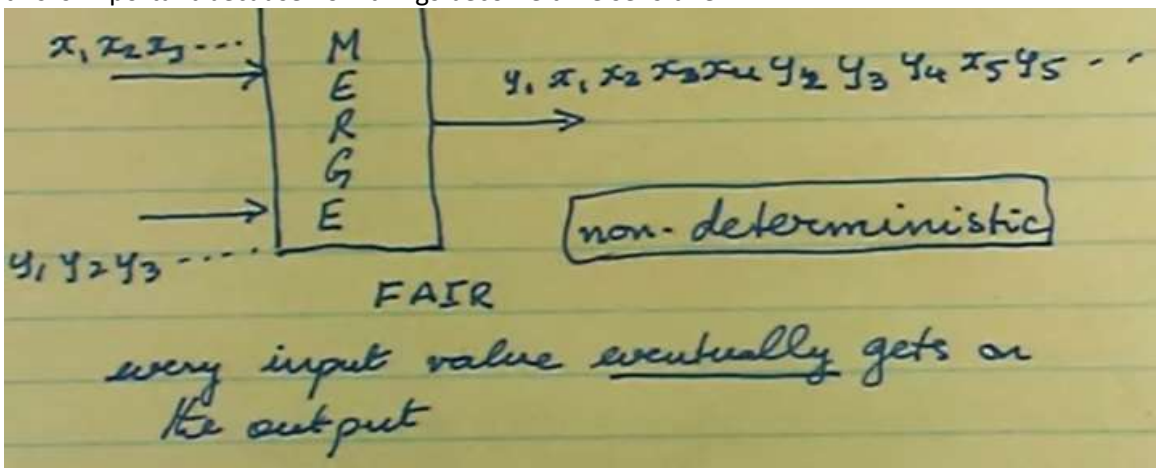
output channel will say: while(exp, ch2)

COMP 302
2016.04.06
topic: Dataflow model, stream signal paradim
What if there is no data on channel 1 when you execute read(ch1)?
the system should suspend until data become available (**aka BLOCKING READ**)

now this may seem restricted, and indeed it is:
proof: the nodes of the network compute stream functions.
These functions are “continuous”.



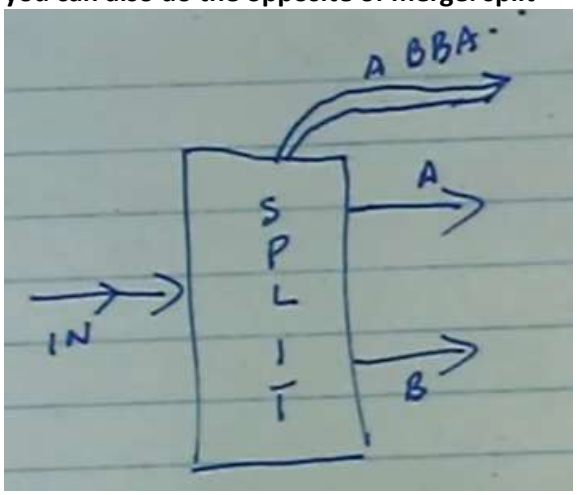
this is important because now things become time sensitive



with poll one can implement fair merge.

the reason people fail to poll fair merge is because it is provably impossible to have fair merge without polling.

COMP 302
2016.04.06
topic: Dataflow model, stream signal paradim
you can also do the opposite of merge: split



thesis: there is no stream primitive more powerful than fair merge.

Continuations

by Fernando Serboncini (fserb@fserb.com)

We are going to talk about **Continuations**.

To understand what continuations are we need to have in mind two things: closures and the control flow of our program. If we get this, continuations will be easy.

We are going to talk a bit about control flow, how functions (and closures) are also control flow structures and then we are going to talk about what *continuations* are and how we can use them.

Finally, I'll try my best to give real life examples on where continuations can be used. So, if at some point early on you have the urge to say "but can't this be done easier with X?", the answer is probably yes. But try to focus on the techniques. And eventually we will get to more interesting examples.

go with the control flow

Control flow is the order in which the statements of your program are executed. Actually, it's about *controlling* the order in which the statements of your program are executed.

Let's think a bit about which control flow mechanisms we know and what they actually do.

At the most basic, the CPU has a control flow policy: every time it reads an instruction to execute, it needs to decide what is the next instruction it's going to execute (which usually is "the next instruction in memory"). This notion becomes so natural that most (but not all) programming languages have some notion of this idea:

Most language structures are control flow structures: if, while, switch, exceptions, for loops. Since for-loops are forbidden in this class, let's see a for loop example:

```
let forbidden a b =
  let mutable sum = 0
  for i = a to b do
    sum <- sum + i
  sum

forbidden 3 9 // ==> 42
```

for loops are weird in functional languages. Nobody does it like that. How would we usually do something like this in F#?

```
let rec myFold f v s = // already in the standard library.
  match s with
    | [] -> v
    | (x::xs) -> f x (myFold f v xs)

let answer a b = myFold (+) 0 [a..b]

answer 3 9 // ==> 42
```

`fold` is a function that reduces a list to a single element. The interesting point here is that we are replacing the *for* loop with a recursive call to `fold`. This works because function calls are also a control flow structure. Let's talk a bit about that.

functions, closures and the pursuit of happiness

What do we know about functions?

We know that in Functional languages they are *first class citizens*. We know that they have arguments and a body. Sometimes they even have names:

```
let awesomestFunction f x = f (f x)

awesomestFunction (fun x -> x * x) 3 // ==> 81
```

We know more things about functions. They can be associated with an environment in a closure:

```
let creator x =
  let creature y = x * y
  creature

let c1 = creator 1
let c2 = creator 2

c1 3 // ==> 3
c2 3 // ==> 6
```

As you can see here `creature` is a function. But every time you call `creator` you get a different closure (on the same function).

There's one more thing: functions can be called and return values and that's what makes them a control flow structure: when they are called, they change the execution of your program to the function and after they are done, they change it back to where the function was originally called.

Let's quickly think about that. When a function is called, it's pretty trivial where the execution has to go to. But how do functions know where to return to?

The way functions know where to return to is by using *The Stack*. *The Stack* is a stack of "execution frames" or "stack frames". Those execution frames contains all the runtime information needed to call a function (like arguments) and for the function to return to its callee.

Different languages have different ways of building this execution frame (and some don't even have that).

Usually when a function in a Stack-based language gets called, here's what happens: the callee has to *push* an execution frame on The Stack that contains the place where the function has to return to, and then it changes execution to the function. The function, on the other hand, when it's time for it to return, assumes there's a properly set up execution frame on The Stack, *pops* it up, and return to the place where that frame points to.

The important bit here is: each stack execution frame must be preserved while the called function is still running so the function knows where to go back to once it's done.

Let's see an example:

```
let divide_failwith x y =
  if y = 0 then failwith "Divisor cannot be zero."
  else
    x / y

let do_something a b =
  if (a = 0) then 0 else (divide_failwith a b)

do_something 4 2  // ==> 2
do_something 4 0  // ==> ??
```

When this code executes, it adds a frame stack to call `do_something` and another one when `do_something` calls `divide_failwith`. When this function ends, it pops up the stack, and then goes back to right after `divide_failwith` was called. The next thing is the end of `do_something`, so it pops another stack and then returns to the `main` of our program.

When `failwith` is called, it raises an exception, which ends up showing:

```
System.Exception: Divisor cannot be zero.
  at divide_failwith (Int32 x, Int32 y)
  at do_something (Int32 a, Int32 b)
  at main@ ()
Stopped due to error
```

What F# does when it realizes there was an uncaught exception is to *"dump the stack"*, i.e., it goes through each frame in The Stack and prints which function it belongs to. The objective is to try to help you debug the error. It can do that since this information is already available, because it's part of the way the function calling control flow is done.

So The Stack is great, right? It gives us the proper function calling and free debugging info. Awesome. So what's the problem with The Stack?

The problem with The Stack is that it has limited space available. Each stack frame takes space which means there's a limit on how many function you can call inside functions.

In functional languages, where recursion is often used, it becomes a big deal. All of the sudden there's a hard limit on how much recursion you can do, which is not great.

A solution to that is the so called *tail call elimination*. It works by the compiler realize that, if a function always ends by calling some other function it doesn't need to create a stack frame for it. Instead it can just jump to the new function and the stack frame used for the initial function will just be reused by the new one.

Continuations through the looking glass

So, now we know a few things: that all languages provide ways for us to change the control flow of our program (that's what makes computer programs interesting); and that functions are control flow mechanisms that use a stack to jump around.

We are going to see a new control flow mechanism. It's called **Continuations**.

Like functions, *continuations* change the order things get executed on a program and can be on a closure. But unlike functions, they don't depend on the growth of the stack. They allow you to build a program where the control flow can be directly manipulated.

So, what continuations look like and how can we use them? *I'm glad you asked this, Emma*. Fortunately for us, all languages that have first class functions and tail-call elimination already allow for *continuations*. In those languages, like F#, continuations are simply closured functions that are not expected to return.

Think about it: if a function is not supposed to return, then there's no need for a stack (Which tail-call elimination will realize and get rid of). So we get the benefits of a function without the downside.

But if that was all, it would be a bit dull, no? A function that just doesn't return it's like a dead end. And also, doesn't a function *always* return when it finishes?

You are right, Emma. The missing part is: instead of returning, continuations are expected to call other functions when they are done.

Let's see some code:

```
1   let print_number x =
2     printf "%.0f\n" x
3     System.Environment.Exit 0
4
5   let add_and_print x =
6     let y = 17.0 + x
7     print_number y
8
9   let square_add_and_print x =
10    let y = x * x
11    add_and_print y
12
13  square_add_and_print 5.0  // prints 42 and exits.
```

First, notice that all 3 functions `print_number`, `add17_and_print` and

`square_add_and_print` are **continuations**: they are functions with closures that are not expected to return and instead call other continuations to continue the flow of the program. Interestingly, `Exit` is also a function that is not expected to return.

But who would program like this? Well, for something simple, nobody. *That's why it's called an example, Greg*.

How can we make this more interesting? Well, we realized that continuations must call other continuations when they are done, right? What if, instead of hard coding the next continuation (as we did on the previous examples), we pass it as a parameter?

```
1  let print_number x =
2    printf "%.0f\n" x
3    System.Environment.Exit 0
4
5  let calculate x cont =
6    let y = x * x + 17.0
7    cont y
8
9  calculate 5.0 print_number  // prints 42 and exits.
```

Now we are getting somewhere! We have two functions that each do something meaningful and we connect them by telling the first function: "when you are done, this is where you are supposed to go". We even got to reuse `print_number` from the previous example.

Look at what we just did. We were able to construct a ~~function~~ *continuation* `calculate` that has a semantic of *"give me a value, I'll do some computation with it and when I'm done I'll send it to the function you pass me"*.

This is normally called *"Continuation Passing Style"* or *CPS*. I.e., we are programming in a way that we always pass the continuations for our program. We are going to focus on this for most of this class, and later I'll get back to continuations in general.

OMGBBQ Continuations

Ok. Let's do one more example. Imagine we want to write a simple login scheme using CPS: ask for a username, password, check if everything is ok and login the user.

```fsharp
// Ends everything.
let finish () =
  System.Environment.Exit 0

// Checks that a @pass is valid for a @user and continues.
let check_password user pass cont =
  printfn "Password is OK."
  cont user

// Do login input, checks for password and continues.
let do_login cont =
  printf "login: "
  let u = System.Console.ReadLine()
  printf "password: "
  let p = System.Console.ReadLine()
  check_password u p cont

// Greets @user and finishes.
let greet_user user =
  printfn "Hey, %s, welcome!" user
  finish ()

do_login greet_user
// login: drumpf
// password: 1729
// Password is OK.
// Hey, drumpf, welcome!
```

Pay a bit of attention to `do_login`: it's a *continuation*, i.e., a function that never returns. But the caller of `do_login` can still control what happens after the login, by telling it to call `greet_user` when everything is finished.

Also, it's important that you think of each of the functions in isolation when we are thinking in terms of continuations. In a way, it's very similar to how we think about recursion (if we don't want to be confused by it). Each function has a clear mandate and responsibility: it gets some params, does what it needs to do and calls the continuation it promised to call. It's almost an afterthought that everything works together as we want, as long as each block does what it said it was going to do.

Moving on, our `check_password` is pretty lame, it just continues. Imagine that we want to do some real work with it. In that case, maybe having that single continuation is kinda silly, since you probably don't want to do the same thing when the login fails or succeeds.

How could we solve this? Well, since we are passing the continuation as a parameter, maybe we can pass two continuations: one for when the password checks, and one for when it doesn't.

```
1   // Checks that a @pass is valid for a @user.
2   // If it is @valid_cont, else @fail_cont.
3   let check_password user pass valid_cont fail_cont =
4     if pass = "1729" then
5       printfn "Password is OK."
6       valid_cont user
7     else
8       printfn "Wrong password."
9       fail_cont ()
10
11  // Do login input, checks for password and continues.
12  let rec do_login login_cont =
13    printf "login: "
14    let u = System.Console.ReadLine()
15    printf "password: "
16    let p = System.Console.ReadLine()
17    let when_fail = fun () -> do_login login_cont
18    check_password u p login_cont when_fail
19
20  // Greets @user and finishes.
21  let greet_user user =
22    printfn "Hey, %s, welcome!" user
23    finish ()
24
25  do_login greet_user
26  // login: drumpf
27  // password: 1
28  // Wrong password.
29  // login: drumpf
30  // password: 1729
31  // Password is OK.
32  // Hey, drumpf, welcome!
```

Let's look at `check_password`. It continues to `valid_cont` if the password is valid, or to `fail_cont` if it isn't. It's a simple *if* statement, but there's something subtle going on here: we are using our continuations as mechanisms to change the flow of the program.

The tricky part here is on line 17, when we define the `when_fail` continuation. Let's think this from `do_login`'s perspective: it wants to call `check_password` and move forward with the process (with `login_cont`). But now it has to deal with the password being wrong. What does it want? One reasonable option is: if the password check fails, let's try the login again. How do we do this? We pass it a continuation that calls `do_login` again. But what is the continuation for this new `do_login` call? Well, the same as before, `login_cont`.

Don't just "I get it" this. Think a little bit further. We are constructing a new continuation on the fly to do what we want. In this case, it's nothing super interesting: just doing the same thing the caller did when calling `do_login`. But we are still building the sequence of code that will happen "on the fly".

We are getting good at this

Ok, let's see another example. Imagine we want to write a function that sums all numbers from 0 to n.

```
let rec stack_sum n =
  if n = 0 then 0
           else n + stack_sum (n - 1)
```

This is not a tail-call. Can you see why? The last function called is not `stack_sum`, but `+` (plus)! What happens when we run this?

```
stack_sum 1000     // ==> 500500
stack_sum 1000000  // ==>
// Process is terminated due to StackOverflowException.
```

But remember how we realized that all continuations are tail-call optimizable and therefore can't stack overflow? So if we rewrite our function using continuations, we won't have this problem. Let's try:

```
let rec cont_sum n cont =
  if n = 0 then cont 0
           else cont_sum (n - 1) (fun x -> cont (x + n))

cont_sum 1000 print_number   // prints 500500
```

Let's read this slowly. First, what is our function signature? `cont_sum` receives a `n` that we wants to count up to (from 0 to n) and a continuation `cont` to be called with the result. This continuation could be `print_number` like we had before.

The `then` statement is pretty obvious, right? If we have to sum to zero, we call our continuation with the result 0.

The `else` statement is the good one. We are defining `cont_sum n` as being equivalent to `cont_sum (n-1)` with a different continuation. What is this continuation? It's the previous continuation wrapped with `+n`.

Let's say this in another way: `cont_sum` keeps building up a continuation: `the original one + n + n-1 + n-2 + ...` until we get to the `0` case, when we call that bigger continuation with `0`, which then calculates all the sum and call the original continuation.

This is an important step and just like last time, let's recap what we are doing: we are also creating a continuation on the fly, but this continuation is not as dull as last time. It's doing the actual heavy work of our function. The recursion is used to actually build a continuation, that eventually gets executed all in one shot.

Our new function doesn't stack overflow, so:

```
1   cont_sum 1000000 print_number  // prints 500000500000
```

oh yeah.

One very quick warning to people doing F# on Mac and Linux. On those platforms, we run our F# code using Mono. Unfortunately, there is an open bug on Mono, because it does not support tail calls at all (i.e. tail calls still increase the stack). You can see the tail call being optimized if you check the compiled code (with the `monodis` tool), but the example above still fails.

Another extra note: sometimes people rename the continuation variable `return`, so the code looks pretty much like a regular function where you `return` the value. It may make for a good gimmicky if you are really confused. But it's very misleading, since the thing that's usually associated with `return` (i.e., the actually go back to where you came from) is definitely not happening.

Exercise: can you implement factorial and fibonacci using continuations?

Say continuation one more time, I dare you

Let's keep working on examples of using CPS to solve interesting problems. Let's imagine we have a binary tree structure:

```
type 'a tree =
  | Empty
  | Branch of  'a * 'a tree * 'a tree

//     a
//    / \
//   b   c
//  / \   \
// d  e    f
//        /
//       g
let tree1 = Branch ('a',
                Branch ('b',
                    Branch ('d', Empty, Empty),
                    Branch ('e', Empty, Empty)),
                Branch ('c',
                    Empty,
                    Branch ('f',
                        Branch ('g', Empty, Empty),
                        Empty)))
```

Now let's write a function that returns true if an element is on this tree, using a standard recursive approach:

```
let rec stack_find el t =
  match t with
    | Empty -> false
    | Branch (x, l, r) ->
      if x = el
        then true
        else if stack_find el l
               then true
               else stack_find el r

stack_find 'g' tree1  // ==> true
stack_find 'h' tree1  // ==> false
```

We recursively look at the current element, the left branch and then the right branch, returning if we ever find the element. But look at what happens at runtime:

```
stack_find 'g' tree1 =
  find (A)
    -> find (C)
      -> find (F)
        -> find (G)
          -> true

```

We keep recursing down the tree and building the stack. As soon as we find the answer, we return `true`. But this `true` has to propagate back all the way up the stack. Each function

returns `true`, then goes back to its callee, that then returns `true`, etc, etc…

Gladly, we know how to do better now. Ideally we want to return as soon as we found something, and not worry about whatever happened before. Let's re-implement this using continuations. Instead of returning `true` or `false` we can have a continuation for success (the element exists) and another one for fail (the element doesn't exist):

```
1   let rec cont_find el t succ fail =
2     match t with
3       | Empty -> fail ()
4       | Branch (x, l, r) ->
5         if x = el
6           then succ ()
7           else cont_find el l succ (fun () -> cont_find el r succ fail)
8
9   let yes () = printfn "yes"
10  let no () = printfn "no"
11
12  cont_find 'g' tree1 yes no  // prints yes
13  cont_find 'h' tree1 yes no  // prints no
```

Wow. Just wow. Isn't this beautiful?

Ok, let's go over what's happening. First, let's check our function signature. `cont_find` receives 4 parameters: `el` and `t` are the same as before (the element we are looking for and the tree we are looking into). The other two are continuations: `succ` is supposed to be called when we find the element, `fail` when we don't.

So what does the function do? If we are in an empty branch, there's obviously nowhere else to go, so we continue to `fail` (i.e., we didn't find the element). If we are in a branch and we have the node we were looking for, we call `succ` (i.e., we found the element). Fair.

Now what happens when we don't have the element and we have the left and right branches to check: we call `cont_find` on the left branch. What are the continuations for this call? If we succeed we just want to call the `succ` we had before. But if we fail, we still have to check the right branch. So we do this. And what are the continuations for this new `fail`? Well, for success, is always the same. And there's nothing else we want to do, so failing here is the same as failing the whole thing, so we just call the original `fail`.

Another way of putting this: we first check the left side of our tree, but as we go down we build a bigger `fail` continuation. this fail continuation contains the find for the right side of the tree. We are saying: Try to find the element here, but if you fail you still need to look on this other side before really giving up (i.e., before calling the `fail` the original callee gave us).

Exercise: can you change `cont_find` to use a single continuation that receives a boolean

telling if the element was found or not?

I like continuations so much I want another example

TK maybe this should be another exercise?

One last example. Let's write a function `cut` that given a list returns the sublist up to the point where a passed function is true. For example:

```
let isEven x = (x % 2) = 0

cut [ 2 ; 4 ; 6 ; 5 ; 2 ] isEven  // ==> [ 2 ; 4 ; 6 ]
cut [ 1 ; 3 ; 5 ] isEven          // ==> [ ]
```

notice that `cut` is not CPS. But we are going to use CPS inside it to solve the problem.

```
let cut list pred =
  // we first create our internal CPS function.
  let rec cc list pred cont =
    match list with
      | [] -> cont []
      | x::xs ->
        if (pred x)
          then cc xs pred (fun a -> cont (x :: a))
          else cont []
  // then we call it. Notice that our continuation is
  // the identity function:
  // let id = fun x -> x
  cc list pred id
```

It's similar to our previous examples: we change the continuation for the recursive call to be the original one wrapped with `x ::`. When we finally get a value that doesn't pass `pred`, we immediately continue.

(I can't get no) exception

We are going to talk now about 2 common uses of continuations *"in the wild"*. The first one is how to do exceptions with continuations. It's actually pretty similar to the list find example we just saw.

If we think about how exceptions work (they are also a control flow structure), we have the

following:

1. a statement can thrown an exception.
2. after each statement there are two possible paths our code can take: continue with the normal flow of the program, or go to the exception flow.
3. the exception flow can also go both ways: return to the normal flow (if it caught that particular exception) or stop everything we've been doing (since the exception hasn't been caught).

Too abstract? Let's see what an exception looks like:

```
let ret =
  try
    failwith "fail"
    "hello"
  with
    | Failure msg -> "caught: " + msg

ret  // ==> "caught: fail"
```

So here `ret` becomes "caught: fail", since `failwith` is caught by our `try/with` statement. Interestingly, in many languages exceptions are actually implemented as continuations. The reason being that exceptions can have extremely complicated interactions with other control flow structures - think of a `break` inside of a `for` inside of an exception handler. Therefore it's not uncommon for compilers to transform your code into a CPS-style code during compilation to solve this (we will see something like this later on).

So, let's do the other way around. Let's think how this would be implemented using continuations. Well, lucky for us, our previous example looks very similar to what we want: we want to write some code that represents the "normal" flow of our program, and some code for when something goes wrong. On our `cont_find` function, we called it `succ` and `fail` and we didn't think of them as actually something going wrong. It was just two options on our code. We can instead phrase them as a regular continuation and an error one.

Remember the `divide_failwith` example we had before:

```
let divide_failwith x y =
  if y = 0 then failwith "Divisor cannot be zero."
  else
    x / y

divide_failwith 4 2  // ==> 2
```

Rewriting it with CPS replacing the exception would look like:

```
let divide_failwith x y cont error =
  if y = 0 then error "Divisor cannot be zero."
  else
    cont (x / y)
```

It's that simple. To call this, we need some code to run as an exception handler. We could have a `default_error` continuation to use everywhere:

```
let default_error msg =
  printfn "Uncaught exception: %s" msg
  System.Environment.Exit 1
```

And then we can use:

```
divide_failwith 4.0 2.0 print_number default_error  // 2.0
divide_failwith 4.0 0.0 print_number default_error
    // Uncaught exception: Divisor cannot be zero.
```

We can also implement `finally`, by simply extending both continuations with our code. Imagine we want to call a function `close_database` in our `finally` statement (i.e., we want that function to be called on all possible outcomes of our code):

```
// we must call close_database. So instead of:
// divide_failwith x y print_number default_error
// we have:
let cont = fun x ->
  close_database ()
  print_number x
let fail = fun x ->
  close_database ()
  default_error x

divide_failwith 4.0 2.0 cont fail
```

In this case `close_database` behaves as a `finally` statement, i.e., it gets called when `divide_failwith` returns, no matter if an error happened or not.

We could go further with our exception mechanism and, for example, pass the `cont` continuation to the exception one, allowing it to recover from an error. But the important thing is the principle of what we are doing: making the control flow of our program explicit and implementing features that may not even be directly available in our code's language (as long as it supports continuations).

So call(back) me maybe

Our second generic application of continuations are callbacks. Callback are interesting because they are probably the simplest form of continuations and yet they are so useful that they are used everywhere in modern "real life" programming.

First, what are callbacks? Callbacks are continuations that we pass in otherwise non-continuation style programs to continue a computation after a certain event has occurred.

For example, imagine that we want to load a file from a URL on the Web. It may take quite a while for the whole file to be downloaded (and it may even fail). So we don't want our program to hang there and wait for the file to complete. In this case, we want an API that allows us to say: here's a URL I want, and please call this continuation when you are done downloading it and ready for me to use the file. In theory, it's identical to the type of code we've been writing here. In practice, the only difference is that our continuation will be called sometime in the future (not right away).

For illustration purposes, here is actual code from Google Chrome:

```
1  void OnThumbnailAvailable(RequestContext* context,
2                            const GURL& url,
3                            const SkBitmap* bitmap) {
4    if (bitmap) {
5      // ... use bitmap as thumbnail of the page.
6    }
7  }
8  thumbnail_manager_->GetImageForURL(url, &OnThumbnailAvailable);
```

We can see that `thumbnail_manager_` provides an API that gets an URL and a continuation for when the image has been loaded. In this case `OnThumbnailAvailable` is our callback (continuation).

Callbacks are also extremely popular in JavaScript (that runs on all web pages), since most events (a mouse click, a keyboard press, a form submit) are handled this way. So it's super common in Javascript to see code like:

```
1  function onKeyPress(event) {
2    // some key has just been pressed.
3  }
4  document.addEventListener("keypress", onKeyPress);
```

Again, what the API asks with the callback is a continuation (`onKeyPress` in this case): when

this event happen (a key press), stop whatever you are doing and continue the program execution there.

Notice that I'm not showing you any code in F# with callbacks. The reason is: callbacks are such a simpler use case of continuations, that all our previous examples are much more interesting than those. Still, I want you to remember that we are doing an advanced version of something that is used everywhere on day-to-day programming.

CPS Transformers: the compiler

Remember when we talked about how exceptions can (and often are) written with continuations? It so happens that a lot of functional languages (Haskell, for example) have an intermediate representation in Continuation-Passing Style during compilation (as opposed to static single assignment style - SSA - for imperative languages).

Luckily, it is relatively simple to automatically (or manually, for that matter) transform any non-CPS code into CPS code. I won't go into compiler theory here, but I want us to have a feeling of how this work. Imagine those 4 bindings:

```
let a = 4
let b = 7
let f x = x * x
let g x y = x + y

f (g a b) ==> 121
```

Let's manually convert it to CPS style:

```
let a_mc cont     = cont 4
let b_mc cont     = cont 7
let f_mc cont x   = cont (x * x)
let g_mc cont x y = cont (x + y)

// that can be called as:
b_mc (a_mc (g_mc (f_mc print_number)))  // ==> 121
```

That was very easy, right? It follows from this simple example that there must be automated ways of doing this transformation (and they are actually as simple as what we see here). We won't go into much details, because it would require us to define a language, etc, but you get the idea.

Why would a compiler do this? We've already seen the foremost benefit of automatic translating

your code to CPS-style: not having to worry about stack overflows. Considering that most functional languages depend heavily on recursion, it becomes a language feature to not have a limit on it.

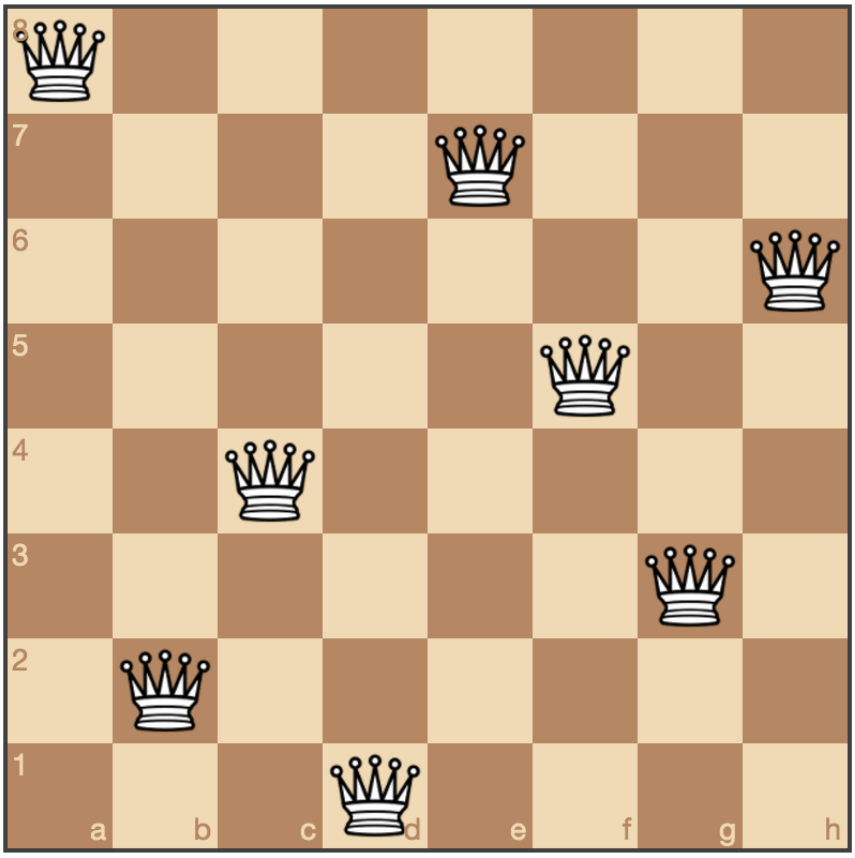
So after transforming your code into continuation-passing style, the compiler is able to tail call optimize every single function in your program. Just remember that even in those languages there can still be positive performance side effects of making your recursive function tail-call. For example, rewriting things with accumulators will reduce the runtime of some algorithms even after tail call.

Imma let you finish, but…

Ok. Now that we are continuations rock stars, we can tackle a real problem.

A class of very well known problems (that you will for sure see again in the future) are the so called backtracking algorithms. Those are algorithms where you have to search a space of solutions by making guesses and being able to backtrack on those guesses once you reach a dead end. Ideally, you are able to backtrack way before you made all decisions you have to make (therefore pruning the search space).

The most well known backtracking problem is the n-queens problems. It goes like this: imagine a `n` by `n` chess board. Find a set up where you put `n` queens on the board that are not threatening each other (i.e., does not share the same row, column or diagonal). This is a solution for `n=8`:



One could think of an algorithm where you generate all possibilities of a board with n queens and then simply check if the condition is satisfied for each one of those arrangements. Unfortunately, this would take a really long time (there are `4,426,165,368` possible arrangements and only `92` solutions for `n=8`). That's why backtracking is considered a good solution for n-queens: it allows us to skip most nonsense arrangements very fast. There are other optimizations one could do to solve this problem, but let's keep it simple.

Let's represent our board by a series of integers for each position, as if our board was linear array from `0 to n*n`. For example, for a `n=8` board:

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

Then, we are going to define a function that tells us if two queen positions are threatening each other:

```
1  // Is @p1 a threat to @p2?
2  let is_threat n p1 p2 =
3    let (x1, y1) = ( p1 % n, p1 / n)
4    let (x2, y2) = ( p2 % n, p2 / n)
5    x1 = x2 || y1 = y2 || x1 + y1 = x2 + y2 || x1 - y1 = x2 - y2
```

Next, let's define a function that tells us, given a position, would it threaten other queens we have already selected:

```
1  // Is @p in conflict with any of positions on @queens?
2  let rec is_conflict n p queens =
3    match queens with
4      | [] -> false
5      | x::xs -> if is_threat n x p then true else is_conflict n p xs
```

Ok. Now we have all domain-specific (n-queens-specific) functions we need. Now let's do the function that will search with backtrack until it gets all solutions using CPS.

This function will get a list of position to test, a list of already selected queens and a backtrack continuation. It will try each position, making sure our constrain (`is_conflict`) is satisfied. When it selects enough queens (`n`), it will record that solution and keep looking for others. When it decides that it wants to try a queen in a particular position, it will change the backtrack continuation to say: in case you can't find a solution with this, use this continuation to keep going without the decision I just made.

```
1  let rec search n clist moves back =
2    match clist with
3      | [] -> back ()
4      | (p::ps) ->
5        let new_backtrack = fun () -> search n ps moves back
6        if is_conflict n p moves
7          then new_backtrack ()
8          else
9            let new_moves = moves @ [p]
10           if List.length new_moves = n
11             then new_moves
12             else search n ps new_moves new_backtrack
```

That's pretty much it. So let's take a closer look at the `search` function. It's trying all positions on `clist` (which is an array of positions). When there's no more positions, it backtracks (line 3). Otherwise it gets the first position and sees if it can add it to the current queen position

(`moves`) without causing conflict (line 4-6). If it can't then it skips this positions and keeps looking with the other positions (line 7).

If it can add that position, it does so (`new_moves`, line 9). If it has gathered enough queens (line 10) then it is going to return this particular selection of positions (line 11). Otherwise it has to keep looking for more queens, but it changes the backtrack continuation to backtrack the choice it just made (i.e., the addition of `p` on `moves`) (line 12).

That's it. Then we can have:

```
1   let queens n = search n [0..n*n-1] [] (fun () -> [])
```

I.e., search all moves, starting with no queens. The default backtrack continuation does nothing. Calling this returns us all solutions for the `n` queens problem. For example, the one above was generated with:

```
1   queens 8
2   // ==> [0; 12; 23; 29; 34; 46; 49; 59]
```

With a little modification, we could make it return all solutions (instead of the first). Then we could do something like:

```
1   List.map (fun n -> List.length (queens n)) [1..10]
2   // ==> [1; 0; 0; 2; 10; 4; 40; 92; 352; 724]
```

which would shows us how many solutions there are for each `n`.

Week 1	Lecture 1	7 Jan	Introduction	
	Lecture 2	8 Jan	Basic F#	
Week 2	Lecture 1	11 Jan	Functions, patterns, recursion	
	Lecture 2	13 Jan	Lists	
	Lecture 3	15 Jan	Collections	
Week 3	Lecture 1	18 Jan	Trees	HW1 out
	Lecture 2	20 Jan	Operational semantics	
	Lecture 3	22 Jan	Operational semantics	
Week 4	Lecture 1	25 Jan	Inductive proofs of program behaviour	Tricia
	Lecture 2	27 Jan	Higher-order functions	
	Lecture 3	29 Jan	Higher-order functions	
Week 5	Lecture 1	1 Feb	The environment model	HW1 due, HW2 out
	Lecture 2	3 Feb	The environment model	
	Lecture 3	5 Feb	Mutable data	
Week 6	Lecture 1	8 Feb	Closures and objects	
	Lecture 2	10 Feb	The stack and the heap	
	Lecture 3	12 Feb	Imperative features	
Week 7	Lecture 1	15 Feb	Typing rules	HW2 due, HW3 out
	Lecture 2	17 Feb	Type Inference	
	Lecture 3	19 Feb	Unification	
Week 8	Lecture 1	22 Feb	Type parametricity	
	Lecture 2	24 Feb	Review	
	MID TERM	25 Feb	**MID TERM**	Evening
Week 9	Lecture 1	7 Mar	Interpreters	HW3 due, HW4 out
	Lecture 2	9 Mar	Parsers	
	Lecture 3	11 Mar	Code generation	
Week 10	Lecture 1	14 Mar	Continuations	Fernando Serboncini
	Lecture 2	16 Mar	Continuations	Fernando Serboncini
	Lecture 3	18 Mar	Continuations	David Thibodeau
Week 11	Lecture 1	21 Mar	Object-oriented type systems	HW4 due, HW5 out
	Lecture 2	23 Mar	Inheritance and Subtyping in Java	
	No Lecture	25 Mar	Good Friday	
Week 12	Lecture 1	28 Mar	Easter Monday	
	Lecture 2	30 Mar	Scheme	
	Lecture 3	1 Apr	Scheme	
Week 13	Lecture 1	4 Apr	Streams in Scheme	HW5 due, HW 6 out
	Lecture 2	6 Apr	Streams F#	
	Lecture 3	8 Apr	Reactive programming	
Week 14	Lecture 1	11 Apr	Concurrency	HW 6 due
	Lecture 2	13 Apr	The Curry-Howard Isomorphism	
	Lecture 3	15 Apr	Review for the Final Exam	