Lecture 1 [9/5/2018]

Background Information

- Grading: 10 homework assignments (10%), midterm (25%), final (65%)
  - Midterm exam: Thursday October 4, 6:05-8:55
  - Exams: few multiple choice, more short answer
- Learn-OCaml: exercise platform – cs302.cs.mcgill.ca
- Prof. Pientka (McConnell Eng. 107N, Friday 9:30-11)
- Goals
  - Thorough intro to fundamental concepts in programming languages (higher-order functions, state-full vs. state-free computation, modelling objects/closures, exceptions to defer control, continuations to defer control, polymorphism, partial evaluation, lazy programming, modules, etc.)
  - Show different ways to reason about programs (type checking, induction, operational semantics, QuickCheck, etc.)
  - Introduce fundamental principles in programming language design (grammars and parsing, operational semantics and interpreters, type checking, polymorphism, and subtyping)
  - Expose students to a different way of thinking about problems
- Why OCaml? Equal playing field, allows us to explain and model object-oriented and imperative programming, isolates concepts, statically typed language enforces disciplined programming and demonstrates types are important maintenance tool, easy to reason about runtime behavior and cost
- Assignments can be tested online and you can continue to submit until you get 100%
- LearnOCaml
  - Go to link above, creates a token and writes it down somewhere (USE STUDENT ID AS NICKNAME)
  - Go to exercises → Demo Exercise, Homework 1

Basic Concepts

What is OCaml?

- Statically typed functional programming language
- Statically typed
  - Types approximate runtime behavior
  - Analyze programs before executing them
  - Find and fix bugs before testing
- Functional
  - Primary expressions are functions
  - Functions are first class (we can return functions or take them as inputs)
  - Pure vs. impure (pure→ doesn't directly manipulate memory, making them easier to reason about; rose in popularity again with parallel programming; OCaml is impure)
  - Call-by-value vs. lazy (call-by-value: evaluate argument of function first, then pass it through function; lazy: evaluates only when needed)

Concepts for Today

- Writing and executing basic expressions
- Learn to read error message: REPL (reevaluate programs and loops)
- Names, values, basic types
- Scope

Basic Expressions

```
# 1;;
```

- always finish line with two semicolons

```
- : int = 1
# 3 + 2;;
-  : int = 5
```

- before this was evaluated, it checked that it made sense (did: 2 and 3 are both integers, + allows you to combine two integers)
- Types approximate the value an expression would compute

```
# 2.2 + 3;;
```

- Type error (expression has type float but was expected of type int)

```
# 3.14;;
- : float = 3.14
# 3.14 +. 2.2;;
- : float = 5.34
```

- Add . after operator for floats

```
# 2.2 + .3;;
```

- Type error (expression has type int but was expected of type float)

```
# true;;
- : bool = true
# 3 = 2;;
- : bool = false
```

- Can use logical operators with Booleans

```
# 3 = 3.0;;
```

- Returns error message, expected type int not float

```
# let pi = 3.14;;
val pi : float = 3.14
# let foo = 3 + 2;;
val foo : int = 5
# let (pi : float) = 3.14;;
val pi : float = 3.14
```

- Note: val stores value (i.e. int) not expression

```
# let m = 3 in
   let n = m * m in              note: n is 9
   let k = m * m in k * n;;      note: k is 9, therefore k * n is 81
- : int = 81
```

Lecture 2 [9/7/2018]

OCaml: A Statically Typed Functional Programming Language

- OCaml type checks programs BEFORE running them, allowing us to find bugs before running and testing the program
- The types approximate runtime behavior

Overview for Today

- Write/execute basic expressions
- Learn to read error messages – debugging is the goal of the first assignment question
- Names, values, and basic types
- Variable, bindings, scope of variables
- Simple functions

Basic Expressions

- Imperative languages generally involve a series of commands modifying the state explicitly, but in functional programming (like OCaml) we typically work with expressions that then evaluate to values
- When typing in your homework, etc., you don't need to use the semicolons as we did last class, that is only to test basic expressions (immediate evaluation, not how programs normally work)
- Expressions have a value and a type most of the time
  - Example: `2+3;;` has the value 5 and the type int, true has the value true and the type bool
  - Some well-typed expressions have no value (example: divide by zero)
- Expressions always have the same type as the final value it computes

Examples

```
if true then 1.0 else 2.3;;
```

- Value is 1.0, type is float (defaults to true)

```
if true then 1.0 else "foo";;
```

- Error, because statically we do not know if the statement is always true, and therefore we don't know which branch is taken, and expressions must have the same type as the final value (would it have the type of float or string??)
- "Error: This expression has type string abut an expression was expected of type float"

```
3/0;;
```

- "Exception: Division_by_zero"
- Note→ has an effect (raises runtime exception) rather than a value (what would the value be, given you cannot divide by zero)

Variables, Bindings, and Variable Scope

- A **binding** binds a variable name to a value
  - Global (must be top level)
    `let <name> = <expr> in <expr>`
  - Local (any level)
    `let <name> = <exp>`

- Bindings are not updated, they are only overshadowed (for example, if you have a variable x that stores int 3, and you declare a variable x that stores string "foo", you will then have two variables x, but if you access it, the program will by default take the most recent one (in this case, "foo")
- ASIDE – functional programming languages tend to have built in garbage collection algorithms (along with many other modern languages)

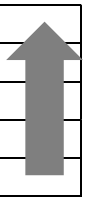
Examples

```
let pi = 3.14;;
let g = 3 + 2;;
let (m : int) = 3 in
let (n : int) = m * m in
let (k : int = m * m in
k * n;;
```

- Documenting the type of variable (ex: `(m : int)`) is unnecessary, and purely for documentation (easy understanding of code)

Variables stored in stack (order: bottom up) before the last line is run (local variables)

k	9 * 9 → 81
n	3 * 3 → 9
m	3
g	5
pi	3.14



Variables stored in stack (order: bottom up) after the last line is run

g	5
pi	3.14

```
let m = 3 and n = 4 in m * n;;
```

- In this example, m and n are created and then subsequently removed when the program runs (will return `- : int = 12`)

```
let x = 3;;
```

- This is a global binding, and most occur on the top level (not in a loop, if-statement, etc.)

Functions

- Previously declared functions see the past, not the future (does NOT go to most recent variable of that name on the stack, instead goes to most recent before the function was declared)
- Function types are right associative
  - Meaning: `f : 'a -> int -> int` will be read as `f : 'a-> (int-> int)`
  - Note: 'a is alpha (polymorphic variable, holds place of something) in OCaml

Examples

```
let pi = 3.14;;
let area = function r -> pi *. r *. r;;
```

- Keyword function indicates there will be an input for the binding
- `area` has type float -> float (returns: `val area : float -> float = <fun>`)

- Equivalent:

```
let area r = pi *. r *. r;;
area (2.0);;
```

- Equivalent:

```
area(2.0);;      area 2.0;;
```

- Note: area is 12.56

```
let a2 = area (2.0);;
let pi = 6.0;;
area 2.0;;
```

- Note: area is still 12.56 despite pi being "updated", could get correct area by redeclaring area function

pi	6.0
a2	12.56
area	function r-> pi *. r *. r
pi	3.14

```
let f x x = x + 2;;
```

- Means `let f = function x -> function x -> x + 2;;`
- Note that the first "x" value entered will be overshadowed by the second and not used

```
f 400 2;;
```

- Returns `- : int = 4` (see stack below)

```
f 400;;
```

- Doesn't complete function (partial evaluation), because not the correct number of inputs (waits for additional input)
- Returns `- : int -> int = <fun>`

f	function x -> function x -> x + 2
x	2
x	400

```
let f (x,y) = y + 2;;
- : val f : 'a * int -> int = <fun>
```

- This function is a **tuple**, and doesn't allow for partial evaluation (indicated by parentheses around multiple variables separated by commas, and by the * (product) in the returned statement)
    - Generally, tuples are easier to debug for beginners (more specific and clear errors), but partial evaluation gives other functions an advantage and neither has a significant runtime advantage

Recursive Function

- Comments are surrounded by (* *) in OCaml
- The `rec` keyword indicates a recursive function is being declared

Examples

```
(* factorial -
     fact n = n!
     invariant: n>=0
```

```
          effect: raises an exception if n<0
     *)
exception Domain
let fact n =
     let rec fact' n =
          if n = 1 || n = 0 then 1
          else n * fact'(n-1)
in
if n<0 then raise Domain
else fact' n
```

- In this program, it will allocate lots of frames and work its way up the stack to get the final result
- **Tail recursion** is the idea where you give an additional argument to the function that acts like an accumulator (see, prev. function vs. next one)

```
exception Domain
let fact n =
     let rec fact' n acc =
          if n = 1 || n = 0 then 1
          else n * fact'(n-1) (acc * n)
in
if n<0 then raise Domain
else fact' n 1
```

Lecture 3 [9/12/2018]

Announcements

- Homework assignments MUST be completed on your own in order to receive credit
- Remember your token to log in to server again, otherwise you could lose all your work and make more work for the TAs

Overview

- Pattern matching
- Creating interesting types rather than just working with Strings, ints, etc.

Pattern Matching

User-Defined (Non-Recursive) Data Type (focus – representing a deck of cards)

- First: represent one individual card
  - Card properties: suit (heart, diamond, club, spade) and rank (A, 2, 3, …, J, Q, K)
  - In OCaml, we declare a new type together with its elements

```
type suit=Clubs | Spades | Hearts | Diamonds
type rank=Ace | Two | Three | Four | Five | Six | Seven |
Eight | Nine | Ten | Jack | Queen | King
```

    - type names must be lowercase in OCaml (suit, rank)
    - We call Clubs, Spades, Hearts, Diamonds **constructors**, which must have a capital letter
  - Use **pattern matching** to analyze elements of a given type

```
match <expression> with
   | <pattern> -> <expression>
   | <pattern> -> <expression>
   …
   |<pattern> -> <expression>
```

    - A **pattern** is either a variable, underscore (wild card), or a constructor
- First problem – comparing suits by writing a function (see function `dom` under Examples)
  - You can write things that return different type (i.e. `int` or `bool`) by creating a type that has a pipe to allow either (see `type int_bool` under Examples)
  - Pattern matching is used to extract data
- What's in your hand?
  - A hand can be empty
  - If c is a card and h is a hand, then the result is also a new hand (constructor = `Hand`) → infinitely many elements to a hand
  - Either empty or it consists of a card together (followed by) another `Hand(c,h)`

Examples

Key Example: Playing Cards

```
type suit = Clubs | Spades | Hearts | Diamonds
type rank=Ace | Two | Three | Four | Five | Six | Seven | Eight | Nine
| Ten | Jack | Queen | King
(* dom(s1,s2) = true iff suit s1 beats or is equal to suit s2 relative
     to the ordering S>H>D>C
```

```
     Invariants: none
     Effects: none
*)
let dom (s1, s2) = match (s1, s2) with
     | (Spades, _) -> true
     | (Hearts, Diamonds) -> true
     | (Hearts, Clubs) -> true
     | (Diamonds, Clubs) -> true
     | (s1, s2) -> s1 = s2
```

- Equivalent to the last match statement: `| ( _, _) -> s1 = s2`
- Note → you can use or statements rather than having separate lines for each pattern matching statements, however it is messy and not advised; for this, add a pipe ("|") between the (x, y) and the ->
- If you miss cases: you may get a warning that "this pattern-matching is not exhaustive", if you use inputs that aren't covered, you may get a "`Match_failure`" exception
- Each matching statement will be tried in order, in a good program every case will be considered

```
type card = rank * suit
let c : card = (Ace, Hearts);;
- : val c : card = (Ace, Hearts)
```

- Note – make variables of type (above)

```
type hand = Empty | Hand of (card * hand)
let h0 = Empty;;
- : val h0 : hand = Empty
let h1 = Hand ((Queen, Hearts), h0);;
- : val h1 : Hand ((Queen, Hearts, Empty)
let h2 = Hand ((King, Hearts), h1);;
- : val h2 : hand = Hand ((King, Hearts), Hand ((Queen, Hearts),
     Empty))
let h3 = Hand((Ace, Clubs), Hand ((Ten, Diamonds), h2));;
- : val h3 : hand =
     Hand((Ace, Clubs,
          Hand ((Ten, Diamonds),
               Hand ((King, Hearts), Hand ((Queen, Hearts), Empty))))
```

- Note: hand is a recursive data type
- Now, we want to write a function to extract an element from a hand (returns a hand consisting of all the cards in the hand of a given suit)

```
(* extract : suit -> hand -> hand
     Extract s h returns a hand consisting of all card in h of suit s
     Invariants: none
     Effects: none
*)
```

```
let rec extract (s:suit) (h:hand) =
     | Empty -> Empty
     | Hand ((_, s') as c, h') ->
          if s = s' then Hand (c, extract s h')
          else extract s h'
```

- Note – (_, s') as c names the tuple but also allows you to look inside
    - The _ instead of r is because in this function, you only care about the suit, not the rank

Aside: int_bool

```
type int_bool = Int of int | Bool of bool
```

- Two constructors, `Int` takes in one, or `Bool` takes in one element
- The keyword `int` is used to indicate what type of element the type will use

```
Int 3;;
- : int_bool = Int 3
Bool true ;;
- : int_bool = Bool true
Int 10;;
- : int_bool = Int 10
```

- The above examples are `int_bool`s

```
10;;
- : int 10
```

- This example is not an `int_bool`

Lecture 4 [9/14/2018]

Lists

- How do we define lists?
- Lists are polymorphic (can store anything), keeping them generic
  - [] is of type `'a list`
  - If `x` is of a type `'a` and `xs` is of type `'a list`, then `x::xs` is an `'a list`
- [] and :: (infix operator) are constructors

Keeping it Old-Style

```
(* head : 'a list -> 'a *)
let head (h::t) = h
(* tail : 'a list -> 'a list *)
Let tail l = match l with
     | [] -> []
     | h::t -> t
(*Destructor style*)
let rec app (l1, l2) =
     if l1 = [] then 12
     else head(l1)::(app (tail(l1), l2))
```

- The above example is bad style → hard to tell if its defined on all inputs, will always succeed, etc. (example – if l1 is an empty list, what is the head, etc.?)
- Compare to the *append* function in the examples below → append is a lot simpler, more readable, and efficient
  - Typically, pattern matching results in shorter programs

Revisiting Cards in a Hand (from last class)

```
(* already defined:
   rank, suit, card – an abbreviation, hand – a recursive data type
*)
```

- `hand` – recursive, starts as `Empty`, constructor = `Hand`
  - Why are we using a recursive data type rather than a list?

```
(* find : (rank * hand) -> suit option
     Find the first card with rank r in hand h and return its
     corresponding suit s by Some(s)
     If there is no card with rank r, return None
*)
```

- In writing this function we make use of the pre-defined, parameterized database `'a option`
- Type `'a option` = `None | Some of 'a`
  - Polymorphic data type (any type)

```
let rec find (r, h) = match h with
     | Empty -> None
     | Hand ( (r', s'),h') ->
          if r' = r then Some s'
```

```
            else find (r, h')
```

- Note → first Some __ call determines the type (in this case, `Some s'` → suit option) i.e. cannot then have `Some 5` (type `int`) in the `else` statement

Other Examples

```
1::[];;
- : int list = [1]
1::(2::(3::[]));;
- : int list = [1; 2; 3]
```

- Note that when :: is used, there is an empty list nested at some point

```
[None ; Some 3; Some 5; None];;
- : int option list = [None; Some 3; Some 5; None]
(* append: 'a list * 'a list -> 'a list
     Append (l1, l2) returns a list consisting of the elements of l1
     followed by the elements of l2
     Invariants: none
     Effects: none
     NOTE – this operation is defined in ML's Standard Basis via the
     right-associative infix operator "@"
     Observe the running time is proportional to the length of the
     first argument
     example – append [1; 2; 3] [4; 5] → [1; 2; 3; 4; 5]
*)
let rec append l1 l2 = match l1 with
     | [] -> l2
     | a::l1' -> a::(append l1' l2)

let list1 = [1;2;3;4];;
val list1 : int list = [1; 2; 3; 4]
let list2 = [8;7;9;10];;
val list2 : int list = [8;7;9;10]
append list1 list2;;
- : int list = [1; 2; 3; 4; 8; 7; 9; 10]
list1;;
- : int list = [1; 2; 3; 4]
list2;;
- : int list = [8; 7; 9; 10]
```

- Note that simply calling append does not alter either `list1` or `list2` (above)
- In order to store the new list, you have to declare a new variable and set it equal to `append list1 list2` (as shown below)

```
let list3 = append list1 list2;;
val list3 : int list = [1; 2; 3; 4; 8; 7; 9; 10]
[1; 2; 3; 4; list2];;
```

- The above line causes an error: “This expression has type int list but an expression was expected of type int”, because list is defined recursively as a polymorphic element and a list of other polymorphic elements. To correctly construct a list using another list, see the line below, or use a function such as `append`

```
1::2::3::(4::list2);;
- : int list = [1; 2; 3; 4; 8; 7; 9; 10]
[1;2;3]::4;;
```

- The above line causes an error: “This expression has type int but an expression was expected of type int list list”. This occurs for similar reasons as the previously discussed error. To do this properly, use either of the lines below (note - @ works like append, but is a built-in function).

```
append [1;2;3] [4];;
- : int list = [1; 2; 3; 4]
[1;2;3] @ [4];;
- : int list = [1; 2; 3; 4]
[ [1;2;3] ; [8;9] ; [8] ];;
- : int list list = [[1;2;3]; [8;9]; [8]]
```

- You are able to have nested lists (see above few lines)

```
(* reverse a list
     rev : ‘a list -> ‘a list
*)
let rec rev l = match l with
     | [] -> []      (* the reverse of an empty list is the same*)
     | h::t -> rev t@[h]
```

- Note – DO NOT write `rev t::h` (common error)

```
let rec rev’ l acc = match l with
     | [] -> acc
     | h::t -> rev’ t ([h] @ acc)
```

- NOTE – tail end recursion isn’t always trivial (i.e. append)

Lecture 5 [9/19/2018]

Announcements

- LearnOcaml platform → don't use safari, don't send screenshots of code to TAs, save work locally by downloading it to your account
- Homework → makes it easier to follow along in class, exams build on assignments, material builds up gradually so it helps to learn it as you go along

Trees

Inductive Definition of a Binary Tree

- The empty binary tree `Empty` is a binary tree
- If `l` and `r` are binary trees and `v` is a value of type `'a` then `Node(v, l, r)` is a binary tree
- Nothing else is a binary tree
- How to define a recursive data type for OCaml

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

Other Examples

```
type 'a tree = Empty | Node of 'a * 'a tree * 'a tree
let t0 = Node ((3, "Charlie"), Empty, Empty)
let t1 = Node ((5, "Patrick"), Empty, Empty)
let t2 = Node ((10, "Tamar"), t0, t1)
```

- In `t2`, Tamar is the parent/root, Charlie and Patrick are the children/leaves

```
let t3 = Node ((21, "Chloe"), t2, t0)
```

- Now Charlie is in the tree twice, no mechanism is in place to prevent this, all the definition says is that each has two children
- Note – trees have set types (can't have ints and floats at different nodes in the tree)

```
(*size of a tree
     size: 'a tree -> int
     size(T) = n where n is the size of the tree determined by the
     number of nodes
*)
let rec size t = match t with
     | Empty -> 0
     | Node (v, l, r) -> size l + size r + 1
(* insert: 'a * 'b -> ('a * 'b) tree -> ('a * 'b) tree
     insert (x,c) T = T' where (x,d) has been inserted into T
*)
let rec insert ((x,d) as e) t = match t with
     | Empty -> Node (e, Empty, Empty)
     | Node ((y, d'), l, r) ->
          if x = y then Node(e, l, r)
          else (if x>y then Node((y,d'), l, insert e r)
```

```
                else Node((y,d'), insert e l, r))
```

- Note → new tree is returned, the tree you call insert on is not altered
- Additional example → lookup (see notes posted on website

Higher Order Functions

- Higher-order functions allow us to abstract over common functionality
- Programs can be very short and compact
- Programs are reusable, well-structured, and modular
  - Generic programs
- Each significant piece of functionality is implemented in one place
- Allow us to pass functions as arguments and return them as results
  - Functions are first-class values (have type, etc.)

Abstracting over Common Functionality

- $\sum_{k=a}^{k=b} k$

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

- $\sum_{k=a}^{k=b} k^2$

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

- $\sum_{k=a}^{k=b} 2^k$

```
let rec sum (a,b) =
if a>b then 0 else exp(2,a) + sum(a+1, b)
```

- Can we write a generic sum function? Yes!

Non-Generic Sum (old)	Generic Sum (function as argument)
`sum: int * int -> int`	`sum: (int -> int) -> int * int -> int`

- Keyword function allows for pattern matching, fun doesn't (when you define functions)
- How about only summing up odd numbers between a and b?

Only Odd/Even

```
let rec sumOdd (a,b) =
     if (a mod 2) = 1 then sum (fun x -> x) (a, b)
     else sum (fun x -> x) (a+1, b)
(* general function, allowing to increment differently*)
let rec sum f (a,b) inc =
     if (a>b) then 0 else (f a) + sum f (inc(a), b) inc
let rec sumOdd (a,b) =
     if (a mod 2) = 1 then sum (fun x -> x) (a,b) (fun x -> x + 2)
     else sum' (fun x -> x) (a+1, b) (fun x-> x+2)
```

Multiplication

```
let rec product f (a,b) inc =
     if (a>b) then  else (f a) * product f (inc(a), b) inc
(* new general function*)
let rec sum f (a, b) inc acc =
```

```
     if (a>b) then 0 else sum f (inc(a), b) inc (f a + acc)
let rec product f (a,b) inc =
     if (a>b) then 1
     else (f a) * product f (inc(a), b) inc
```

Very Generalized Function

```
(*    comb: combine, either * or +
      f: what we do to the a
      inc: how we increment a to get b
      base: what we return when a>b
*)
let series comb f(a,b) inc base =
      let rec series' (a,b) =
            if a>b then base
            else series' (inc(a), b) (comb base (f a))
      in
      series' (a,b) base
```

- Takeaway: abstraction is good, too much causes headaches; abstraction and higher-order functions are very powerful mechanisms for writing reusable programs

Other Examples

```
let id x = x
let sum f (a,b) = if (a>b) then 0 else f a + sum (a+1, b)
# let plain_sum = sum id;;
-: val plain_sum : int * int -> int = <fun>
```

- This is equivalent to

  ```
  let plain_sum (a,b) = sum id (a,b);;
  ```

  OR

  ```
  let plain_sum = sum (function x-> x);;
  ```

  but the first way is preferred

```
let rec sum f (a,b) = if (a>b) then 0 else f a + sum f (a+1, b)
# plain_sum (1,4);;
- : int = 10
# let sq_sum = sum (function x -> x * x) ;;
```

Lecture 6 [9/21/2018]

Announcements

- Midterm is 1.5 hrs
- Class on Wednesday (led by TA), no class Friday, no classes Monday but review session by TA, review session on the following Wednesday
- Questions → type of this function/program, what is the final value, writing recursive/pattern matching programs, higher order functions

Higher Order Functions

- Functions are first class values and can be returned as well as used as parameters
- Common higher-order functions (built in)
  list functions (`map, fold_right, fold_left, tabulate,` etc. → see implementation online)
- `List.map` applies a function to every element in a list, and returns a new list of the results

```
(* map : ('a -> 'b) -> 'a list -> 'b list
   Map f l = l' where l' is the list we obtain by applying f to each
   element in l
*)
let rec map f l = match l with
     | [] -> Empty
     | h::t -> (f h) :: map f t
# map (fun x -> string_of_int x) [1; 3; 7];;
- : string list = ["1"; "3"; "7"]
# map (fun x -> x + 3) [1; 3; 7];;
- : int list = [4; 6; 10]
# map string_of_int [1; 3; 7];;
- : string list = ["1"; "3"; "7"]
```

- This works too, and is better than the first call of map above, because `string_of_int` is a preexisting function
- `List.filter` returns all elements of list `l` that satisfy the predicate `p`

```
(* filter : ('a -> bool -> 'a list -> 'a list *)
```

- How do we get this type for filter?
  - Type of `p` -> type of `l` -> whatever it returns
    - Call: `match l` → `'a list`
    - `[] -> []` → some kind of list is returned
      - then `h::filter p t` → `h` is `'a` → `'a list`
    - `p h` → `p` is a function, we know h is the head of 'a list therefore `h` is `'a`; is in an if statement as a guard → computes a boolean → type (`'a -> bool`)

```
let rec filter p l = match l with
     | [] -> []
     | h::t ->
```

```
            if p h then h::filter p t
            else filter p t
```

- `List.fold_right` and `List.fold_left` use an initial accumulator to compute using the values of a list from right to left and left to right, respectively

```
(*fold_right (‘a -> ‘b -> ‘b) -> ‘a list -> ‘b -> ‘b
     fold_right f [x1; x2; …; xn] init returns
          f x1 (f x2 … (f xn init) …)
          + x1 (+ x2 … (* xn 1)
          ^x1 (^x2 … (^xn “”))
     or init if the list is empty
*)
```

- Computes the “right” value of `f` first (last element → first element, right → left)

```
let rec fold_right f l init = match l with
     | [] -> init
     | h::t -> f h (fold_right f t init)
# fold_right (fun h acc -> h + ac) [1; 2; 3; 4] 0;;
- : int = 10
Process – fold right
→ 1 + fold_right (fun h acc -> h + acc) [2; 3; 4] 0
→ 1 + 2 + fold_right (fun h acc -> h + acc) [3; 4] 0
→ 1 + 2 + 3 + fold_right (fun h acc -> h + acc) [4] 0
→ 1 + 2 + 3 + 4 + fold_right (fun h acc -> h + acc) [] 0
→ 1 + 2 + 3 + 4 + 0
→ int = 10
```

- Why is `acc` needed? You need to know what to add `h` to, you can’t modify the value of `h` so you need the accumulator

```
# fold_right (fun h acc -> h * acc) [1; 2; 3; 4] 1;;
- : int = 24
# fold_right (fun h acc -> if h mod 2 = 0 then h + acc else acc) [1;
2; 3; 4] 0;;
- : int = 6
```

- Only added up the even numbers

```
(* fold_left f init [x1; x2; …; xn]
   returns
   f xn … (f x2 ( f x1 init))…)
   or init if the list is empty
*)
----- fold_left was left as an exercise -----
```

- From posted notes:

```
let rec fold_left f b l = match l with
     | [] -> b
     | h::t -> fold_left f (f h b) t
```

```
(* tabulate (n, f)
   Returns a list of length n equal to [f(0), f(1), …, f(n-1)],
   created from left to right.
*)
let rec tabulate n f =
     if n<0 then acc
     else tabulate (n-1) f ((f (n-1)) :: acc)
```

Returning functions as outputs

- Simplest possible function as an example: `curry`
  - Takes as input `f: (‘a*’b) -> ‘c`
  - Returns as result a function `‘a -> ‘b -> ‘c`
- `uncurry` does the opposite

```
let curry f = (fun a b -> f(a,b))
```

- Equivalent: `let curry f a b = f(a, b);;`

```
let uncurry f = (fun (a,b) -> f a b)
```

- Functions are opaque, you can only observe what they’re doing but you can’t inspect the code

  Example:

```
# let f0 = curry (fun (x,y) -> x+y);;
- : val f0 : int -> int -> int = <fun>
```

- You can’t compare functions to see if they’re equal to each other, you can only observe whether they compute the same value (not practical, would have to be run on all possible

```
# let swap f = fun(b,a) -> f(a,b)
- : val swap : (‘a * ‘b -> ‘c) -> ‘b * ‘a -> ‘c = <fun>
```

- Swap – swaps the order of inputs for a function
- Function types are right associative!!

Lecture 7 [10/3/2018]

Midterm Questions

- Is Church encoding on the midterm? Yes (is it zero, make it an integer, addition, etc.)
  - An example that allows you to practice higher order functions
  - No partial evaluation, etc. (after exam)
- Midterm room assignments: posted tonight/tomorrow morning by last name on myCourses under Course Information

Midterm Topics Summary

- Reasoning about programs
  - Static type checking (what is the most general type of an expression, when is type checking done – before evaluating program)
  - Semantics (how is a program evaluated, what value does it produce)
- Principles of functional programming
  - Pattern matching and recursion
  - Higher order functions
    - Passing functions as arguments (abstracting over common functionality (ex- sum function), proficiency with using built-in higher-order functions)
    - Returning functions as results (examples we have seen, returning a function that swaps the arguments, currying, uncurrying, Church numerals)

Type Questions

1. Expression `fun x y -> if x * (-1) < 0 then (y,x) else (x,y)` has the most general type
   a. int * int
   b. int -> int -> int * int
      i. x is multiplied by -1, so it must be an int
      ii. as (x,y) or (y,x) can be returned, and they both must be of identical type, because x is an int, it must be int*int, and y must be an int
   c. int * int -> int * int
   d. int -> ‘a -> int * int
   e. int -> ‘a -> int * ‘a
   f. int * ‘a -> int * int
   g. int * ‘a -> int * ‘a
2. What is the most general type of the following expression

```
let rec f b l = match l with
      | [] -> b
      | x::xs -> f x
```

   a. ‘a -> ‘a list -> ‘a
   b. (‘a -> ‘b) -> ‘a -> ‘a list -> ‘b
   c. **Ill-typed**
      i. f takes in two arguments when it is first called, and only one when it is called recursively
   d. int -> int list -> int

3. What is the most general type of the following expression

```
let rec f b l = match l with
      | [] -> b
      | x::xs -> f x l
```

Answer: ‘a -> ‘a list -> a (let b be of type ‘a, l of type ‘b list. As f is called on b l and then x l, where x is the head of l, b and x must be of the same type, therefore l is also an ‘a list. b is returned, so it returns type ‘a)

4. What is the most general type of the following expression

```
let rec zoom x =
      if zoom x>3 then zoom (x-1) else zoom (x+1)
```

   a. Infinite loop
   b. int -> int
      i. zoom x is compared to 3, so it is an int (result is an integer), zoom is called on x-1 or x+1, so x is also an int
   c. int

5. What is the most general type of the following expression:

```
let double (f, x) = f (f x)
```

   a. infinite loop
   b. (‘a -> ‘a) -> ‘a -> ‘a
   c. (‘a -> ‘a) * ‘a -> ‘a
   d. (‘a -> ‘b) -> ‘a -> ‘b

Value Questions

1. What value does this expression have?

```
let x = 3 in
let y = x + 3 in
let check x = x = 3 in
let x = y in
      check x
```

   a. ANSWER: x is 3, y is x + 3 or 6, check takes in x and decides whether it equals 3, x is overshadowed to equal 6, check is called on x, as x is not equal to 3, the result is **false**

Programming Questions

```
(* simplified roulette, with two colours that we can bet, but zero
     means everyone loses *)
type colour = Red | Black        (* 2 colours you can bet on *)
type result = colou roption      (* result of a run *)
type amt = int
type bet = amt * colour          (* bet amount and to what colour *)
type id = string
type player = id * amt * bet option
let compute (am, col : bet) : result -> int = function
     | None -> 0
     | Some col’ -> if col = col’ then am * 2 else 0
```

Commented [VP1]: Go over later

```
(* equivalent:
     let compute (am, col : bet) (r : result) = match r with
          | None -> 0
          | Some col' -> if col = col' then am * 2 else 0
*)
let players = [ ("Aliya", 1000, Some (400, Red));
                ("Jerome", 800, Some (240, Black));
                ("Mo", 900, Some (200, Black));
                ("Andrea", 950, Some (100, Red))]
(* solve the following questions using higher order functions rather
     than pattern matching or recursion *)
(* Q1: given a list of players compute the new amounts each player has
     by adding their win/loss bets to None *)
let rec compute_all_results l r = match l with
     | [] -> []
     | (id, amt, b)::xs-> (id, amt + (compute b r), None) ::
          compute_all_results xs r
```

- Note – this first solution uses pattern matching and recursion rather than HOF, and would only receive partial credit

```
let compute_all_results l r =
     List.map (function (id, amt, Some b) -> (id, amt + (compute b r),
          None)
                      | (id, amt, None) -> (id, amt, None)) l
```

- A version of List.fold is advisable when elements of a list are added, here we are taking a list and returning a new list, so List.map is better for this case (alters each element)

Lecture 8 [10/10/2018]

Midterm Aftermath

- Grades will be released towards the end of the week, as well as solutions
- Concerns: office hours Fridays 9:30-11:00 am
- Grades will not be adjusted on the midterm
  - In the past: 3 bonus questions were released to boost grade and provide more practice
- Announcement: new homework out (quiz – today's material)
  - Very different than what you use LearnOCaml for → ONE SHOT TO SUBMIT ANSWERS
  - 10 multiple choice questions, but on the final it will be short answer so it is advised to actually write the proofs and ask at office hours about it
  - Announcement will be on myCourses

Proofs

Introductory Example

- How would you prove that all slices of cake are tasty?
  - Define what it means to have a cake inductively
  - One slice is cake, two slices combined are still cake
  - Prove a single piece of cake is tasty
  - Use the recursive definition of the set to prove that all slices are tasty, concluding that all slices of cake are tasty
- We will look at how proofs can use the inductive definition of things (not just numbers)

Lists

- The empty list [] is a list of `type 'a list`
- If x is an element of `type 'a` and `xs` is a list of `type 'a list` then `x::xs` is a list of `type 'a list`
- Nothing else is a list of type `'a list`
- How to prove properties about lists?
  - Step 2: how to reason inductively about lists? Analyze their structure
    - To prove a property P(1) about a polymorphic list l
    - Base case: l = []
      Show P([]) holds
    - Step case: l=x:xs
      IH P(xs)
    - Assume the property P holds for lists smaller than l
      Show P(x::xs holds)
      - Show the property P holds for lists

Two Programs: Do they compute the same value?

A)

```
(* rev : 'a list -> 'a list *)
let rec rev l = match l with
     | [] -> []
```

```
    | ::l -> (rev l) @ [x]
```

B)

```
(* rev' : 'a list -> 'a list *)
let rev' l =
    (* rev_tr : 'a list -> 'a list -> 'a list *)
    let rec rev_tr l acc = match l with
        | [] -> acc
        | h::t -> rev_tr t (h::acc)
    in
    rev_tr l []
```

- How would you show they compute the same value?
  - Theorem – for all lists l, rev l = rev' l
  - Consider, what is the relationship between l, acc and rev_tr l acc

PROVE (attempt 1): For all `l`, `rev l = rev_tr l []`

Induction on `l`

Base: `l = []`

`rev []`

➔ [] by program `rev`

`rev_tr [] []`

➔ [] by program `rev_tr` (evaluates to `acc`, `acc` is `[]`)

Therefore, `rev []` and `rev_tr [] []` both evaluate to `[]`

Step: `l = x:xs`

Induction hypothesis: `rev xs = rev_tr xs []`

To prove: rev (x::xs) = rev_tr (x::xs) []

`rev (x::xs)`

➔ `rev xs @ [x]` by program rev

`rev_tr (x::xs) []`

➔ `rev_tr xs [x]` by program rev_tr

➔ Note: you can go no further, as the accumulator now being studied in the tail recursive scenario is not empty, as it is in the induction hypothesis ➔ try proving something more general

PROVE (attempt 2): For all `l, rev l @ acc = rev_tr l acc`

Induction on `l`

Base: `l = []`

What to prove?

~~(A) ➔ `rev [] @ []`~~ doesn't work

(B) ➔ `rev [] @ acc`

- You have `acc` in the statement, not `[]`, so using `[]` doesn't apply to every case

`rev [] @ acc`

➔ `[] @ acc` by program `rev`

➔ `acc` by program `@`

`rev_tr [] acc`

➔ `acc` by program `rev_tr`

Commented [VP2]: What does it mean "by program @"

Therefore, `rev [] @ acc` and `rev_tr [] acc` both evaluate to acc

Step: l = x::xs

Induction hypothesis: for all `acc`, `rev xs @ acc = rev_tr xs acc`

Prove `rev (x::xs) @ acc = rev_tr (x::xs) acc`

`rev (x::xs) @ acc`

- ➔ `(rev xs @ [x]) @ acc` by program `rev`
- ➔ `rev xs @ ([x] @ acc)` by lemma: associativity of `@`
- ➔ `rev xs @ (x::acc)` by program `@`

`rev_tr (x::xs) acc`

- ➔ `rev_tr xs (x::acc)` by program `rev_tr`

By the induction hypothesis, choosing for `acc'` the value `x::acc, rev xs @ acc = rev_tr xs acc`, so `rev xs @ (x::acc)` and `rev_tr xs (x::acc)` evaluate to the same value

Main theorem: `for all l, rev l = rev' l`

`rev' l`

- ➔ `rev_tr l []` by program `rev'`
- ➔ `rev l @ []` by previous theorem (proved)
- ➔ `rev l` by lemma: `l@[]=l`

- Recap important principles
  - The induction hypothesis should fall out of the statement that we are trying to prove
  - You must prove the statement for the base case and the step case, assuming it holds for the smaller thing
  - Every line in the program must be appropriately and clearly justified

Example

- Theorem: for all l, acc,
  length (rev_tr l acc) v and length l + length acc v

Inductive definition of a binary tree

- You can use induction for other things too, not just list
- The empty binary tree `Empty` is a binary tree
- If `l` and `r` are binary trees and `v` is a value of `type 'a`, then `Node(v, l, r)` is a binary tree
- Nothing else is a binary tree
- How to define a recursive data type for trees in OCaml
  `type 'a tree = Empty | Node of 'a * 'a tree * 'a tree`
- How to prove it?
  - How to reason inductively about trees? Analyze their structure
  - To prove a property P(t) holds about a binary tree `t`
    - Base case: `t = Empty`
      Show P(`Empty`) holds

- Step case: `t = Node(x,l,r)`
  IH P(`l`), IH P(`r`)
  assume the property P holds for trees smaller than `t`
- Show P(Node(x,l,r) holds → show the property P holds for the tree t)

```
let rec insert ((x,d) as e) t = match t with
     | Empty
     | Node ((y, d'), l, r) ->
          if x = y then Node(e,l,r)
          else (if x<y then Node((y, d'), insert e l, r)
               else Node((y, d'), l, insert e r))
let rec lookup x t = match t with
     | Empty -> None
     | Node ((y, d), l, r) ->
          if x = y then Some(d)
          else  (if x<y then lookup x l else lookup x r)
```

Theorem: for all trees `t`, keys `x`, and data `dx`, `lookup x (insert (x, dx) t)` → `Some dx`

- Base case: tree is empty, step case: tree has >=1 Node
- See full proof in posted code

Exercise

- Give an OCaml data type definition for cake
  - View as binary tree with nothing stored in parent node

Tutorial [9/26/2018 – Prof. Pientka is absent, TA led class]

Church Encoding

- Way of encoding data in terms of higher order functions
- Specifically, we will be focusing on natural numbers today

```
(* define natural number *)
type nat = Zero | Successor of nat
```

- Ex – to get 1: `Successor (Successor Zero)`

```
(* to define a function on a list *)
let rec f xs = match xs with
     | [] -> b
     | h::t -> g h::t
```

- Where b is the base case and g is a function on h::t

```
(* to define a function on natural numbers *)
let rec f n = match n with
     | Zero -> b
     | Succ m -> g f m
```

- Where b is the base case and g is a function on f m
- The function g is called m times
- In church encoding: n = fun f x → f^n x
  - If x is of type ‘a, f is of type (‘a -> ‘a), so f^n x is of type ‘a
  - Overall type: (‘a -> ’a) -> ‘a -> ‘a

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

- Due to technical difficulties, the class was instructed to visit the notes posted on myCourses rather than the tutorial continuing

Lecture 9 [10/12/2018]

State

Computation and Effects

- So far: expressions in OCaml have characteristics
  - An expression has a type
  - An expression evaluates to a value (or diverges)
- Now:
  - Expressions in OCaml may also have an effect (many things, i.e. exceptions, allocating memory, etc.)

Recall: Variable Bindings and Overshadowing

```
# let (k : int) = 4;;
# let (k : int) = 3 in k * k ;;
# k;;
```

- Returns 4, as k = 3 was only allocated temporarily (note: `in` key word)
- We can temporarily overshadow, but not permanently update the binding of a value

How do we program with state - Demo

```
# let r = ref 0;;
val r : int ref = {contents = 0}
```

- In OCaml, we allocate memory using the keyword `ref`
- In this example, `r` stores a reference to the value `0`
- The type `int ref` is an allocation in memory able to store integers

r	→	0

```
# let s = ref 0;;
val s : int ref 0;;
```

r	→	0
s	→	0

```
#s = r;;
- : bool = true
# s == r;;
= : bool = false
```

- To compare the content of cells in memory, use s = r
- To compare the location of cells in memory, use s==r

```
# r + 2;;
```

- Error: This expression has type int ref but an expression was expected of type int
  - Pointer arithmetic is dangerous, does not occur in safe languages such as Java and OCaml
- Note – inaccessible bindings will automatically be collected, don't need to worry about that

```
# r := 3;;
- : unit = ()
# !r;;
- : int = 3
```

- `:=` erased content that `r` points to and replaced it with `3`
  - Returned `unit`
  - Indicates that memory update was successful
- The memory update is known as a side effect
  - Can be observed by reading from r, using `!r`

r	→	**3**
s	→	0

```
# r := 2 + 3;;
- : unit()
# !r;;
- : int = 5;;
```

- `Note that the int sum of 2 and 3 is stored, not the function 2+3`

r	→	**5**
s	→	0

```
# let r = ref 4;;
val r : int ref = {contents = 4}
# !r;;
- : int = 4
```

- Note that the previous value for r doesn't disappear (unless removed via garbage collection), it merely is overshadowed

r	→	**4**
s	→	0
r	→	3

```
# let l = ref [];;
val l : '_a list ref = {contents = []}
# l := [1];;
- : unit = ()
```

- At this point, it can be an '_a list, but once it is used with a type (i.e. int), another type (i.e. float) cannot be used in it

```
# let f = ref (fun x -> x)
val f : ('_a -> '_a) ref = {contents = <fun>}
```

How to program with state? Allocate and compare

- Allocate state
  let x = ref 0 allocates a reference cell with the name x in memory and initializes it with 0
- How to compare two reference calls?
  - Compare address (r==s)
    - Succeeds if both r and s are names for the same location in memory
  - Compare content (r=s)
    - Succeeds if both r and s are names for locations storing the same values
- Read and write
  !x read value that is stored in the reference cell with name x

let {contents = x} = r — Pattern match on value that is stored in the reference cell with name x

- How to update the value stored in a reference cell:
  x:=3
  - Write the value in the reference cell with the name x
  - Previously stored value is overwritten

```
let run = ()
     let pi = 3.14 in
     let area (r:float) = pi *. R *. R in
     let a2 = area (2.0) in
     let (p1:float) = .6.0;
     print_string (“Area a2 = “^string_of_flfot a2
let untasteful
     let pi = ref 3.14 in
let area (r:float) = !pi *. R *. R in
let a2 (2.0) in
(pi := 6.0 ;
     Print_string (“Area a2 = “^string_of_float a2^”\n|
     Print_string (“Area a3 = “^string_of_float
# run());;
Area sa2 = 12.56
- : unit
Foo- : unit = ()
# (3 + 2 ; r := 3.3);;
   • Warning: this expression should have type unit]
#print_to_string;;
- : string -> unit =<fun()>
# print_string “3”;;
3- : unit = ()
```

- Unit has just one value, logically configures to sigma

Global Counter Example

```
let counter = ref 0
let newName () =
     (counter := 1 + !counter ;
     “a” ^ string_of_int (!counter))
let tick () = (counter := !counter + 1; !counter)
let reset () = (counter :=0)
# tick ();;
- : int = 1
# tick ();;
- : int = 2
# tick ();;
```

```
- : int = 3
# newName ();;
- : string = “a4”
```

- How can we create shared state? Counter is globally available right now – what if the user shouldn’t be able to alter it directly? Make counter local
- Revised code:

```
let counter = ref 0
let newName () =
     (counter := 1 + !counter ;
     “a” ^ string_of_int (!counter))
type count = {tick : unit -> int;
              reset: unit -> unit}
```

- Record = two fields, tick and reset

```
let c = (let counter = ref 0 in
let tick () = (counter := !counter + 1; !counter) in
let reset () = (counter :=0) in
{tick = tick;
 reset = rest})
# c;;
- : count = {tick = <fun>; reset = <fun>}
# c.tick;;
- : unit -> int = <fun>
# !counter;;
- : int = 0
# c.tick();;
- : int = 1
# c.tick();;
- : int = 2
# c.tick();;
- : int = 3
# !counter;;
- : int = 0
# c.reset();;
- : unit = ()
# c.tick();;
- : int = 1
```

- Have created an “object” with two methods (tick and reset) that exist only for that method, they alter counter that also exists only for that method
- How do we create more than one object of a given type? Revised code:

```
let newCounter () =
     let c = (let counter = ref 0 in
     let tick () = (counter := !counter + 1; !counter) in
```

```
     let reset () = (counter :=0) in
     {tick = tick;
      reset = rest})
# let c1 = newCounter ();
Val c1 : count = {tick = <fun>; reset = <fun>}
# let c2 = newCounter ();
#c1.tick();;
- : int = 1
#c1.tick();;
- : int = 2
#c1.tick();;
- : int = 3
#c2.tick();;
- : int = 1
#c2.tick();;
- : int = 2
#c1.reset();;
- : int = 0
#c1.tick();;
- : int = 1
#c2.tick();;
- : int = 3
```

- Note → c1 and c2 have their own counters (own tick/reset, alter counters separately, as shown above)
- In this way, objects can be created in OCaml
- In record, the name of the "methods" must match the names of the functions when they were declared

Example

```
let rotate (a, b, c) =
     let temp = !a in
     (a := !b; b := !c; c := temp)

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

Imperative programming in OCaml:

```
let imperative_fact n =
     begin
          let result = ref 1 in
          let i = ref 0 in
          let rec loop() =
               if !i = n then ()
               else (I := !i + 1; result := !result * !i; loop())
```

```
        in
        (loop (); !result)
    end
```

- More complicated than the purely functional version
- Considered bad style in a functional language
- Harder to reason about its correctness

Good uses of state

- Global counter
- Objects with shared state
- Immutable data structure

Lecture 10 [10/17/2018]

Computation and Effects

- Expressions in OCaml may have an effect in addition to its type or value

Type, Value, and Effect (Warmup)

- Given the following expressions, write down its type, value, and its effect, if any:
  - `let x = 3 + 2 in x`
    type – int, value – 5, effect – no
  - `fun x -> x + 3 * 2`
    type – (int -> int), value – function, effect – no
  - `let x = ref (3 + 2) in x`
    type – int ref, value – location in memory that references 5, effect – allocated in memory a location named x, and stored 5 in that location
  - `let x = ref (3 + 2) in x + 1`
    type – does not type check (x is a location in memory, you cannot perform arithmetic on it = static violation)
  - `let x = ref 3 in x := !x + 2`
    type – unit (the assignment is something where you can observe what happens, but lacks a specific, helpful value), value – unit, effect – update x such that it is 5, note that unit is returned
  - `let update x v = x := v`
    type – (‘a ref -> unit), value – function, effect – updates value stored at x
  - `let read x = !x`
    type – (‘a ref -> ‘a), value – function
  - `fun x -> (x := 3; x)`
    type – (int ref -> int ref)

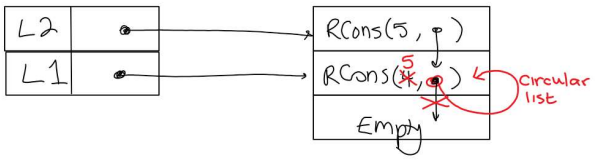
Good uses of state: mutable data structures

- So far, we have worked with immutable data structures such as lists, trees, etc.
  - Cannot directly update them
- What are immutable data structures?
  - It is impossible to change the structure of the list without building a modified copy of that structure
  - Persistent (operations performed on them don’t destroy the original structure)
    - Ex: original structure remains in the stack, even if it becomes inaccessible
  - Implementations using immutable data structures are easier to understand and reason about
- What are mutable data structures?
  - Examples: linked lists, arrays, etc.
  - Update in place and modify and existing structure without rebuilding it
  - Mutable data structures are ephemeral (operations performed on them do modify the original structure)

Linked lists

```
type ‘a rlist = Empty | RCons of ‘a * ‘a rlist ref
let l1 = ref (RCons(4, ref Empty))
```

```
let l2 = ref (RCons, 5, l1))
l1 := !l2;;
```




```
(* observe its output behavior given a linked list l and a bound n
   observe(l, n) will print the first n elements
   If we didn't have the bound n, then observe would print repeatedly
   The elements in a circular list
*)
(* observe: int rlist * int -> unit *)
let rec observe (l, n) = match l with
     | Empty -> print_string "EMPTY\n"
     | RCons(x, xs) ->
          if n = 0 then print_string "STOP\n"
          else print_string (string_of_int x ^ " ");
               observe (!xs, (n-1))
(* rapp : 'a refList * 'a refList -> unit *)
let rec rapp (r1, r2) = match !r1 with
     | Empty -> r1 := !r2
     | RCons (h,t) -> rapp (t, r2)
(* reverse : 'a refList -> 'a refList *);;
let rev l = ref Empty in
     let rec rev' l  = match !l with
          | Empty -> ()
          | RCons (h, t) ->
               (r := RCons (h, ref !r);
               rev' t)
     in
     (rev' l; l := !r)
```

- Note – these can be more efficient than alternatives, but they require more care in controlling where pointers are going (avoid circular data structures)
- We talked about linked lists, but there are other effects (updating memory, printing to screen, and raising exceptions)

Failures and Exceptions

- Primary benefits of exceptions
    - Force you to consider the exceptional case
    - Allows you to segregate the special case from other cases in the code, avoiding clutter

- Diverting control flow

Static vs. Dynamic Violations

- Static violations are signaled by type checking error (before running code)
- Dynamic violations are signaled by raising an exception (when code runs)
- Examples
  - `3/0`
    type int, no value, effect: raises run-time exception Division_by_zero
  - let head_of_empty_list =
    let head (x::t) = x in head[]

Defining your own exceptions

```
exception Domain
let fact n =
     let rec fact' n =
          if n = 0 then 1
          else n * fact' (n-1)
     in
     if n<0 then raise Domain else fact' n
(* alternative to raising Domain in the function as above: *)
let runFact n =
     try
          let r = fact n in
          print_string ("Factorial of " ^ string_of_int n ^ " is
string_of_int r ^ "\n")
     with Domain -> print_string "You should not give us a negative
number"
type key = int
type 'a tree = | Empty | Node of 'a tree * (key * 'a) * 'a tree
let rec find t k =
     | Empty -> None
     | Node(l, (k', d'), r) ->
          if k = k' then Some d'
          else (if k<k' then find l k else find r k)
(* alternative using exceptions*)
exception NotFound
let rec find' t k = match t with
     | Empty -> raise NotFound
     | Node(l, (k', d'), r) ->
          if k = k' then d'
          else (if k<k' then find' l k else find' r k)
(* find an element with the key k in a tree t by looking for the
   element up to depth n *)
exception Error of string
```

```
let rec find'' t k n =
     if n<=0 then raise (Error "Bound exceeded")
     else (match t with
          | Empty -> raise (Error  "Not found in the tree")
          | Node(l, (k', d'), r) ->
               if k = k' then d'
               else (if k<k' then find'' l k else find'' r k))
```

Lecture 11 [10/19/2018]

Exceptions (cont.)

Exceptions: Benefits

- Force you to consider the exceptional case
- Allows you to segregate the special case from other cases in the code, avoiding clutter
- Diverting control flow

Backtracking

- General algorithm for finding all, or some, solutions incrementally
  - Abandons partial candidates as soon as it determines that it cannot lead to a successful solution
- Important tool to solve constraint satisfaction problems such as crosswords, puzzles, Sudoku, etc.

Example: Backtracking Through a Tree

```
(* note – uses implementation of tree, node *)
let rec find t k = match t with
     | Empty -> raise NotFound
     | Node(l, (k', d'), r) ->
          if k = k' then d'
          else (if k<k' then find l k else find r k)
```

- This algorithm isn't exhaustive, it only works with binary trees
- Try to find the key in the left side
  - If you fail, continue on the right side of the tree
- Revised algorithm:

```
let rec find t k = match t with
     | Empty -> raise NotFound
     | Node(l, (k', d'), r) ->
          if k = k' then d'
          else (try find l k with NotFound -> find r k)
let l = Node (Node (Empty, (3, "3"), Empty), (7, "7"), Node (Empty,
(44, "44"), Empty))
let r = Node (Node (Empty, (55, "55"), Empty), (8, "8"), Empty)
let t = Node (l, (50, "50"), r)
# find t 3;;
- : string = "3"
```

- Note – 3 is not the best test case, as no backtracking is needed, 8 is a better test case

```
# find t 8;;
- : string = "8"
# find t 302;;
Exception: NotFound
```

Example: Got Change?

- Implement a function `change` that takes as input a list of available coins and an amount `amt`. It returns the exact change for the amount (i.e. a list of available coins, `[c1; c2; …; cn]` such that `c1+c2+cn=amt`), if possible; otherwise it raises an exception `Change`
- The idea is to proceed greedily, but if we fail (i.e. we get stuck), we "undo" the most recent greedy decision and proceed from there

```
(* change [50; 25; 10; 5; 2; 1] 43 ➔ [25; 10; 5; 2; 1]
   change [5; 2; 1] 13 ➔ [5; 5; 2; 1]
*)
(* change: int list -> int -> int list *)
let rec change coins amt =
     if amt = 0 then []
     else match coins with
          | [] -> raise Change
          | (h::t) ->
               if h<= amt then
                    try [h]::(change (h::t) (amt-h))
                    with Change -> change t amt
               else change t amt
```

Test Case: Traced

```
change [6; 5; 2] 9
→ try 6::change[6; 5; 2] 3 with Change -> change [5; 2] 9
→        change [5; 2] 3
→       change [2] 3
          → try 2::change [2] 1 with Change → change [] 3
                    change [] 1
                    false Change →              raise Change
```

Parsing Arithmetic Expressions

- Step 1: How to define the syntactically well-formed arithmetic expressions?
  - For this, we often use Backus Normal Form (BNF), a notation technique for context-free grammars, often used to describe the syntax of languages
  - Example:

    Final expression E := S;
    S-expression (expression with plus) S := P+S | P-S | P
    P-expression (expression with times) P := A*P | A/P | A
    A-expression (atomic expression) A := n | (S)
    - According to this, the following are well/ill formed expressions:

Well-formed	Ill-formed
3+2-1;	+2
3*2+1;	3+2
(2+3)*4;	(3+

- Step 2: Take an input string and produce the abstract syntax tree for an arithmetic expression
  - Lexer: take an input string and produce a list of tokens

```
type token = SEMICOLON | PLUS | SUB | DIV | TIMES | LPAREN |
RPAREN | INT of int
```

- Parser: take a list of tokens and produce an arithmetic expression following the given grammar

```
type exp = Sum of exp * exp | Minus of exp * exp
           Div of exp * exp | Prod of exp * exp
           Int of int
```

- Example: input to parser = `[INT(5); PLUS; INT(3); SUB; INT(2); SEMICOLON]`
  output to parser = `Sum(Int 5, Minus(Int(3), Int(2))`

Challenge – Parsing Arithmetic Exceptions

- Parsing E-expression
  - Parse a list toklist of tokens into an E-expression s and a remaining list toklist' of tokens that describes what remains
- Parsing S-expression
  - Parse a list toklist of tokens into an S-expression s and a remaining list toklist' of tokens that describes what remains
    exception SExpr of exp * token list
- Parsing P-expression
  - Parse a list toklist of tokens into a P-expression p and a remaining list toklist' of tokens that describes what remains
    exception PExpr of exp * token list
- Parsing A-expression
  - Parse a list toklist of tokens into an A-expression s and a remaining list toklist' of tokens that describes what remains
    exception AtomicExpr of exp * token list

Parsing

- Note that in each of these cases we know what toklist' is supposed to be
- Upon success raise the corresponding exception to propagate the expression built so far and the remaining list of tokens

```
let rec parseExp toklist = match toklist with
     | [] -> raise (Error "Expectetd Expression: Nothing to parse")
     | _ ->
          try parseSExp toklist
          with SEpr (exp, [SEMICOLON] ) -> exp
               | _ -> raise ("Error: Expected Semicolon")
```

Lecture 12 [10/24/2018]

Continuations

What is a continuation?

- Representation of the execution state of a program (for example a call stack) at a certain point in time
- Save the current state of execution into some object and restore the state from this object at a later point in time resuming its execution
    - Gives programmer control
    - Can model this with higher order functions

First-class support for continuations

- C#: async / wait
- Racket: call-with-current-continuation
- Ruby: callcc
- Scala: shift / reset
- Scheme: callcc

Tail-Recursion Revisited

- A function is said to be tail-recursive if there is nothing to do except return the final value
- Since the execution of the function is done, saving its stack frame (where we remember the work we still in general need to do), is redundant
- Can every function be written tail-recursively?
    - Yes, using continuations
    - Not always trivial (when order is preserved, i.e. map, this is difficult)

How to write a function tail-recursively?

- Add an additional argument, a continuation, which acts like an accumulator
- In the base case, we call the continuation
- In the recursive case, we build up the computation that still needs to be done
- A continuation is a stack of functions modelling the call stack, i.e. the work we still need to do upon returning

Example – append

```
(* direct-style append: 'a list -> 'a list -> 'a list *)
let rec append l k = match l with
     | [] -> k
     | h::t -> h::append t k
```

- Not tail recursive, since we use `h::append t k`

```
(* tail-recursive append
   app_tl: 'a list -> 'a list -> ('a list -> 'a list) -> 'a list
*)
let app_tl l k =
     let rec app' l k c = match l with
          | [] -> c k      (* call continuation, feeding it k*)
          | h::t -> app' t k (fun r -> c (h::r))
     in
```

```
    app' l k (fun r -> r)
```

- Note: continuation as a function that takes as input the result of a recursive computation and continues the work it needs to do in the next recursive call
- Is this tail-recursive append method better than the direct one? Is it more efficient?
  - Tested with a function that generates very long lists
  - Generated lists of 4-8 million elements, and tried to append them (using @ = direct)
    - Stack overflow during evaluation (looping recursion?)
  - Generated lists of 4-8 million elements, and tries to append them (using app_tl = tail-recursive)
    - Took a while, but succeeded in appending the lists ➔ stack space has nothing to do in each recursion, difficult to run out of memory (stack space runs out less quickly)
  - Generally better (not as fast always, but at least succeeds with large inputs)
    - Jane St. aimed for this, rewrote all the library functions as such
- What happens when this function is called?

```
app' [1;2] [3;4] (fun r -> r)
app' [2]   [3;4] (fun r1 -> (fun r -> r) (1::r1))
app' []    [3;4] (fun r2 -> (fun r1 -> (fun r -> r) (1::r1)) (2::r2))
```

- At this point, the stack has been built up in each recursive call
- Collapse it by calling the continuation

```
(fun r2 -> (fun r1 -> (fun r -> r) (1::r1)) (2::r2))
(fun r1 -> (fun r -> r) (1::r1)) (2::[3,4])
(fun r -> r) (1::2::[3,4])
(1::2::[3,4])
```

Example – rewrite map tail recursively

```
(* map_tr : 'a list -> ('a -> 'b) -> ('b list -> 'b list) *)
let rec map_tr l f c = match l with
     | [] -> c []
     | h::t -> map_tr t f (fun r -> c ((f h) :: r))
```

Call as (assume l2 to be a list):

```
     map_tr l2 (fun x -> x+1) (fun x -> x)
```

When to use continuations?

- Tail-recursion: the continuation is a functional accumulator
  - Represents the call stack built when recursively calling a function and builds the final result
- Failure continuation: the continuation keeps track of what to do upon failure and defers control to the continuation
- Success continuation: the continuation keeps track of what to do upon success, defers control to the continuation, and builds the final result

Failure continuations

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

```
let leaf n = Node (Empty, n, Empty)
let r = Node (leaf 22, 35, leaf 70)
let ll = Node (leaf 3, 5, leaf 7)
let l = Node (11, 9, leaf 15)
let t = Node (l, 17, r)
(* version 1 – using option types; find : (‘a -> bool) -> ‘a tree ->
‘a option *)
let rec find p t = match t with
     | Empty -> None
     | Node (l, d, r) ->
          if (p d) then Some d
          else (match find p l with
               | None -> find p r
               | Some d’ -> Some d’)
```

- Not efficient, lots of calls regardless of matches

```
(* version 2 – exceptions; find_ex : (‘a -> bool) -> ‘a tree -> ‘a
option *)
exception Fail
let rec find_ex p t = match t with
     | Empty -> raise Fail
     | Node (l, d, r) ->
          if (p d) then Some d
          else (try find_ex p l with Fail -> find_ex p r)
let find’ p t =
     (try find_ex p t with Fail -> None)
(* version 3 – failure continuation; find_count : (‘a -> bool) -> ‘a
tree -> (unit -> ‘a option) -> ‘a option*)
let rec find_fc p t fc = match t with
     | Empty -> fc ()
     | Node (l, d, r) ->
          if (p d) then Some d
          else find_fc p l (fun() -> find_fc p r fc)
let find0 p t = find_fc p t (fun () -> None)
```

Finding all elements satisfying a given property

```
(* version 1 – straightforward recursive program*)
let rec findAll p t = match t with
     | Empty -> []
     | Node (l, d, r) ->
          let el = findAll p l in
          let er = findAll p r in
          if (p d) then el @ (d::er)
          else el @ er
```

Lecture 13 [10/26/2018]

Continuations

When to use continuations?

- Tail-recursion: the continuation is a functional accumulator, representing call stack built when recursively calling function and builds final result
- Failure continuation: keeps track of what to do upon failure and defers control to the continuation
- Success continuation: the continuation keeps track of what to do upon success, defers control to the continuation, and builds the final result

Success Continuations: Collecting all elements in a tree

- Note – to collect all elements in the tree, you really need to traverse the ENTIRE tree

```
(* the tree we are working with *)
let ll = Node (Empty, (3, “3”), Empty)
let lr = Node (Empty, (44, “44”), Empty)
let l = Node (ll, (7, “7”), lr)
let rl = Node (Empty, (55, “55”), Empty)
let r = Node (rl, (8, “8”), Empty)
let t = Node (l, (1, “1”), r)
(* straightforward recursive program, finds them in order - findAll:
     (‘a -> bool) > (‘a * ‘b) tree -> ‘b list *)
let rec findAll p t = match t with
     | Empty -> []
     | Node (l, (k, d), r) ->
          if p k then (findAll p l)@(d::(findAll p r))
          else (findAll p l)@(findAll p r)
# findAll (fun x -> (x mod 2) = 0) t;;
- : string list = [“44”; “8”]
(* using success continuation, finds them in order – findAll’:
     (‘a->bool) -> (‘a * ‘b) tree -> (‘b list -> ‘b list) -> ‘b list
*)
```

- Note – (‘b list -> ‘b list) is a continuation, and combines the elements collected in the left and right sides of the tree
    - Success continuation will know what to do once findAll’ has traversed the left side of the tree

```
let rec findAll’ p t sc = match t with
     | Empty -> sc [] (* call the continuation *)
     | Node (l, (k,d), r) ->
          (if (p k) then (findAll’ p l (fun el -> findAll’ p r (fun er
-> sc (el @ (d::er))))
          else findAll’ p l (fun el -> findAll’ p l (fun el ->
findAll’ p r (fun er -> sc (el@er)))))
(* alternate version *)
```

```
let rec findAll0 p t sc = match t with
     | Empty -> sc []
     | Node(l,d,r) -> findAll0 p l (fun el -> findAll0 p r (fun er ->
if (p d) then sc (el@(d::er)) else sc (el@er)))
(*final version*)
let rec findAll1 p t sc = match t with
  | Empty -> sc []
  | Node(l,d,r) ->
     (if (p d) then
       findAll1 p l (fun el -> el@d::(findAll1 p r sc))
     else
       findAll1 p l (fun el -> el@findAll1 p r sc ))
let r = findAll0 (fun x -> x mod 3 = 0) tt (fun l -> 1);;
```

- Ex: find all values in the tree that are divisible by 3

Regular Expression Matcher

- Singleton – matching a single character
- Alternation – choice between two patterns
- Concatenation – succession of patterns
- Iteration – indefinite repeats
- BNF grammar for regular expressions:
  Regular Expression r ::= a | r1 r2 | 0 | r1 + r2 | 1 | r* , where a represents a single character
- Ex : a(p*)/(e+6)
  - Matches apple, appley, ale, aly
- Ex : g(1+4)(e+a)y
  - Matches grey/gray/gay
    - 1 allows you to skip
- Ex: g(1+0)*(gle)
  - Matches google, gogle, goooogle or ggle
- Ex: b(ob0 + oba) matches boba but not bob

How does this work?

- Let s be a string, or a list of characters
  - Never matches 0
  - Matches 1 iff s is empty
  - S matches a iff s = a
  - S matches a iff s=a
  - S matches r1+r2 iff either s matches r1 or s matches r2
  - S matches r* iff either s is empty or s=s1s2 where s1 matches r and s2 matches r*
- Step 1 – define data type for regular expressions
  - Turning r := a | r1 r2 | 0 | r1 + r2 | 1 | r * into code:

```
type regexp =
Char of char | Time of regexp * regexp | One | Zero | Plus
of regexp * regexp | Star of regexp
```

Regular expression	Code
a(p*)	`Times(Char 'a', Star (Char 'p'))`
l(e+y)	`Times(Char 'l', Plus(Char 'e', Char 'y'))`

- Step 2 implementing the regular expression matcher
  - Turning the requirements above into a function acc: regexp -> char list -> (char list -> bool) -> bool
  - (char list -> bool) is a success continuation which when given a char list continues with the next task on its stack

```
type regexp = Char of char | Times of regexp * regexp | One | Zero |
Plus of regexp * regexp | Star of regexp

(* acc: regexp -> char list -> (char list -> bool) -> bool *)
let rec acc r clist sc = match r with
     | Char c -> (match clist with ->
          | [] -> false
          | c'::clist' -> (c=c') && (sc clist'))
     | Times (r1, r2) -> acc r1 clist (fun clist' -> acc r2 clist' sc)
     | One -> sc clist (* passing to call stack, done for now *)
     | Zero -> false
     | Plus (r1, r2) -> acc r1 clist sc || acc r2 clist sc
     | Star r -> (sc clist) || acc r clist (fun clist' -> acc (Star r)
clist' sc)
```

- Functions such as the ones below are used to turn strings into lists of characters, making it easier to parse regular expressions

```
let rec tabulate f n =
     let rec tab n acc =
          if n=0 then (f 0)::acc
          else tab (n-1) ((f n)::acc)
     in
     tab n[]
let string_explode s = tabulate (fun n -> String.get n)
((String.length s) - 1)
let string_implode l = List.fold_right (fun c s -> Char.escaped c ^ s)
l ""
let accept r clist = acc r (string_explode clist) (fun l -> l = [])
```

Lecture 14 [10/31/2018]

Lazy Programming

Eager vs. Lazy Programming

- Eager evaluation
  - Evaluate expressions by call-by-value
  - Variables are bound to values
- Concretely, for example:

```
let x = 3+2 in x * 2
```

  - Evaluates expression 3+2 to the value 5
  - Evaluate expression x*2 in the environment where variable x is bound to the value 5 to the final value 10
- Example:

```
let x = horribleComp (345) in 5
```

  - Uselessly evaluates potentially time-consuming horribleComp (345) to some value, but evaluates the expression to and then returns 5

Partial Evaluation

```
let test x y =
     let w = horribleComp x in
     w + y
```

- Imagine that we want to run test on varying inputs- it happens that the values for x vary much less frequently than the inputs for y
- Let's partially evaluate test 10 and test 5

```
# let test10 = test 10;;
val test10 : int -> int = <fun>
# let test5 = test 5;;
val test5 : int -> int = <fun>
```

- Will it be more efficient to compute test10 5 instead of test 10 5?
  - No – test 10 evaluates to fun y -> let w = horribleComp 10 in w + y
  - Remember:
    - Functions are first-class values
    - We never evaluate inside a function
    - We only evaluate the body of a function when we call it
  - How can we make it more efficient? Delay when part of it occurs

Partial Evaluation Done Correctly

```
let test x =
     let w = horribleComp x in
     fun y -> 2 + y
```

New Function:

```
#let test10 = test10;;
Val test10 : int -> int = <fun>
```

- Will this new function be more efficient to compute?
  - Yes, since:

`test 10` evaluates to
`let w = horribleComp 10 in fun y -> w + y` evaluates to
`let w = 772 in fun y -> w + y` evaluates to
`fun y -> 772 + y`

- Functions block evaluations
- You never evaluate inside bodies
- Staging → do one thing first (in this case, test 10), then run on second set of inputs (process in stages)

Unusual Effectiveness of Partial Evaluation

- Partially evaluating functions can be effective
- We must control when we create functions and closures

Eager Computation

- Easier to reason about
- Clear when evaluation occurs
- May evaluate expressions that are never needed

Lazy Computation

- Concretely for example – let x
- Bind variables to unevaluated expressions (not values)
- Suspend computation horribleComp (345)
- Memorize results
- Demand-driven
- Harder to reason about but very useful for
  - Infinite data (ex. All prime numbers)
  - Interactive data (ex. For example sequence or stream of inputs)

Finite Data

```
type 'a list = Nil | Cons of 'a * 'a list
```

- Encodes an inductive definition of finite lists
  - Nil is a list of type 'a list
  - If x is of type 'a and xs is a list of type 'a list, then cons (x,xs) is a list of type 'a list
  - NOTHING else is a list
- How do we take apart lists? Pattern matching
- How do we reason with lists? Induction on the structure of lists

`Infinite Data`

- Infinite data is defined by the observations about it, rather than how it is constructed
- Given a stream 1, 2, 3, 4, 5, … we can ask for
  - Head of stream → 1
  - Tail of stream → 2, 3, 4, 5, …
  - This does not terminate, and a program may run forever. On the other hand, the program, if well written, will remain productive, so that we can make an observation at each step, thus eliminating the need for it to run for every possible result

```
type ‘a susp = Susp of unit -> ‘a
let force (Susp f) = f ()
let x = Susp(fun () -> horribleComp(345)) in force x + force x
```

- Note that in this case, susp and force control the evaluation of a function, by suspending it (i.e. evaluates first step, then stops) until it is forced (called upon) to continue
    - This is efficient because it only runs the program when the information is needed

Streams

```
let ‘a str = {hd : ‘a ; tl : (‘a str)_ sup}
```

- Encodes a coninductive definition of infinite streams using the two observations: hd and tl
    - Asking for the head using the observation hd returns an element of type ‘a
    - Asking for the tail using the observation tl returns a suspended stream of type (‘a str) susp
    - If more specific elements are required, you have to ask for more (using force)

Examples

```
type ‘a susp = Susp of (unit -> ‘a)
let rec ones = {hd = 1 ; tl = Susp (fun() -> ones)}
# val ones : int str = {hd=1, tl=Susp<fun>}
```

```
let rec take n s = if n = 0 then [] else ..hd :: take (n-1) s.tl)
val take : ‘a -> ‘b
```

```
let rec numsFrom n = { hd = n; tl = Susp( fun () -> numsFrom (n+1) }
let no_divider m n = not (n mod m = 0)
let rec sieve s =
     {hd = s.hd;
     tl = Susp (fun () -> sieve (sfilter (no_divider s.hd) (force s.tl
     )))}
# take 10 (sieve (numsFrom 2));;
- : int list = [2; 3; 4; 7; 11; 13; 17; 19; 23; 29]
```

Lecture 15 [11/2/2018]

Lazy Programming, continued

Finite Objects, continued

- Processing finite objects lazily is useful when it corresponds to demand driving computations

```
type 'a lazy_list = {hd = 'a; tl = 'a fin_list susp}
fin_list = Empty | Nonempty of 'a lazy_list
```

- Example: take

```
let rec take nl = match n with
     | 0 -> []
     | n -> l.hd::take'(n-1)(force l.tl)
let take' n l = match l with
     | Empty -> []
     | Nonempty l' -> take n l'
```

- Example: natsTo – natural numbers up to n

```
let rec natsTo =
     { hd = n; tl = Susp (fun() -> natsTo' (n-1))}
let natsTo' n = if n<0 then Empty else Nonempty (natsTo n)
```

- Example: map

```
let rec map f ll =
     {hd = f ll.hd; tl = Susp (fun() -> map' f (force ll.tl))}
let map' f fl = match fl with
     | Empty -> Empty
     | Nonempty ll -> Nonempty (map f ll)
```

How can you suspend and prevent evaluation of an expression?

- Functions! See last lecture: susp, force, streams

```
fun() -> 3 + 2 * 10
```

- This function only evaluates when you pass it
- It evaluates to itself: fun() -> 23 X

Example: power series

  - K=2 → 1, 2, 4, 8, 16, …
  - K=3 → 1, 3, 9, 27, …
  - K=5 → 1, 5, 25, …

- In the power series example, the i-th element in the series is K^i
  - To get the first element, set i=0
- This would be defined:

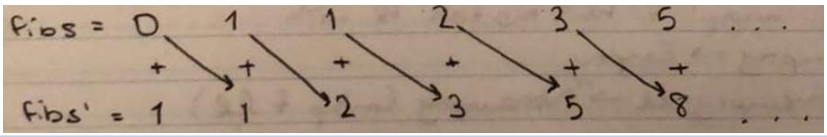
```
let rec power_series n k =
     { hd = n; tl = Susp(fun() -> power_series(n*k) k)}
```

Example: geometric series

```
let rec geometric_series n =
     { hd = 1.0 /. n; tl = Susp (fun()->geometric_series (n *. 2.0))}
```

Example: Fibonacci numbers

- The Fibonacci sequence is the series of numbers 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, … where each number is found by adding up the two numbers before it
- Idea: use two streams
  - fibs = 0, 1, 1, 2, 3, 5, …
  - fibs' = 1, 1, 2, 3, 5, 8, …
- Using the addStreams function, we can get the next Fibonacci number by adding the two streams:




```
let rec addStreams s1 s2 =
     { hd = s1.hd + s2.hd; tl = Susp(fun()-> addStreams (force s1.tl)
     force s2.tl))}
let rec fibs = {hd = 0; tl = Susp(fun() -> fibs')}
let rec fibs' = {hd = 1; tl = Susp(fun() -> add fibs fibs')}
```

Lecture 16 [11/7/2018]

Modules

Modules for Data Abstraction: Primary Benefits

- Control complexity of developing and maintaining software
- Split large programs into separate pieces
- Name space separation
  - Don't have to come up with new names constantly
- Allows for separate compilation
- Incremental development
- Clear specifications at module boundaries
- Programs are easier to maintain and reuse
- Enforces abstractions
- Isolates bugs
- Etc.

Modelling Stacks

```
module type STACK =
sig
     type el
     type stack
     val empty : unit -> stack
     val is_empty : stack -> bool
     val pop : stack -> stack option
     val push : el -> stack -> stack
     val top : stack -> el option
end
```

```
module Stack : STACK =          (* implementation of module Stack
                                implements the template from module
                                type STACK *)
     struct
          type el = int
          type stack = int list
          let empty () : int list
          let empty () : stack = []
          let push i (s : stack) : stack = i::s
          let is_empty s = match s with
               | [] -> true
               | _::_ -> false
          let pop s = match s with
               | [] -> None
               | _::t -> Some t
          let top s = match s with
```

```
            | [] -> None
            | h::_ -> Some h
        let rec length s acc = match s with
            | [] -> acc
            | x::t -> length t 1+acc
        let size s = length s 0
        let stack2list (s:stack) = s
    end
```

- Might want to capture functionality of stack in module:
  - Different lengths, etc.
- Similar to interfaces: module type
  - Declares signature for module
  - Compiler checks that everything whose signature is given in the module type is implemented in a module of that type
  - Order that things are implemented does not matter
  - Should be precise, but not fully general (idea is to hide some things)
    - Things in the module, not the module type cannot be externally accessed → essentially private variables/functions
- In the above example, STACK is the module type, Stack is a module of type STACK
- When does a module implement a given module type?
  - Flexibility → order doesn't need to be the same
    - Provide required functionality in any order
  - Minimize bureaucracy → you can implement more functions than those that are required by the module type
    - In the Stack example, Stack has length, size, stack2list that are not required in the module type
    - Hides information from someone using that module structure
      - Encapsulated within module, not visible in the type
      - Isolates bugs, functionality, etc.
  - Enhance reuse → a module may provide values with a more general type than is required
  - Avoid overspecification → a data type may be provided whenever a type is required

```
# let s = Stack.empty ();;
val s : Stack.stack = <abstr>
```

- In the module type, the type of type stack is not defined, thus it is abstract

```
# Stack.is_empty s;;
- : bool = true
# Stack.is_empty [];;
    Error: This expression has type 'a list but an expression was
    expected of type Stack.stack
```

- Despite the fact that Stack is implemented to use lists, this is concealed as it isn't revealed in the module type, therefore this doesn't work (is concealed, otherwise would break the abstraction)
    - Allows developer to change their mind (alter implementation, i.e. represent stacks with trees rather than lists)

```
# Stack.length;;
     Error: Unbound value Stack.length
```

- Cannot use it, as it isn't in the module type

```
# length s;;
     Error: This expression has type Stack.stack but an expression was
     expected of type 'a list
```

- Even though s is a stack implemented using a list, list functions do not affect it (the fact that it is really a list is concealed)

```
# let s = Stack.empty ();;
val s : Stack.stack = <abstr>
# Staack.is_empty s;;
- : bool = true
```

- How do you push something onto the stack, given you don't know what type the element is?
    - Stack needs to be populated with some elements → specialize in some way the type of elements (instance of this module type?)

```
module Stack : (STACK with type el = int) =
struct
     type el = int
     type stack = int list
     …
     let size s = length s 0
     let stack2list (s:stack) = s
end
```

- Note: now it is known to the public that el is of type int → now can push to stack/use other functions that require knowledge of what el's type is

```
# let s1 = Stack.push 1 s
val s1 : Stack.stack = <abstr>
# let s2 = Stack.push 2 s1;;
val s2 : Stack.stack = <abstr>
# let e2 = Stack.top s2
val e2 : Stack.el option = Some 2
# module S = Stack;;
module S = Stack
# S.empty;;
- : unit -> S.stack = <fun>
# S.pop;;
```

```
- : S.stack -> S.stack option = <fun>
# open S;;
```

- Note: Dangerous
    - All abstractions are gone

```
# empty;;
- : unit -> S.stack = <fun>
# open Stack;;
# empty;;
- : unit -> Stack.stack = <fun>
```

- Note : cannot close them again

Bank Example

```
module type CURRENCY =
sig
     type t
     val unit : t
     val plus : t -> t -> t
     val prod : float -> t -> t
     val toString : t -> string
end;;
```

```
module Float =
struct
     type t = float
     let plus = (+.)
     let prod = (*.)
     let unit = 1.0
     let toString x = string_of_float x
end;;
```

```
module Euro = (Float : CURRENCY);;
module CAD = (Float : CURRENCY);;
module USD = (Float : CURRENCY);;
```

- Note that all three currency modules (Euro, CAD, USD) have the same interface

```
let euro x = Euro.prod x Euro.unit;;
let usd x = USD.prod x USD.unit;;
let can x = CAN.prod x CAN.unit;;
# let x = Euro.plus (euro 10.0) (euro 20.0);;
val x : Euro.t = <abstr>
# Euro.toString x;;
- : string = “30.”
# let w = USD.plus (usd 5.0) (usd 7.0);;
val w : USD.t = <abstr>
```

```
# USD.toString w;;
- : string = “12.”
# Euro.plus x w
     Error: This expression has type USD.t but an expression was
     expected of this type
```

- Note – despite same interface, USD and Euro cannot be properly combined

```
module type Client =
sig
     type t (*account*)
     type currency
     val deposit : t -> currency -> currency
     val retrieve: t-> currency -> currency
     val print_balance : t -> string
end
```

- In the module type below, BANK inherits all values declared in CLIENT (include CLIENT), so if we declare names already declared there, we overshadow them

```
module type BANK =
sig
     include CLIENT
     val create : unit -> t
end;;
```

- In the Old_Bank example below, all information lies in the individual account, so it would be a poor setup for an actual bank, as the there are no records in the event that an error occurs, making it possible for the client to lose money
    - On the other hand, it demonstrates benefits of modules, as the client and banker have different views (most information is concealed for the banker to view, not the client, ex: the banker needs to create new accounts, the client cannot)

```
module Old_Bank (M : CURRENCY)_ : (BANK with type currency = M.t)=
stuct
     type currency = M.t
     type t = {mutable balance : currency}
     let zero = M.prod 0.0 M.unit
     and neg = M.prod (-1.0)
     let create() = {balance = zero}
     let deposit c x =
          if x>zero then
               c.balance <- M.plus c.balance x;
               c.balance
     let retrieve c x =
          if c.balance>x then
               deposit c (neg x)
          else
```

```
                c.balance
end;;
```

- The code below illustrates a few examples of inheritance and code reuse

```
module Post = Old_bank (Euro);;
module eTD = Old_Bank (CAN);;
module Post_Client : (CLIENT with type currency = Post.currency and
type t = Post.t) =
     Post : client
```

Lecture 17 [11/9/2018]

Intro to Programming Languages

- NOTE – due to the pace of this and the next few lectures, as well as the less typical symbols used, I was unable to properly transcribe every step of expression evaluation. As such, I recommend reviewing the posted slides on Prof. Pientka's page (as they will have anything I did not copy down) for a more detailed approach. Additionally, I abbreviated some symbols later for easier transcription, so the equivalents she used can also be found there.

Introduction

- What are syntactically legal expressions? Grammar
  - Expressions the parser accepts
- What are well-typed expressions? Static semantics
  - What expressions does the type-checker accept?
- How is an expression executed? Dynamic semantics

Syntactically legal expressions

- The set of expressions is defined inductively by:
  - A number *n* is an expression
  - The Booleans true and false are expressions
  - If e1 and e2 are expressions, then e1 op e2 is an expression where op = { +, -, =, *, <}
  - If e1 and e2 are expressions, then if e then e1 else e2 is an expression
- This is represented in Backus-Naur Form

How to implement expressions in Ocaml

- Backus-Naur Form (BNF) –
  - Operations op := + | - | * | < | =
  - Expressions e := n | e1 op e2 | true | false | if e then e1 else e2
- Representation in OCaml

```
type primop = Equals | LessThan | Plus | Minus | Times
type exp =
      | Int of int
      | Bool of bool
      | If off exp * exp * exp
      | Primop of primop * exp list
```

- Example – If 3<0 then 1 else 0 is represented as

```
if (Primop (LessThan, [Int 3, Int 0]), Int 1, Int 0)
```

How to evaluate an expression

- A better question: how to describe evaluation of expressions
- Example
  - We want to say: Expression e evaluates to a value v
  - What are values?
    - Numbers and Booleans
      Values v := n | true | false

- Definition: evaluation of the expression e to a value v is defined inductively by the following clauses
  - A value v evaluates to itself
  - If expression e evaluates to the value true and expression e1 evaluates to a value v, then if e then e1 else e2 evaluates to the value v
  - If expression e evaluates to the value false and expression e2 evaluates to a value v, then if e then e1 else e2 evaluates to the value v
- This definition is very verbose → we need some better, more compact notation

Turning an Informal Description into a Formal One

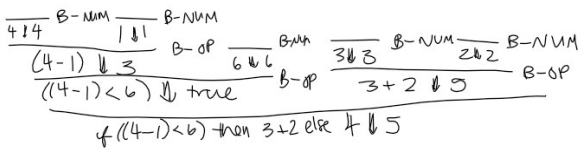
- Step 1 –
  - Let's write e ↓ v for expression e evaluates to value v
  - Definition: e ↓ v is defined inductively by the following clauses
    - v ↓ v
    - if e ↓ true and e1 ↓ v then if e then e1 else e2 ↓ v
    - if e ↓ false and e2 ↓ v then if e then e1 else e2 ↓ v
- Step 2 –
  - $\frac{premise1 \ldots premise(n)}{conlusion}\, name$
    - If premise1 and premise2 and … and … premise(n) then conclusion
  - $\frac{}{v \downarrow v}\, B-VAL$ $\qquad \frac{e \downarrow \text{true}\ e1 \downarrow v}{if\ e\ then\ e1\ else\ e2 \downarrow v}\, B-IFTRUE$ $\qquad \frac{e \downarrow \text{false}\ e2 \downarrow v}{if\ e\ then\ e1\ else\ e2 \downarrow v}\, B-IFFALSE$
  - Evaluation rules do not impose an order on the premises (purely declarative)

Extending It to Primitive Operator Example

- Definition: e ↓ v is defined inductively by the following clauses
  - v ↓ v
  - if e ↓ true and e1 ↓ v, then if e then e1 else e2 ↓ v
  - if e ↓ false and e2 ↓ v, then if e then e1 else e2 ↓ v
  - if e ↓ v1 and e2 ↓ v2, then e1 op e2 ↓ v where v = v1 op v2

What Does This Mean in Practice?

- Read it operationally
  - Evaluating if ((4-1)<6) then 3+2 else 4 returns 5



- The derivation tree example essentially describes the execution of a recursive program which computes 5 from the input if ((4-1)<6) then 3+2 else 4

Dynamic Semantics as a Recursive Program

- Dynamic semantics: e ↓ v
- A declarative description on how to evaluate an expression e
- Task: implement a function eval that does what e ↓ v describes
- Extending it to primary operators

- E ↓v is defined inductively by the following:

Example

```
module Exp =
struct
     type name = string
     type primop = Equals | LessThan | Plus | Minus | Times | Negate
     type exp =
          |Int of int
          |Var of name
          |Bool of bool
```

(etc., see remainder of cases in posted code)

```
     and dec =
          |val of exp * name
```

```
module Eval =
     struct
          open Exp
          let evalOp op = match op with
               | (Equals, [Int i; Int i']) -> Some (Bool (i=i'))
               | (LessThan, [Int i; Int i') -> Some Bool (i<i1)
               | (Plus, [Int i, Int i') -> Some (Int (i+i'))
               | (Minus, [Int I, Int I']) -> Some (Int (i-i'))
               | (Times, [Int I, Int I']) -> Some (Int (i*I'))
               | (Negate, [Int i]) -> Some (Int (-i))
               | _ -> None
          let rec eval e = match e with
               | Int _ -> e
               | Bool _ -> e
               | If (e, e1, e2) -> (match eval e with
                    | Bool true -> eval e1
                    | Bool false -> eval e2
                    | _ -> raise (Stuck "guard is not a bool")
               | Primop (op, elist) ->
                    let vl = List.map eval elist in
                    (match evalOp (op,vl) with
                         | None -> raise (Stuck "Illegal Use of
          operator"
                         | Some v -> v)
     end
module E = Exp
let e1 = E.If (E.Primop (E.Equals, [E.Int 3; E.Int 2]),
              E.Primop (E.Plus, [E.Int 5; E.Primop (E.Times, [E.Int 3;
```

```
                E.Int 5])])
              E.Primop(E.Plus, [E.Int 1; E.Primop(E.Times, [E.Int 3;
                E.Int 5])]))
```

Similar logic follows in the posted code for e2, e3, and e4

Advantages of a formal description

- Establish properties
- Guarantee
  - Coverage -→ for all expressions e there exists an evaluation rule
  - Determinacy → if e ↓ v1 and e ↓ v2

How to Describe Well-Typed Expressions

- How can we statically check whether an expression would potentially lead to a runtime error?
- Static type checking
  - Types approximate runtime behavior
  - Lightweight tool for reasoning about programs
  - Detect errors statically, early in the development cycle
  - Great for code maintenance
  - Precise error messages
  - Checkable documentation of code

Basic Types

- Types classify expressions according to the kinds of values they compute
  - Statically determine possible value an expression might return
- What are values
  - Values v ::= n | true | false
- Hence, there are two basic types
  - Types T ::= int | bool.

Define Typing

- e : T expression e has type T
- We define when an expression e is well-typed inductively
- e:T is defined inductively by the following clauses
  - n : int
  - true : bool and false : bool
  - if e : bool and e1 : T and e2 : T, then if e then e1 else e2 : T
  - if e1 : int and e2 : int, then e1 + e2 : int
  - if e1 : int and e2 : int then e1 = e2 : bool

Two readings of typing

- Type checking e : T
  - Given the expression e and the type T, we check that e does have type T
- Type inference e : T
  - Given the expression e, we infer its type T
  - Much harder than type checking

Implementing Type Interface

```
module Types =
     struct
     module E = Exp
     type tp = Int | Bool
     let typ_to_string t = match t with
          | Int -> “Int”
          | Bool -> “Bool”
     exception TypeError of string
     let fail message = raise (TypeError message)
     let primopType p = match p with
          | E.equals -> ([Int; Int], Bool)
          | E.LessThan -> ([Int; Int], Bool)
          | E.Plus -> ([Int; Int], Int)
          | E.Minus -> ([Int; Int], Int)
          | E.Times -> ([Int; Int], Int)
          | E.Negate -> ([Int], Int)
     let rec infer e = match e with
          | E.Bool b -> Bool
          | E.Int _ -> Int
          | E.If(e, e1, e2) -> (match infer e with
               | Bool ->  let t1 = infer e1
                          let t2 = infer e2 in
                          (if t1 = t2 then t1 else raise (Error
     “msg”))
               | _ -> raise (Error “???”))
          |E.Primop(po, args) ->
               let (expected_arg_types; resultTyppe) = primopType po
in
               let inferred_arg_types = List.map infer aregs in
               let rec compare tlist1 tlist2 = match tlist1 tlist2
with
                    |[], [] -> resultTyppe
                    |t::tlist, s::slist ->
                         if t=s then compare tlist slist
                         else fail (“Expected” ^ typ_to_string t ^ “
-Inferred” ^ typ_to_string s)
                    |_,_ -> fail (“Error: primitive operator”)
               in
               compare expected_arg_types inferred_arg_types
     end
```

Lecture 18 [11/14/2018]

Key Questions

- What are the syntactically legal expressions?
  - Grammar
  - i.e. What expressions does the parser accept?
- What are well-typed expressions?
  - Static semantics
  - i.e. What expressions does the type-checker accept?
- How is an expression executed?
  - Dynamic semantics
- Today: expand on this with variables and let-expressions

Adding Variables and Let-Expressions

- BNF Expressions can now be defined as:
  Expressions e ::= n | e1 op e2 | true | false | if e then e1 else e2 | x | let x = e1 in e2 end
- Well-formed expressions:
  - 3 + (2 + 4)
  - true + (2 + 4)
  - if (2 = 0) then 5 + 3 else 2
  - if true then (if false then 5 else 1 + 3) else 2 = 5
  - let z = if true then 2 else 43 in z + 123 end
  - let z = if 7 then 2 else 43 in z + false end
  - let x = x + 3 in y + 123 end
  - let x = x + 3 in x + 123 end
  - Note that these examples make sense syntactically, but don't necessarily type check
- Ill-formed expressions, according to grammar defined previously:
  - If true then 2 else
  - -4
  - +23
  - let z = if true then 2 else 43 in -123 end
  - let x = 3 in x
  - let x = 3 x + 2 end
- It is important to focus on the scope of variables, and note that their names should not matter

Free variables

- Unused elsewhere in the program, not "bound"
  - May become bound in the let-expression
  - Closed expressions have no free variables
- The following code computes the set of free variables in an expression

```
let rec freeVars e = match e with
     | Var y -> [y]
     | Int n -> []
```

```
    | Bool b -> []
    | If (e, e2, e2) -> union (freeVars e, union (freeVars e1,
freeVars e2))
    | Primop (po, args) -> List.fold_right (fun e1 fv -> union
(freeVars e1, fv)) args []
    | Let (Val (e1, x), e2) -> union (freeVars e1, delete ([x],
freeVars e2)
```

Variable names shouldn't matter

- i.e. let x = 5 in (let y = x + 3 in y + y end) end is equivalent to let x = 5 in (let x = x + 3 in x + x end) end which is equivalent to let v = 5 in (let w = v+ 3 in w + w end) end
- As such, variables can be renamed… a process known as substitution

Free Variables, Continued

- FV(e) returns the set of free variable names occurring in the expression e, and is defined inductively based on the structure of expression e (see posted code for specifics)
- When is a variable free, and when is it bound?
- Example:
  let x = 5 in (let x = x + 3 in x + x end) end
  - x is free → bound in the expression by x = 5
  - x is free → bound in the expression by x = x + 3
- Examples of computing free variables:
  - FV(x) = {x}
  - FV(e1 op e2) = FV(e1) union FV(e2)
  - FV(if e then e1 else e2) = FV(e) union FV(e1) union FV(e2)
  - FV(let x = e1 in e2 end) = FV(e1) union (FV(e2)/{x})

Substitution

- Defined as: [e'/x]e
  - Replace all free occurrences of the variable x in the expression e with expression e'
- As a function: input e', x, e; output: an expression where all free occurrences of x in e have been replaced by e'
- Defined by considering different cases for e

$$
\begin{aligned}
&[e'/x](x) &&= e' \\
&[e'/x](e_1 \text{ op } e_2) &&= [e'/x]e_1 \text{ op } [e'/x]e_2 \\
&[e'/x](\texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2) &&= \texttt{if } [e'/x]e \texttt{ then } [e'/x]e_1 \texttt{ else } [e'/x]e_2 \\
&[e'/x](\texttt{let } y = e_1 \texttt{ in } e_2 \texttt{ end}) &&=
\end{aligned}
$$

Example:

$$
\begin{aligned}
&[5/x](\texttt{let } y = x + 3 \texttt{ in } y + x \texttt{ end}) = \\
&\texttt{let } y = [5/x](x + 3) \texttt{ in } [5/x](y + x) \texttt{ end} = \\
&\texttt{let } y = 5 + 3 \texttt{ in } y + 5 \texttt{ end}
\end{aligned}
$$

- This appears to suggest: [e'/x](let y = e1 in e2 end) = let y = [e'/x]e1 in [e'/x] e2 end
  - Attempt with another example:

$[y+1/x](\texttt{let } y = x+3 \texttt{ in } y+x \texttt{ end})$
$\texttt{let } y = [y+1/x](x+3) \texttt{ in } [y+1/x](y+x) \texttt{ end}$
$\texttt{let } y = (y+1)+3 \texttt{ in } y+(y+1) \texttt{ end}$

- This shows that as logical as the suggestion above seems, y gets captured → was free, now is bound, thus the suggestion is false

- More Examples

Example: (where we rename the bound variable y to w.)

$$\begin{aligned} & [y+1/x](\texttt{let } w = x+3 \texttt{ in } w+x \texttt{ end}) \\ = \; & \texttt{let } w = [y+1/x](x+3) \texttt{ in } [y+1/x](w+x) \texttt{ end} \\ = \; & \texttt{let } w = (y+1)+3 \texttt{ in } w+(y+1) \texttt{ end} \end{aligned}$$

Substitution isn't stable under renaming of bound variable names, since before renaming we got:

$$\texttt{let } y = (y+1)+3 \texttt{ in } y+(y+1) \texttt{ end}$$

Capture-Avoiding Substitution

- Let us redefine substitution to avoid this type of conflict:

$$\begin{aligned} [e'/x](x) &= e' \\ [e'/x](e_1 \text{ op } e_2) &= [e'/x]e_1 \text{ op } [e'/x]e_2 \\ [e'/x](\texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2) &= \texttt{if } [e'/x]e \texttt{ then } [e'/x]e_1 \texttt{ else } [e'/x]e_2 \\ [e'/x](\texttt{let } y = e_1 \texttt{ in } e_2 \texttt{ end}) &= \texttt{let } y = [e'/x]e_1 \texttt{ in } [e'/x]e_2 \texttt{ end} \\ & \qquad \text{provided } x \neq y \text{ and } y \notin \mathrm{FV}(e') \end{aligned}$$

OCaml implementation of BNF with let-expressions and variables

```
type name = string
type primop = Equals | LessThan | Plus | Minus | Times
type exp =
     | Var of name
     | Int of int
     | Bool of bool
     | If of exp * exp * exp
     | Primop of primop * exp list
     | Let of dec * exp
and dec = Val of exp * name
```

Evaluation, Revisited

Operations op ::= $+ \mid - \mid * \mid < \mid =$
Expressions e ::= $n \mid e_1 \text{ op } e_2 \mid \texttt{true} \mid \texttt{false} \mid \texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2$
$x \mid \texttt{fn } x \texttt{ => } e \mid e_1\ e_2 \mid \texttt{let } x = e_1 \texttt{ in } e_2 \texttt{ end}$
Values v ::= $n \mid \texttt{true} \mid \texttt{false} \mid \texttt{fn } x \texttt{ => } e$

$$\frac{}{v \Downarrow v}\ \text{B-VAL}$$

$$\frac{e \Downarrow \texttt{true} \quad e_1 \Downarrow v}{\texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2 \Downarrow v}\ \text{B-IFT} \qquad \frac{e \Downarrow \texttt{false} \quad e_2 \Downarrow v}{\texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2 \Downarrow v}\ \text{B-IFF}$$

$$\frac{e_1 \Downarrow v_1 \quad [v_1/x]e_2 \Downarrow v}{\texttt{let } x = e_1 \texttt{ in } e_2 \texttt{ end} \Downarrow v}\ \text{B-LET}$$

$$\frac{e_1 \Downarrow \texttt{fn } x \texttt{ => } e \quad e_2 \Downarrow v_2 \quad [v_2/x]e \Downarrow v}{e_1\ e_2 \Downarrow v}\ \text{B-APP}$$

- Note – down arrow → evaluates to (over multiple steps, potentially)

Lecture 19 [11/16/2018]

Static Type Checking

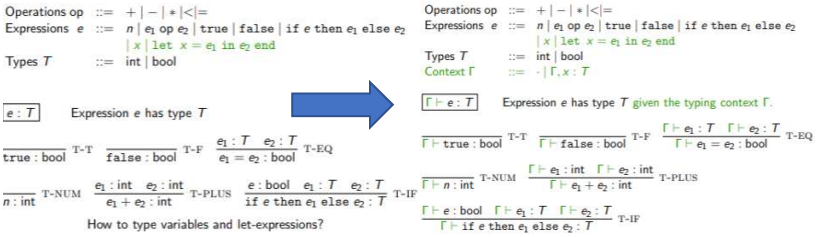
- Types approximate runtime behavior
- Lightweight tool for reasoning about programs
- Detect errors statically, early in the development cycle
- Great for code maintenance
- Precise error messages
- Checkable documentation of code

Typing in Context

- Let's reconsider the assessment e : T expression e has type T
- The expression let x = e1 in e2 end has type S if
  - e1 has type T
  - Assuming x has type T, show that e2 has type T
- Or, generally:
  - Given assumptions x1: T1, …, xn: Tn (these assumptions being known as the typing context r), expression e has type T

Mini Language

- Revisits language started previously, see minilang.ml (posted) for specific implementations



Reading These Rules, Revisited

- Using T-IF from the above right image:
  - Expression if e then e1 else e2 has type T given the typing context gamma if
    - Expression e has type bool given the typing context gamma
    - Expression e1 has type T given the typing context gamma
    - Expression e2 has type T given the typing context gamma

$$\frac{\Gamma(x) = T}{\Gamma \vdash x : T}\ \text{T-VAR} \qquad \frac{\Gamma \vdash e_1 : T_1 \quad \Gamma, x{:}T_1 \vdash e_2 : T}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2\ \texttt{end} : T}\ \text{T-LET}\ \ x \text{ must be new}$$

- Collecting and using assumptions, using T-VAR and T-LET (as defined above)
  - T-VAR: variable x has type T in a typing context gamma, if there exists a declaration x : T in gamma
  - T-LET: expression let x = e1 in e2 has type T in typing context gamma if:
    - Expression e1 has type T1 in the context gamma
    - Expression e2 has type T in the extended context gamma, x : T1

Two types of reading when typing

- Type checking gamma t e : T
  - Given assumptions gamma, the expression e and the type T: check that e does have type T
- Type inference gamma t e : T
  - Given the expression e and typing context gamma: infer its type T
- Thus, generalize implementation of type inference:

```
type tp = Int | Bool
type ctx = (E.name * tp) list
type rec infer gamma e = …
```

Generalizing to functions and their application

$$\frac{\Gamma, x{:}T_1 \vdash e : T_2}{\Gamma \vdash \texttt{fn}\ x \texttt{=>}\ e : T_1 \to T_2}\ \text{T-FN} \qquad \frac{\Gamma \vdash e_1 : T_2 \to T \quad \Gamma \vdash e_2 : T_2}{\Gamma \vdash e_1\ e_2 : T}\ \text{T-APP}$$

- T-FN can be read: expression fn x => e has type T1 → T2 in a typing context gamma if expression e has type T2 in the extended context gamma, x : T1
- T-APP can be read: expression e1 e2 has type T in a typing context gamma, if expression e1 has type T2 → T in the context gamma and expression e2 has type T2 in the context gamma

Lecture 20 [11/21/2018]

- Fill out course evaluations and LearnOCaml survey
  - LearnOCaml survey – 75% participation gets everyone a bonus point
    - About platform, not specific homework questions
- COMP 302 has experienced massive growth since 2006, resources don't reflect that

Typing, Continued (Static Semantics)

Generalizing to functions and function application

$$\frac{\Gamma, x{:}T_1 \vdash e : T_2}{\Gamma \vdash \texttt{fn}\ x \texttt{=>}\ e : T_1 \to T_2}\ \text{T-FN} \qquad \frac{\Gamma \vdash e_1 : T_2 \to T \quad \Gamma \vdash e_2 : T_2}{\Gamma \vdash e_1\ e_2 : T}\ \text{T-APP}$$

- Read T-FN rule as expression fn x → e has type T1 → T2 in a typing context gamma, if expression e has type T2 in the extended context gamma, x : T1
  - Cannot be used for type inference directly because the type of x is unclear
  - Where is T1 coming from?
    - Use a type variable for describing an unknown type (Damas-Hindley-Milner type inference)

Type Variables: Two Different Views

- Types T ::= int | bool | T1 → T2 | T1 x T2 | α
- Gamma
- View A: Type Checking
  - Are all substitution instances of e well typed? That is for every type substitution sigma, we have [sigma]gamma t e : [sigma] T
  - Examples
    - t fn x → x has type α → α
    - t fn f → fn x → f(f(x)) has type (α→ α) → α → α
    - x : α t fn f → f x has type (α → beta) → beta
- View B: Type Inference
  - Is some substitution instance of e well-typed? That is we can find a a type substitution sigma, such that [sigma]gamma t e : [sigma] T
  - Examples
    - t fn x → x + 1 has type α→ α
      - choosing int for α (i.e. int/ α)
    - fn x → x + 1 has typte α → beta
      - choosing int for α
      - choosing int for beta (i.e. int/ α int/beta)
    - x: α t fn f → f x has type beta → gamma
      - choosing (α → gamma) for beta

Substitution: the MOST general one is inferred

- x : α t fn f → f x has type beta → gamma, choosing (α → gamma) for beta
  - i.e. (α → gamma)/beta
- what about choosing int/ α, (int → gamma)/beta
  - fn f → f x has type (int → gamma) → gamma under the assumption x:int
  - However, this is not the most general option

Damas-Hindley-Milner Style Type Inference

- Given a typing context gamma and an expression e, infer a type T (and some constraints) as follows
  - Analyze e as before following the given typing rules
  - When we analyze e recursively and we miss type information, introduce a type variable and generate possible constraints
  - Type T is a skeleton that may contain type variables
  - To determine whether e is well-typed, solve the constraints
  - Solving constraints generates a type substitution sigma
  - Consequently, we now have type expression *******

Inferring Types and Constraints by Example

Infer the type of : fun x -> x + 1

Infer the type of (x+1) under the assumption that x : ‘a

1. Infer the type of x under the assumption that x : ‘a ➔ **‘a (constraint: ‘a = int)**
   a. CHECK THE RESULT OF INFER
      i. Whatever I infer for x, it must be an int (+ requires two integers)
2. Infer the type of 1 under the assumption that x : ‘a ➔ **int**
   a. CHECK THE RESULT OF INFER
      i. Whatever I infer for 1, it must be an int (+ requires two integers)

We conclude the result of infer (x + 1) under the assumption x : ‘a is an int (plus of two ints returns an int)

Type: fun x → x + 1 ➔ ‘a -> int

Therefore, fun x → x+1 has type int → int

- How to infer the type of fn x → fn y → if x then y else 2 + 2
  - Assume x has type alpha and y has type beta
  - Infer the type of if x then y else 2 + 2
    - Infer the type of x as alpha and constraint alpha = bool
    - Infer the type of y as beta
    - Infer the type of 2 + 2 as int
    - Generate constraint beta = int
  - Fn x → fn y → if x then y else 2 + 2 has type alpha → beta → beta given the constraints alpha = bool and beta = int

Type Inference, More Formally

- Gamma t e → T/C: infer type T for expression e in the typing environment gamma modulo the set of constraints C
- Example with constraints (inputs = green, outputs = blue)

$$\frac{\Gamma \vdash e \Rightarrow T/\mathcal{C} \quad \Gamma \vdash e_1 \Rightarrow T_1/\mathcal{C}_1 \quad \Gamma \vdash e_2 \Rightarrow T_2/\mathcal{C}_2}{\Gamma \vdash \texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2 \Rightarrow T_1/\mathcal{C} \cup \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{T = \text{bool},\ T_1 = T_2\}}\ \text{T-IF}$$

Reading Statements

- Read T-IF rule from “Type Inference, More Formally”:
  - Given the typing context gamma and expression if e then e1 else e2, infer a type T1 and a set of constraints C U C1 U C2 U {T = bool, T1=T2} as follows

- Given the typing context gamma and expression e1, infer the type T1 together with the constraint C1
- Given the typing context gamma and expression e2, infer the type T2 together with the constraint C2

Inferring Types with Functions

- Examples
  - f x → x + 1 has type alpha → int/{alpha = int}
  - fn x → if x then x+1else 2 …/{alpha = bool, alpha = int} (type alpha → int, would not work)
  - fn f → fn x → f (f x)

How to solve constraints

- Can we solve the following constraints
  - {alpha = int, alpha → beta = int → bool}
    - Yes, alpha is an int, beta is a bool
  - {alpha1 → alpha2 = int → beta, beta = bool}
    - Alpha1 is an int, alpha2 is an int, beta is a bool (yes)
  - {alpha1 → alpha2 = int → beta, beta = alpha2 → alpha2}
- Constraint solving via unification
  - Two types, T1 and T2 are unifiable if there exists an instantiation sigma for the type variables in T1 and T2 such that [sigma]T1 = [sigma]T2, i.e. [sigma]T1 is syntactically equal to [sigma]T2

Unification via Rewriting Constraints

- Given a set of constraints C try to simplify the set until we derive the empty set
  - We write C for C1, …, Cn and assume constraints can be reordered
- Unification is a fundamental algorithm to determine whether two objects can be made syntactically equal

Let-Polymorphism

```
let
     double = fn f → fn x → f (f x)
in
     if double (fn x → x) false then
          double (fn x → x + 1) 3
     else 4
```

double is of type (alpha → alpha) → alpha → alpha
the code that follows typechecks
double is polymorphic (generic function, used at different types)

- Let expressions abstract over polymorphic variables → think of it as for all alpha

```
let test f x = if f x then (f 3) + 1 else (f 2) * 2
     (f  : a) → (x : b) → c
     a = b → bool
     a = int → int (does not work)
fun x ->
let f = fun x → x in
```

```
if f x then (f 3) + 1 else (f 2) * 2
```

- Type checks (identify function for all types)

Lecture 21 [11/23/2018]

- Last lecture on testable material for the final

Subtyping

- Every expression so far has had a unique type (even with modules) in OCaml
- In OCaml, every expression has a most general type
  - As such, this is a theoretical discussion
- Why do we want subtyping, why doesn't OCaml use it?

The Essence of Subtyping

- Basic subtyping principle: S is aa subtype of T if we can provide a value of type S whenever a value of type T is required
  - Example: integer is a subtype of number
  - In some languages, int is a subtype of float → int automatically converted to a float
- Subtyping goes a lot further than just thinking about integers and floats

How can we describe rigorously the subtyping relation?

- Relations for base types:
  - Int is a subtype of float, pos is a subtype of int, zero (singleton type – one element) is a subtype of int
- Types give us stronger runtime invariances
  - However, some things don't happen during runtime
  - Stronger type system → compiler doesn't need to catch certain errors
    - Implementation of evaluator → cases for expressions that were not well typed needed to be caught
      - Wouldn't arise if it only considered well-typed expressions
      - Doesn't need runtime check
      - Compiler produces better code
- Reflexive transitive closure: S-ref (T is a subtype of T) S-trans (S is a subtype of T if S and T are a subtype of T')

Example – Cross Products

- When is S1 x S2 a subtype of T1 x T2?
  - When S1 is a subtype of T1, S2 is a subtype of T2
- Cross products are co-variant
  - NOT commutative, order matters
  - Just looks at parts, compares S1 and T1, S2 and T2 only

Example – Records

- Records are n-ary labelled tuples
- Structural subtyping – looks at structure of types, rather than just name
- S1, S2, … Sn is a subtype of T1, T2, … Tn if S1 is a subtype of T1, S2 is a subtype of T2, … Sn is a subtype of Tn

$$\frac{S_1 \le T_1 \;\ldots\; S_n \le T_n}{\{X_1 : S_1, \ldots, X_n : S_n\} \le \{X_1 : T_1, \ldots, X_n : T\}}\ \text{S-RDepth}$$

- Object with fewer methods is higher in the hierarchy as it makes fewer commitments, has fewer requirements on what it needs
- N is a subtype of k if {x1:T1, …, xk:Tk is a subset of {x1:T1, …, xn:Tn}
- Sigma is a permutation if {x1:T1, …, xk:Tk} is a subtype of {x_sigma(1):T_sigma(1), …, x_sigma(n):T_sigma(n)}

$$\frac{n < k}{\{x_1 : T_1, \ldots, x_k : T_k\} \leq \{x_1 : T_1, \ldots, x_n : T_n\}}\ \text{S-Width}$$

$$\frac{\phi \text{ is a permutation}}{\{x_1 : T_1, \ldots, x_k : T_k\} \leq \{x_{\phi_1} : T_{\phi_1}, \ldots, x_{\phi_n} : T_{\phi_n}\}}\ \text{S-Perm}$$

Functions

- S1→ S2 is a subtype of T1 → T2 if we can provide a value of type S1 → S2 whenever a value of type T1 → T2 is required
- Can we provide a function float → int whenever a function float → float is required?

```
let areaSqr (r:float) = r *. r in
let fakeArea (r:float) = 3 in
```

  - Clearly, areaSqr 2.2 +. 4.2 has type float
  - areaSqr 2.2 has type float
  - areaSqr has type float → float
  - fakeArea has type float → int
- *Should the typechecker accept (not in OCaml specificallly):*
  - fakeArea 2.2 +. 4.2
    YES – fakeArea 2.2 produces an int and we can convert an int to a float
    NOTE – in OCaml, ints are not converted to float (for these examples we are assuming subtyping exists)
    - float → int is a subtype of float → float
- This seems to suggest that we can provide a function of type float → int whenever a function float → float is required

$$\frac{S_2 \leq T_2}{S \to S_2 \leq S \to T_2}\ \text{S-Fun-Try1}$$

```
let areaSq (r:int) = r*r in
let fakeArea (r:float) = (int_of_float r) + 3 in
```

  - Clearly, areaSqr 2 + 4 has type int
    areaSqr has type int → int
    fakeArea has type float → int
  - Should the typechecker accept the following: fakeArea 2 + 4
    YES – because 2 is an integer, it can be converted into a float, thus fakeArea accepts a valid input that returns an nit
- This seems to suggest we can provide a function of type float → int whenever a function int → int is required

$$\frac{T_1 \leq S_1}{S_1 \to S \leq T_1 \to S}\ \text{S-Fun-Try2}$$

  - Co-variant → T1 is a subtype of S1, but in the check, the function of input S1 is the subtype of the function of input T1
- S-Fun-Try1 → co-variant in the return type

- S-Fun-Try2 → contra-variant in the input type
- To accommodate both scenarios, we combine the two rules into one

$$\frac{T_1 \leq S_1 \quad S_2 \leq T_2}{S_1 \to S_2 \leq T_1 \to T_2}\ \text{S-Fun}$$

Revisiting References

Expression e ::= … | e1:=e2 | !e | ref e | ()
Types T : ::== … | T ref | unit

- Example:
  let x = ref (2 + 3) in
  x := !x * 2
- Types are a general organizing principle to

The Essence of Subtyping

- So far: base types int, float, etc.
  - Products
  - Functions
  - Co-variant, contra-variant input, co-variant output
- How can we combine subtyping and state?

State and Subtyping – Read Perspective

```
let x = ref 5.0 in
let y = ref (2 + 3) in
    !x *. 3.14
```

- Is it safe to write !y *. 3.14 instead of !x *. 3.14?
  - Can we provide a value of type int ref when a value of type float ref is required? YES
- This seems to suggest S ref is a subtype of T ref when S is a subtype of T

$$\frac{S \leq T}{S\text{ref} \leq T\text{ref}}\ \text{S-REF}$$

State and Subtyping – Write Perspective

```
let x = ref 5.0 in
let y = ref (2 + 3) in
    y:=4
```

- Is it safe to write x:= 4 instead of y:=4?
  - Can we provide a value of type float ref when a value of type int ref is required? NO
- This seems to suggest S ref is a subtype of T ref if S is a subtype of T

$$\frac{S \leq T}{S\text{ref} \leq T\text{ref}}\ \text{S-REF}$$

State and Subtyping

- A sound rule for subtyping in the presence of references cannot allow any subtyping to happen
- References are co-variant and contra-variant, in other words, they are invariant

$$\frac{T \leq S \quad S \leq T}{S\text{ref} \leq T\text{ref}}\ \text{S-Ref}$$

Java and Subtyping

- Subclassing is a form of subtyping
- Nominal subtyping → user declares what type (class) extends another type (class)

```
class mPoint {
     private int n;
     public mPoint(int n){this.n = n;}
     public int getval(){return n;}
}
class colorPoint extends mPoint{
     private int color;
     public coclorPoint(int n, int c){…}
}
public static void main(String[] args){
     mPoint p = new mPoint (5);
     colorPoint cp = new colorPoint(6,3);
     colorPoint [] A = new colorPoint[10];
     mPoint[]B = new mPoint[15];
     B[0]=p;
     A[0]=cp;
     //upcast since colorPoint < mPoint
     mPoint[] C = (mPoint []) A;
     C[1] = cp; //since colorPoint < mPoint
     C[2] = p;  //compiles, but runtime exception (adding element of
                different types)
}
```

Lecture 22 [11/28/2018]

- Still have office hours every Friday until exam

Final Exam Review

Main goals of the course

- Provide a thorough introduction to fundamental concepts in programming languages
  - Higher order functions, state-full vs. state-free computation, modelling objects and closures, exceptions and continuations to defer control, polymorphism, partial evaluation, lazy programming, modules, etc.
- Show different ways to reason about programs
  - Type checking, induction, operational semantics, type inference
    - Induction proof on final
- Introduce fundamental principles in programming language design
  - Grammar and parsing, operational semantics and interpreters, type checking, polymorphism, subtyping
- Expose students to a different way of thinking about problems
  - Trains you to think recursively

Functional vs. Object Oriented Programming

- Functional programming:
  - Data and functions are decoupled
  - Easy to add new functionality
- Object Oriented Programming
  - Data and functionality are coupled
  - Easy to add new data

Example from class: implementing an expression parser

```
type people = Student of string * int * float | Profs of string * int
     | Admin of string * int
let p2st p = match p with
     | Student (s, id, gpa) -> print_string s
     | Prof (s, id) -> print_string s
     | Admin (s, id) -> prints_string s
type exp = Int of int | Bool of bool | Plus of exp * exp
let rec exp2strng e = match e with
     | Int n -> string_of_int n
     | Bool true -> “true”
     | Bool false -> “false”
     | Plus (e1, e2) -> exp2string e1^ “+” ^ exp2string e2
```

- In an objeoriented language, you would have a class for each type, with methods specific to each rather than functions
  - Object is primary, functionality is secondary (scattered throughout, with each object)
  - Easier to add new functionality in functional programming

Effect of programming languages on software quality

- Summary of results
  - Better than procedural languages
  - Disallowing implicit type conversion is better
  - Static typing is better than dynamic
  - Managed memory usage is better than unmanaged
    - Garbage collection
  - The defect proneness of languages is not associated with software domains

Typing

1. Expression fun x y -> if x * (-1) < 0 then (y,x) else (x,y) has the most general type
   a. int * int
   b. int → int → int * int
      i. function, takes in two values
      ii. x is int because multiplied by int
      iii. (x,y) or (y,x) can be returned, thus a tuple of (int,int) must be returned and y must be the same type as int
   c. int * int → int * int
   d. int → ‘a → int * int
   e. int → ‘a → int * ‘a
   f. int * ‘a → int * int
   g. int * ‘a → int * ‘a
2. What is the most general type of the following expression?

```
let rec f b l = match l with
     | [] -> b
     | x::xs -> f x
```

   a. ‘a -> ‘a list -> ‘a
   b. (‘a -> ‘b) -> ‘a -> ‘a list -> ‘b
   c. Ill-typed
   d. int -> int list -> int
3. What is the most general type of the following expression

```
let rec f x = f (f x) in (f true, f 1)
```

   a. bool * int
   b. ill typed
   c. stack overflow, looping recursion
4. What is the most general type of the following expression

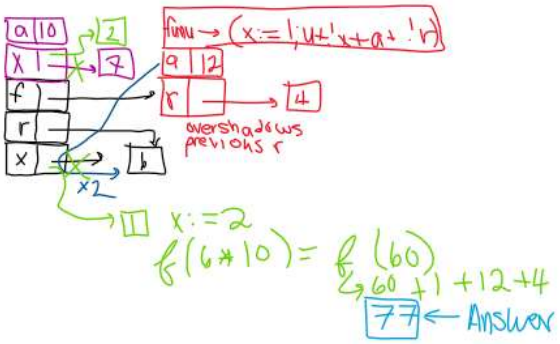
```
let combine f g = function x -> f x (g x)
```

Values

5. What value does this expression have?

```
let x = ref (3 * 2) in
let r = x in
let f = (let r = ref 4 in
         let a = !x * 2 in
         fun u -> (x := 1; u + !x + a + !r)) in
let x = ref 7 in
```

```
let a = 10 in
      x := 2; f (!r * a)
```



Answer: 77

Variables

6. What is the result of substituting y+1 for x in the expression

```
let y = x + 3 in fun y -> x + y
```

   a. let y = (y+1) + 3 in fun w -> (y + 1) + 2
   b. let y = y + 4 in fun w -> (y+1) + 2
   c. let v = (y+1) + 3 in fun w -> (y+1) + w
      i. change name of bound, not free variables
   d. let y = (y' + 1) in fun w -> (y' + 1) + w

Effects

7. When evaluating freshVar "y" what is the effect?

```
let genCounter =
let counter = ref 0 in
((fun x ->
      Let _ = counter := !counter + 1 in
      x ^ string_of_int (!counter)),
fun() ->
      counter :=0
let (freshVar, resetCtr) = genCounter
```

   a. Returns "y1"
      i. Not an effect, would be the value
      ii. Effect is always something in memory, exception, printing
   b. Updates counter to 1
   c. Nothing

- More practice on Friday