McGill University
Department of Computer Science

Hair Parra 2020 ©

COMP 302: Programming Languages and Paradigms

- Winter 2020 -

By Student Hair Albeiro Parra Barrera
Based on professor Prakash Panangaden's lecture notes
April 12, 2020

Contents

Hair Parra 2020 ©

Hair Parra 2020 ©

Hair Parra 2020 ©

Hair Parra 2020 ©

Hair Parra 2020 ©

1 Introduction

Programming language design issues and programming paradigms. Binding and scoping, parameter passing, lambda abstraction, data abstraction, type checking. Functional and logic programming.).

2 Lecture 1: 2020-01-07

2.1 Introduction

The following applies to this class:

1. **Course webpage:** `https://www.cs.mcgill.ca/~prakash/Courses/302/comp302.html`
2. **Piazza page:** `https://piazza.com/class/k4x9e0ra1344zj?cid=9`
3. Grading: See outline.

2.2 Something else?

Definition 1 (**Paradigms**)**.** A disctinct concept or **thought pattern**.

- **Functional programming**: higher-order, polymorphically typed (OCaml)
- **Imperative programming**: (OCaml)
- **Object-oriented programming**: inheritance and subtyping (Java)

2.3 Topics

1. **Recursion**
2. How to think about it, not how it is implemented with stacks!
3. **Inductively defined types and structures**
4. **Operational Semantics**
5. **Higher-order functions**

Hair Parra 2020 ©

6. **Updatable data: references**
7. **Environments and bindings**
8. **Closures and Objects**
9. Some other topics

2.4 Topics II

1. Types, typing rules
2. **Type inference**
3. **polymorphism**
4. Interpreters, parsers and compilers.
5. Object oriented paradigm.
6. Subtyping and inheritance (not the same thing!)
7. Stream programming, if time permits

Definition 2 (**Type Inference**)**.** The interpreter/system has to figure out what the type of a certain "object" is.

Definition 3 (**Polymorphism**)**.** From greek "many-shapes"; the same object may have (possibly) infinitely many types.

2.5 The role of language

- We all speak natural languages with varying degrees of precision and accuracy,
- Sloppy language causes confusion in mathematics.
- Computers are completely **unforgiving**
- "Every sentence has to be constructed with care".
- Ex. "can" vs. "cannot", "that" and "which".
- "I would like eggs and bacon or sausages" (**Ambiguity**)

Hair Parra 2020 ©

2.6 What is the study of language?

- **Linguistics**: syntax and semantics (in programming languages).
- **Syntax**: what is a correctly formed sentence/program?
- **Semantics**: what does a sentence mean? – what does a program do when you run it?
- Commonly written in manuals (but ambiguous)
- Will study a bit of ***formal semantics*** and ***typing rules***.

2.7 The role of abstraction

Definition 4 (**Abstraction**)**.** Conceptualization *without* reference to specific instances. Isolate the fundamental, essential issues without irrelevant details.

NOTE: Our only technique for handling **complexity**.

2.8 Pieces of a programming language

- Values (FP)
- Names (FP)
- Variables(Location)
- Expressions (Description of the state of the world, need name+context, **True/False** value) (FP)
- Commands (Just happen)

2.9 Higher-level pieces

- Combination mechanisms
  1. control-flow constructs
  2. combinators
- Parametrized expressions = functions (**immutable**)
- Parametrized commands = procedures (methods) (**might modify things**)
- Modules: independent compilation

Hair Parra 2020 ©

2.10 Types

1. Classify values and expressions, functions and procedures.
2. Purpose: *restrict* what can be expressed.
3. Give up expressive power for the guarantee of good behaviour.

SIDE NOTE

- We have functions that are **computable** and other that are not (Alonso Church). We would like to get some guarantee of whether functions are computable.

  Definition 5 (**Computable Function**)**.** A function $\mathbb{N}^k \to \mathbb{N}$ is computable if and only if there is an effective procedure that , given any k-tuple $\mathbf{x} \in \mathbb{N}$, will produce the value $f(\mathbf{x})$.

  Definition 6 (**Turing Completeness**)**.** A system of data-manipulation rules (such as a programming language) is said to be **Turing complete** if it can be used to simulate any Turing Machine

- In some situations, we may give up Turing Completeness, so that we can have something else; i.e. a **trade-off**.
  - Ex: **Lambda Calculus**: Not Turing complete, but guarantees that every program will terminate.

2.11 What to understand

1. Names: binding and scoping
2. Evaluation rules: expressions $\implies$ values
3. Typing rules (which may not be exclusive)

2.12 Where do we go from here?

1. Intimate connection between logic and computation: **the Curry-Howard isomorphism**
2. New directions in type theory: **guaranteeing security**
3. New logics and new programming paradigms: **linear logic**
4. Probabilistic programming languages designed for machine learning

Hair Parra 2020 ©

2.13 OCaml Overview

1. **Functional**: functions are the main entities
2. **Higher Order**: functions may take other functions as arguments, and may even return functions as results
3. **Typed**: every entity has a type
4. Types are described in their own little language; types are not just the basic types.
5. **Expressions** may have multiple types: **polymorphism**

2.14 Basic Components of any programming language

1. Basic values: `true`, `false`, 1,2,...,1.34,'a','b'
2. Compound values: data structures,
3. Expressions: an entity that triggers a computation resulting in a value, e.g. $1+2 \rightarrow 3$
4. Names: symbols that denote values
5. Bindings: correspondence between name and value established by a definition
6. Parametrized expressions: functions(procedures, methods)

2.15 Things we will do without for now

1. **Updatable storage**: abstraction of memory locations
2. **Control flow**: the only control flow will be a function applied to an argument
3. Will incorporate these later

2.16 Some concrete examples

Example 1. Binding using the keyword **let**:

```
# let x = 1 ;; (*Assigning a value to x*)
val x: int 1 (*feedback*)
# x ;;
-: int
```

Hair Parra 2020 ©

Example 2. An evaluation expression:

```
# let y = x + 1729 ;;
val y : int = 1730
```

Example 3. Function definition and application

```
# let inc = fun n -> n + 1;;
val inc: int -> int = <fun>
# let foo = inc 5;;
val foo: int 6
```

2.17 Recursion

Example 4. Classic recursion

```
# let rec fact n =
if n = 0 then 1
else
   n * fact(n-1);;
val fact : int -> int = <fun>
# fact 5;;
-: int = 120
```

2.18 Thinking recursively

Three things to keep in mind:

1. **exit condition**
2. recursive calls must make progress towards the exit condition
3. if the recursive calls are **assumed to work** then check that the body works correctly.

2.19 Tail recursion

Definition 7 (**Tail recursion**)**.** There should be only one recursive call and it should be **outermost**.

Example 5. Fast factorial

Hair Parra 2020 ©

```
let fastfact n =
   let rec helper(n,m) =
      if n =0 then m
      else helper(n-1, n*m)
   in
   helper(n,1);; (*Call to the function*)
val fastfact : int -> int <fun>
```

Notice how this function works, let's say, call fast fact with 3 (not actual OCaml):

```
fast fact 3
-> helper(3,1)
-> helper(2,1*3) = helper(2,3)
-> helper(1,2*3) = helper(1,6)
-> helper(0,1*6) = helper(0,6)
-> return m = 6
fast fact returns 6
```

3 Lecture 2: 2020-01-09

3.1 Basic Recursive Functions

Note 1. Note that

1. We'll be using OCaml for the course and assignments.
2. Go to the following link: `http://winter2020-comp302.cs.mcgill.ca/`
3. Example code inside the link.

Definition 8 (**Function**)**.** A mapping for which, for a given input, the output is unique.

Definition 9 (**Function composition**)**.** Informally, it means putting a function inside another function. See fig 1

Remark. Functional programming guarantees that the mathematical definition of function composition is well represented.

Definition 10 (**Binding**)**.** A correspondent assignment of a value to a certain symbol(name)

Definition 11 (**Environment**)**.** Consists of layers of such bindings.

Hair Parra 2020 ©

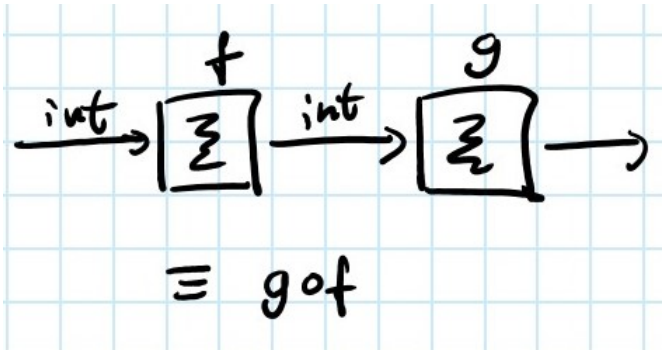

Figure 1: Composition of two functions f and g, mapping from int $->$ int

Example 6 (**Regular factorial**)**.**

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

Problem!; the function above has too many useless calls. Instead, use **Tail recursion**.

Example 7. Fast factorial

```
let fastfact n =
   let rec helper(n,m) =
      if n =0 then m
      else helper(n-1, n*m)
   in
   helper(n,1);; (*Call to the function*)
val fastfact : int -> int <fun>
```

Example 8. Fibonacci

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

Problem! Runtime $O(2^n)$. We can make it $O(n)$.

Hair Parra 2020 ©

Example 9. Tail recursive fibonacci

```
let rec tailfib n =
   let sec helper(n,a,b) = (*Subfunction with extra parameters a and b*)
      if n=0 then a (*Base case 1*)
      else if n=1 then b (*Base case 2 *)
      else
         (*Tail recursive call: decrease n , cumulate result in third argument*)
         helper(n-1, b, a+b)
   in
   helper(n,0,1) (*Repeat procedure n times *)
```

Example 10. Russian peasant exponentiation

```
let even n = (n mod 2) = 0;;
let odd n = (n mod 2) = 1;;

let rec rpe base power =
   if base = 0 then 0
   else
      if power = 0 then 1
      else
         if (odd power) then
            base * (rpe base (power - 1))
         else
            let tmp = (rpe base (power/2)) in
            tmp * tmp;;
```

Example 11. Tail recursive peasant exponentiation

```
let even n = (n mod 2) = 0 ;;

let fast_exp (base, power) =
   let rec fast_exp_aux (base, power, acc) =
      if base = 0 then 0
      else
         if power = 0 then acc
         else
            if  (odd power) then
               fast_exp_aux (base*base,((power-1)/2),(acc * base))
            else
               fast_exp_aux ((base*base), (power/2), acc)
   in
   fast_exp_aux(base,power,1)
;;
```

Hair Parra 2020 ©

Example 12. A simpler tail-recursive Russian peasant exponentiation

```
let even n = (n mod 2) = 0 ;;

let fast_exp (base, power) =
   let rec fast_exp_aux (base, power, acc) =
      if base = 0 then 0
      else
         if power = 0 then acc (*return accumulator parameter*)
         else
            if (odd power) then
               fast_exp_aux (base, power-1, acc * base) (* x*x^{n-1} *)
            else
               fast_exp_aux (base, power/2, acc*acc)
   in
   fast_exp_aux(base,power,1)
```

Remark. Key to Tail recursion:

1. ONLY 1 recursive call
2. Recursion must be outermost

Example 13. The following functions sums up the numbers from `lo` to `hi`, cumulatively.

```
(*Version 1*)
let sec sumnums lo hi =
   if lo > hi then 0
   else lo + sumnums (lo+1) h (*Not tail recursive!*)

(*Version 2*)
let tailsum lo hi =
   let  helper lo hi tally = (*tally carries the cumulative sum*)
      if lo > hi then tally (*nosense, stop recursion*)
      else
         (*increase count (lo) and cumulate result to tally *)
         helper (lo+1) hi (tally+lo)
   in
   helper lo hi 0
```

This is an example of a **multi-argument function**.

3.1.1 Multi-argument functions

- Note that

Hair Parra 2020 ©

```
let myadd(n,m) = n + m ;;
!=
let myadd2 n m = n + m ;;
```

- `( , )`: package two values in a single entity called **pair**
- type of `myadd`: `int * int -> int`
- (,) allows even two values of different type. Ex.: `(7,true)`: `int * bool`.
- NOTE: `myadd 5 ;; -> type error`
- NOTE: `int -> (int -> int)` # Input an integer, return a function that takes and integer and returns another integer, i.e.

```
let foo = myadd 2 5 -> a function
foo: int -> int
foo 3
8 : int
myadd2 5 3 -> 8
```

- This process is called **currying**.
- NOTE: this is allowed: `((3,true),"foo")` (but it's quite barbaric, don't do this).
- In general `(x1,x2,...,xn)` is called a **tuple**, and can include any type of values inside, even other tuples.

3.2 Pattern Matching

- **Parameter**: name that you use when you define the function
- **Argument:** the actual value employed
- **Binding** between the parameter and the argument.

Definition 12 (**Pattern Maching**)**.** A parameter can be a "structured" name or a pattern, and OCaml provides a way to allow for multiple possible input patterns.

Example 14. The following function will consider different patterns as arguments

```
let myadd(n,m) =
  match (n,m) with
    |(0,_) -> m  (* '_' means anything *)
    |(_,0) -> n
    |(0,x) -> x
    | _ -> n + m
```

Hair Parra 2020 ©

Example 15. Silly string

```
let silly str =
   if str = "Prakash" then str ^ " is awesome."
   else if str = "Prof." then str ^ " Prakash, is incorrect."
   else "Who cares";;
```

Example 16. Less silly string

```
let less silly str =
   match str with
   | "Prakash" -> str ^ "is awesome"
   | "prof." -> str ^ " Prakash, is incorrect."
   | _ -> "Who cares?";;
```

Example 17. Consider the following two functions: one without pattern matching, and the other one with.

```
let add_pair pair = fst pair + snd pair ;; (*input is a pair ( , )*)

let add_pair pair =
   match pair with
   | (1,1) -> -1
   | (x,y) -> x + y (*figure out and match*)

# add_pair(1,1) ;;
- : int = -1
```

NOTE! Cannot match on something of a different type.

```
(*another way to write it*)
let add pair (x,y) = x + y ;;
let add pair' x y = x + y ;;

val add_pair: int*int -> int = <fun>
val add_pair': int -> int -> int = <fun>
```

Hair Parra 2020 ©

3.3 Lists: Built-in structure

3.3.1 How to build a list

- Construction:

```
:: 'a * 'a list -> 'a list
hd: 'a list -> 'a
tl: 'a list -> 'a list
```

- Construction & destruction of lists is based on pointer manipulations. It does not matter what is stored in the cells. These act like *linked lists*, but **all the elements have to be the same time.**
- **'a list** is a **polymorphic type**.
- " 'a " is called a **type parameter** and/or **type variable**. (i.e., can be instantiated with types).

Example 18.

```
[] ;; (*an empty list*)
[17;29; 41;5] int list (*example list instantiation/assignment*)
3 :: [1;5;7] ~-> [3; 1; 5; 7] (*inserting an element at head*)
2 :: (1 :: []) ;;
- : int list = [2,1]
```

Example 19. Another example

```
let sec sumlist l =
   match l with
   | [] -> 0 (*This is an empty list*)
   | x:x -> x + (sumlist xs)
```

Example 20 (Concatenating two lists)**.** Use the @ operator to concatenate two lists

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

Note 2. Note that inserting at the beginning is $O(1)$, while appending at the end is $O(n)$!

```
let vowels = ['a';'e';'i';'o';'u'] (*create list*)
val vowels = char list = ['a';'e';'i';'o';'u']

'y' :: vowels (*append at the eginning*)
-: char list = ['y';'a';'e';'i';'o';'u']
```

Hair Parra 2020 ©

```
vowels :: 'y' ;;
This expression has type char but an expression was expected of type char list list

vowels @ ['y'] ;;
-: char list = ['a';'e';'i';'o';'u';'y'] (*legal, but not very efficient...*)
```

Note that lists are immutable!

4 Lecture 3: 2020-01-14

4.1 From last week

Example 21 (Russian Peasant Exponentiation Algorithm)**.**

```
let rec rpe b e = (*b=base,e=exponent*)
   if b=0 then 0
   else if e=0 then 1
   else if (odd e) then
      b * rpe(b,e-1)
   else
      ///// rpe(b, e/2)* rpe(b, e/2)//// (*bad! two recursive calls*)
      let v = rpe(b,e/2)
      in v * v
```

Note that this is not tail recursive!! The significance of tail recursion is reducing the number of stack calls.

4.2 The "let" keyword

- `Let` creates a binding.
- **Bindings** come in layers.
- We never bind a name to an unevaluated expression :

```
y = x + 1 (*BAD!!*)

let x = 1729 in
   let y = x+1 in
      x + y
```

Hair Parra 2020 ©

- A new binding always hides the old one.

```
let x = 1 in
   let y = x + 1 in
      let x = 2 in
         y
(*returns 2, NOT 3*)
```

- The meaning of a binding does not change: STATIC BINDING, (do not employ "dynamic binding"=variables that change around).
- In the example above, there is a masking of binding.

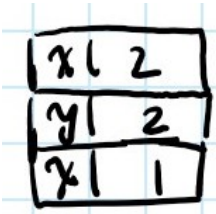

Figure 2: Stacking bindings

Example 22 (Binding process)**.**

What happens in the following?

```
inc 3 ;;
inc 5 ;;
inc (inc (inc 3)) ;;
```

Above, in the third line, the inner evaluation will be performed first and killed, and then fed to the next one and killed, then once again fed to the outer one and killed.

4.3 Lists

- We have the following pieces:

```
::    constructor
hd    (head) extractor / destructor
tl      (head) extractor / destructor


let liszt = [1; 2; 3; 4; 5] ;;
val liszt : int list = [1; 2; 3; 4; 5]
```

Hair Parra 2020 ©

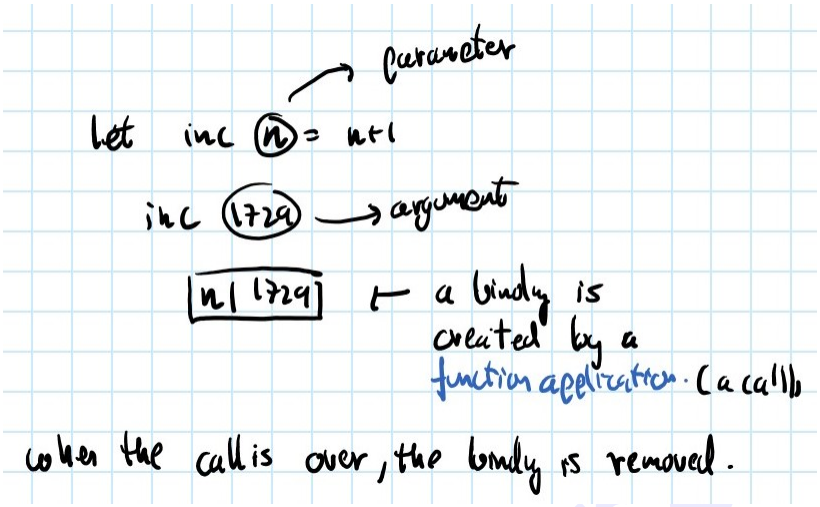

Figure 3: Binding process

```
let v = List.hd liszt ;;
val v : int = 1

let t = List.tl liszt ;;
val t : int list = [2; 3; 4; 5]
```

- 'hd' stands for head (returns the head element) and 'tl' for tail (returns list without head element). We can use these to take a list a apart. However this is heavily discouraged. Instead of destructors, we will use pattern-matching.

Example 23 (Inserting an element into a list).

```
0 :: [1;2;3] ~> [0;1;2;3] (*notice the :: operator*)
```

What if we want to to append 0 to `[1;2;3]` to obtain `[1;2;3;0]` ?

```
0 :: [1;2;3] ~> [0;1;2;3] (*notice the :: operator*)

[1;2;3] :: 0 (*Gives a type error!!*)
```

Note the type!
α `*` α`-list -> `α`-list`
Instead, make 0 into a list and concatenate! In OCaml, we can use the `@` operator to concatenate two lists:

```
let mylist = [1;2;3]
let mylist2 = mylist @ ['0']
mylist -> [1;2;3;0]
```

Hair Parra 2020©

```
let mylist3 = [4;5;6]
let catted = mylist @ mylist2
mylist2 -> [1;2;3;4;5;6]
```

Example 24 (**Append / Concat function**)**.** Suppose we want to append an item (or a list) at the end of a list.

```
let rec append(l1,l2) ~> l1 l2

Goal:
append([1;2;3], [4;5;6] ~> [1;2;3;4;5;6])
```

This is a built-in function in OCaml, with the operator `@`, but we will write it from scratch anyway:

```
let rec append (l1,l2) =
   match l1 with
   | [] -> l2  (*nothing to return*)
   | x :: xs -> x :: (append(xs, l2)) (*Tail recursive call to the sublist!*)
```

This runs in the order $O(n)$, where n is the length of l1. (Note that `::` is the insert at head operator)

Example 25 (**Reverse a list**)**.** The following function works but is $O(n^2)$, as we have linearly many linear time calls. The basic idea is that we append the first element of the list at the end at each iteration.

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

We will do it again with Tail recursion, using an accumulator parameter; the idea is that we cumulatively append the result in acc as we go when reversing the list, passing again the inserted element of acc as itw own parameter. When the list is full, acc (the reversed list) is returned. This works in $O(n)$ time.

```
let rev l =
   let rec helper(l, acc) =
      match l with
      | [] -> acc
      | x::xs -> helper(xs, x::acc)
   in
helper(l,[]) (*Call with list and empty list acc*)
```

Hair Parra 2020 ©

Example 26 (**Zipper**)**.** The length of the lists are unspecified. The zipper function cross-combines the elements of two lists into a new one.

```
GOAL: zip([1;3;5],[2;4;6]) ~> [1;2;3;4;5;6]

let rec zip (l1,l2) =
   match l1 with
   | [] -> l2
   | x::xs -> x::zip(l2,xs)) (*Note we flipped the arguments! *)
```

Above, we flip the arguments so that the argument is taking from one list and the other.

Note 3 (Bad Zipper)**.** The zipper function could be implemented using the methods `hd` and `tl` as follows:

```
let rec badzip (l1,l2) =
   if l1 = [] then l2
   else
      (List.hd(l1))::(badzip (l2,List.tl(l1)))

let foo = badzip(liszt,liszt2);;
```

Don't do this.

Example 27 (**Insertion and Sorting**)**.** We want to take the first item of a list, and put it in the right place.

- `insert: int * int list -> int list`
- Expects a sorted input & produces a sorted list

```
let rec insert n l = (*l is already sorted*)
   match l with
   | [] -> [n]   (*empty list, just place the element*)
   | x::xs ->    (*match with first element and rest of list*)
      if (n <= x)   (*if n is less than first element*)
         then n::l  (*put in front of list*)
      else
         x::(insert n xs) (*concat first elem. to insert n in sublist xs*)
```

We can now use this insertion function to create **insertion sort**.

Hair Parra 2020 ©

```
(*Insertion Sort*)
let rec isort l =
  match l with
  | [] -> []
  | x::xs -> insert(x, isort(xs));; (*Insert element into sorted sublist*)
```

5 Lecture 4: 2020-01-16

5.1 Function evaluation

- How to build complex programs? **function composition** .
- Recall the figure 4

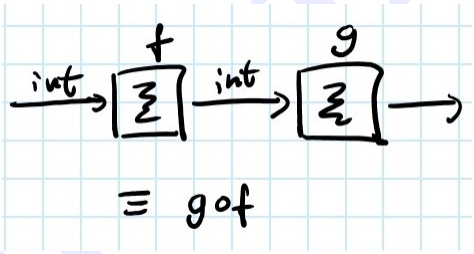

Figure 4: Function composition

- i.e., we can use functions as arguments for other functions.

Example 28. Consider the following examples:

```
let inc n = n+1 ;;
   OR
let inc = fun n -> n+ 1

let id fun x -> x (*IDENTITY*)

inc 7 (*NOT A FUNCTION!!!*)
```

Note that `inc 7` is not a function, it is an **evaluation expression**.
We have the following:

```
Parameter   n
Body        n+1
```

So what is actually happening? Answer:

Hair Parra 2020 ©

1. **Substitution Model**
2. Observer the parameter
3. SUBSTITUION: replace parameter by evaluated argument in the body
4. `n+1 > 7 + 1 > 8`

5.2 Environment Model of Functional Expression Evaluation

Example 29.

```
let inc = fun n -> n+1
```

- Function application creates a binding. (Fig 5)

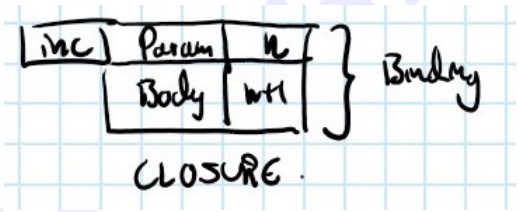

Figure 5: Function binding: the picture indicates the different parts in function binding.

Example 30. Consider the following function; how does it get evaluated? This is depicted in figure 33.

```
let x = 7 in
   let inc = fun n -> n+1 in
      inc x ;;
```

1. Fist, a binding for `[x|7]` is created.
2. A functional binding for `inc` is created, along with the parameter `n` and the expression `n+1`
3. The function `inc` is called (third line).
4. As x was called in inc, but doesn't exist in it , we go to the lower level of calling stack, to and retrieve `x=7`.
5. A binding `n=7` is produced as argument to `inc`
6. `inc` is executed , the expression `n+1` is evaluated to 8 and returned.
7. The binding `n=7` is destroyed, then `inc`, then `x=7` as well.

Hair Parra 2020 ©

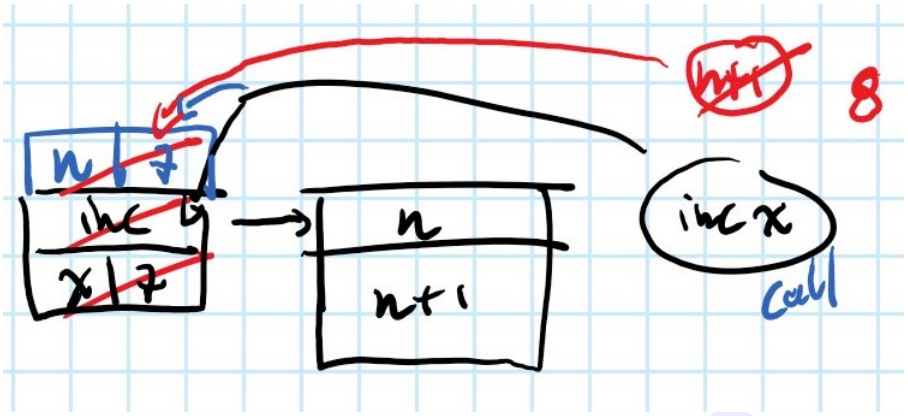

Figure 6: Structure of function application and binding

5.3 Higher order functions

Definition 13 (**Higher-order functions**)**.** Any amount of nested composition of functions that takes another function as an input, etc.

5.3.1 Higher order functions specific to lists

- We want to apply a given function to every element of a list.
- This is called a **map**.

Definition 14 (**Map**)**.**
Applies a function to every element in a list.
Type: `map:` $(\alpha \to \beta) \to \alpha - list \to \beta - list$.
For `f:` $\alpha \to \beta$ and `l:` $\alpha - list$:

```
let sec mymap f l =
   match l with
   |[] -> []
   |x::xs -> (f x)::(mymap f xs) (*use recursion to reapply to each elements*)

mymap (fun n -> n + 1) [1;2;3];;
value -> [2;3;4] ;;
```

This function is built in , located in the `module List`. (Command: `List.map`).

```
open List ;;
map inc [1;2;3] ~> [2;3;4]
```

Hair Parra 2020 ©

Definition 15 (**Filter**)**.** Applies a function to all elements that passes a certain test (another function $\alpha \to$ `bool`).

```
let rec myfilter test l =
  match l with
  | [] -> []
  | x :: xs -> if (test x) then x :: (myfilter test xs)
               else (myfilter test xs);;

let l2 = myfilter odd l1;;

let odd n = (n mod 2) = 1;;
let l1 = List.init 9 (fun n -> n * n);;
```

Example 31 (Using Init)**.**
Function implemented in OCaml to initialize a list.

```
open List;;

filter odd l1;;
map (fun n -> n * n * n) (init 10 (fun n -> n));;
```

The following is the most powerful list processing function in the universe:

Definition 16 (**fold-left**)**.**
This function is a combiner.
Type: $(\alpha \to \beta \to \alpha) \to \alpha \to \beta\text{-}list \to \alpha$.
`fold-left` f a $[x_1, x_2, \ldots, x_n] = (f \ldots (f(f\ a\ x_1)\ x_2) \ldots x_n)$
Takes a function, an accumulator and a list.

```
let sec fold_left f acc l =
   match l with
   |[] -> acc                          (*done, ret accumulator*)
   |x::xs -> fold_left f (f acc x) xs  (*process along the list*)
```

Note that this is **tail recursive**.

Example 32 (sum_list)**.**

```
let sum lst =
   fold_left (+) 0 lst;;
```

Hair Parra 2020 ©

Example 33 (**Concat_lists**).
Receives a list of lists and combines them into a single one.

```
let myconcat = fold_left (@) [] l
val myconcat : 'a list list -> 'a list = <fun>
```

Example 34 (string_mash).

```
let stringmash strlst =
   fold_left (fun s1 -> fun s2 -> s1^s2) "" strlst (* ^ is the built-in string append*)
```

Example 35 (**Reverse a list**).

```
let rev l = fold_left (fun a x -> x::a) []
```

Definition 17 (**fold-right**).
`fold_right`: $(\alpha \to \beta \to \beta) \to \alpha - list \to \beta \to \beta$. This function is also a combiner.
`fold_right` f $[x_1, x_2, \ldots, x_n]$ b $= (f \ldots (f(f\ x_n\ b)\ x_{n-1}) \ldots x_1)$

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

Note that this is **not tail recursive** (as the program has to wait for fold_right when inside the accumulator).

What can we do with `fold_right`?

Example 36 (**Map**).

```
fold_right (fun x a -> (f x)::a) l []
```

Example 37 (**Newstringmash**).

```
let newstringmash strlst =
   fold_right (fun s1 -> fun s2 -> s1 ^ s2) strlst "";;
```

Example 38 (**Several functions**).

```
(* We can write many things in terms of folds. *)
let len l = fold_left (fun a _ -> a + 1) 0 l;;
let rev l = fold_left (fun a x -> x :: a) [] l;;
let newmap f l = fold_right (fun x a -> (f x) :: a) l [];;
let newfilter f l = fold_right (fun x a -> if f x then x :: a else a) l [];;
```

Hair Parra 2020 ©

5.4 Why pure functional programming? (propaganda time)

- No side-effect, e.g. update
- No control flow (**if-then-else** is a is a conditional expression)
- Limited sense of computational precedence (time)
- We can execute computations in parallel.
- Data-flow architecture: computational triggered by arrival of data.

6 Lecture 5: 2020-01-21

6.1 Environment diagrams

Definition 18 (**Terminology**)**.**

- Terminology: Names, values, expressions(include names and values).
- **Names** and **values** are special cases of an **expression**.
- **Binding**: Association between a **name** and a **value** (never a general expression!).
- **Static binding/lexical scoping**: The meaning of a construct should not change; you should be able to look at a program and figure out what it means. *Think is structurally and not dynamically.*
- **Environment**: A structure containing bindings.

Example 39 (???)**.**

```
let x = 1 ;;
let x = 2 ;;
```

What's going on here? We are creating two separate bindings. In this case, the first binding is **masked** by the first one.

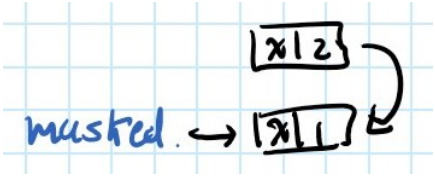

Figure 7: Masking

Hair Parra 2020 ©

Definition 19 (**Local bindings**).

Consider the statement `let name exp1 in exp2`:

1. Evaluate `exp1` to get `v1`
2. Bind name to `v1` and put it on top of the environment.
3. Use this binding to evaluate `exp2`
4. When this evaluation is complete, remove binding from 2.

For more detailed notes on environments and binding, see course website

Example 40 (**Bindings, Closure and function evaluation**).

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

What is happening here? The process is depicted in figure 8.

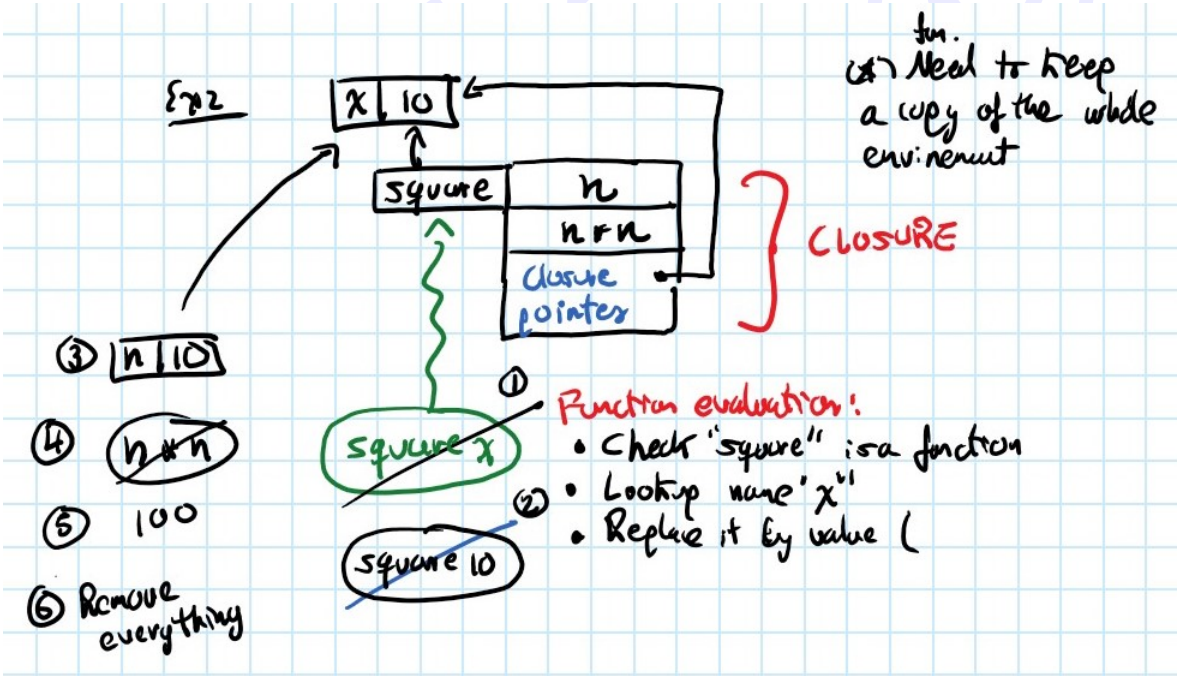

Figure 8: Binding, closure and evaluation 1

Note 4. Recursive functions are special: if you use `let rec` the closure pointer will include the frame that is being set up.

Example 41 (**Recursive Factorial Closure**).

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

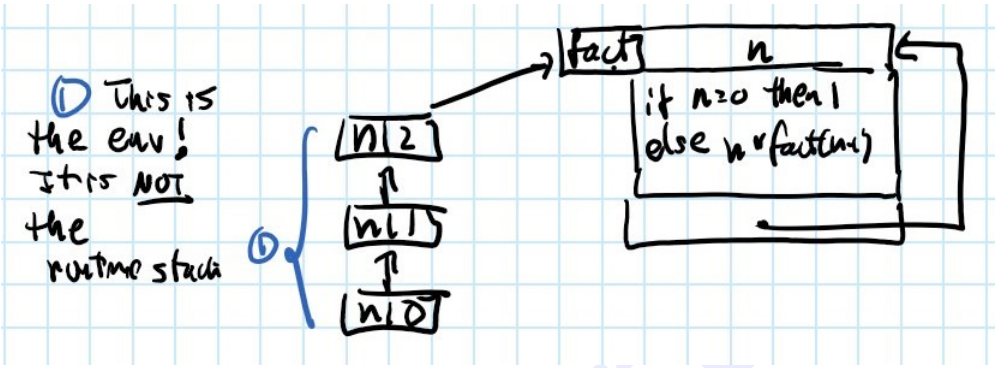

Figure 9: Recursive factorial closure: The closure pointer points to the definition environment, since this function is recursive, then it points to the function itself. The environment calls then get stacked one on top of the other, while updating the values correspondingly.

Example 42 (Nested let expressions)**.**

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

This is called **nested let expressions**.

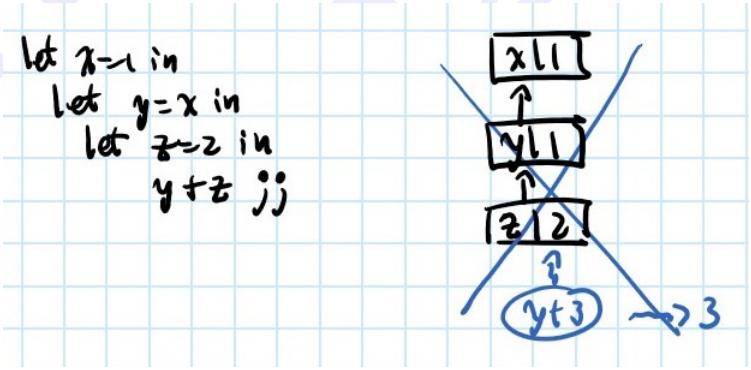

Figure 10: First `x` is bound to 1, then `y` is bound to `x`, which has value 1, so `y` gets bound to the value 1. Then `z` gets bound to the value 2, and the expression `y+z` is called. Since `y` has value 1, the expression is evaluated to 3, and the whole environment is deleted.

Hair Parra © 2020

Example 43 (Nested function application)**.**

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

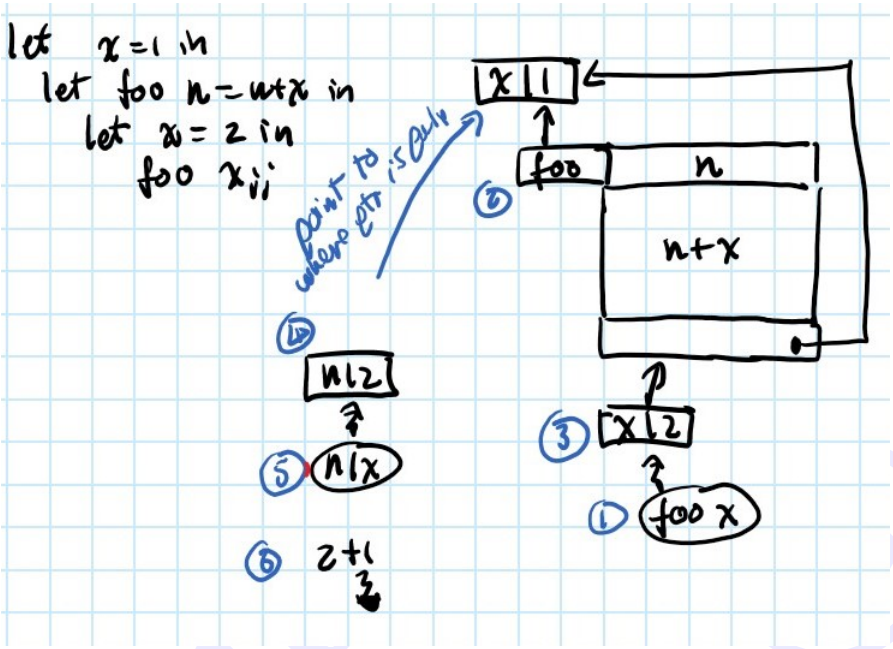

Figure 11: (After the environment is set) (1) `foo x` is called; this looks like a function with an argument. (2) compiler checks that foo is indeed a function. (3) compiler looks for a name `x`, which is bound to the value 2. It uses that value. (4) The value 2 is bound to the parameter `n`. (5) the expression `n + x` is called, this time, `x` is looked up in the preceding environment, having a value of 1. (6) The expression is evaluated as `2+1`, and 3 is returned.

Example 44 (Midterm/Final level example)**.**

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

For this example, please refer to figure 12 .

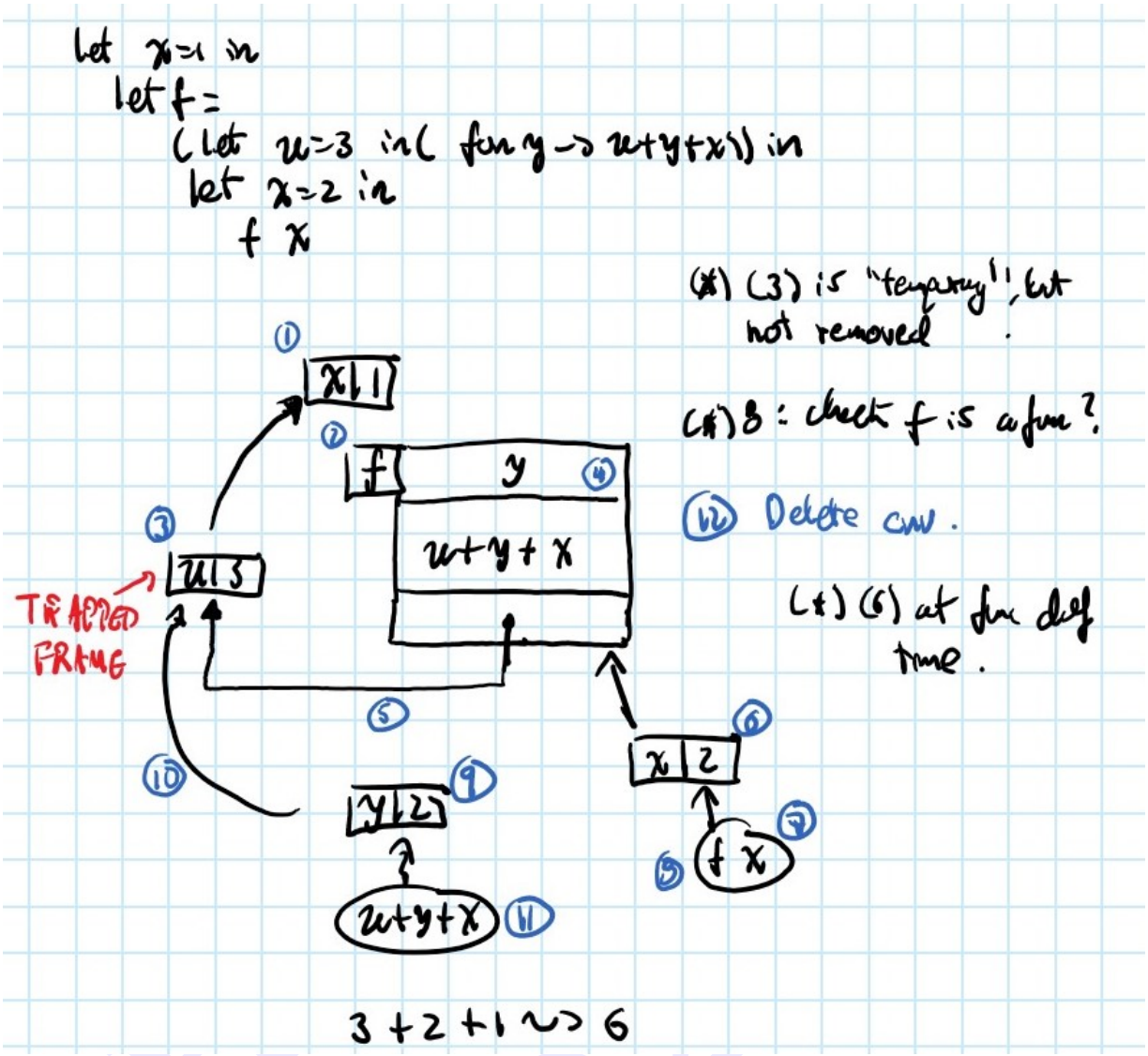

Figure 12: (1) First, $x = 1$ defines a binding. (2) The function definition is about to happen (although not for sure yet). (3) We define the "external" binding `u=3` (4) The previous binding exist in the context of the function that is now being defined: a parameter `y` is defined , and the body is also defined. (5) The closure pointer remembers the calling environment; so it points to the binding `u=3`. (6) The binding `x=2` is created, and points to the previous function. (7) The expression `f x` looks like a function, it goes to the body; note that `x=2` in the env. (8) (9) As `x=2`, and the call is `f x`, then in the function body, the parameter `y` is bound to the value 2. (10) `u` appears in the expression, but `u=3` from the environment. (11) name `x` appears in the function body, so the compiler looks it up in its environment. As it appears in the upper environment, this `x` is bounded to value 1. The expression `u+y+x` then evaluates as `3 + 2 + 1 -> 6`. (12) The whole environment is deleted. NOTE: in step (3), although `u` is "tempoarary", it is not removed as it interacts with upper bindings as well.

Example 45 (Correction and bindings)**.**
Consider the following situation:

```
let foo = ----
let bar = ---- foo

(*foo has an erro,it needs to be corrected*)

let foo = --- <CORRECTION> --

(*Error still happens! Foo remembers old definition. *)
```

Note: Error still happens because bar still remembers `foo` at the old definition.
You need to re-evaluate `bar` as well.

Example 46 (Equivalence of callings)**.**
The following two functions are equivalent:

```
let myadd a b = a + b
==
let myadd = fun a -> (fun b -> a+b)
```

6.2 Fixed point induction and inductive proofs

Example 47 (List insertion)**.**

```
let rec insert (n,l) =
   match l with
   |[] -> [n]
   | x::xs -> if n <= x then n::l
           else x::(insert n xs)
```

- ASSUMPTION: `l` is already sorted.
- WE WANT: output is also sorted

Proof.
We prove the claim by **induction** on the length of `l`.

Base case: `l=[]`

Hair Parra 2020 ©

- `output = [n]` is sorted.

Inductive step

Assume `insert` works correctly when $|l| < k$ for some $k \geq 0$.
Now we consider an l of length $k + 1$.
Let x be a number, $|xs| = k$.

The output is:

case 1 n ≤ x:
$\implies$ `n::(x::xs)` is clearly sorted.

case 2 `n > x`:
$\implies$ `x::(insert n xs)`
$\implies$ `x` ≤ anything in `xs`
$\implies$ `x` ≤ `n`
$\therefore$ (insert n in xs) is sorted by inductive assumption.

■

7 Lecture 6: 2020-01-23

7.1 Inductive definitions

Definition 20 (**Inductive definition**). 1. Base cases

2. Rules for adding new elements

Example 48 (Natural Numbers)**.**

- $0 \in \mathbb{N}$
- if $n \in \mathbb{N} \implies succ(n) \in \mathbb{N}$
- $Nat_0 = \varnothing$
- $Nat_1 = \{0\}$
- $Nat_2 = \{0, succ(0)\}$
- $\vdots$
- $Nat_k = \cup_{k \geq Nat_k}$

Hair Parra 2020 ©

Example 49 (List)**.**
`LIST = {[]} ∪ Nat*LIST`

- $LIST_0 = \{\}$
- $LIST_1 = \{[]\}$
- $LIST_2 = \{[0], [1], [2]\}$
- ⋮
- $LIST_{k+1} = LIST_k \cup Nat * LIST_k$
- $\therefore$
- $LIST = \cup_{k \geq 0} LIST_k$

7.2 Trees

Definition 21 (Trees)**.**

```
(* Lists *)
type 'a tree = Empty | Node of 'a tree * 'a * 'a tree ;;
```

- `Empty` is an `a' tree`
- Node is a function or constructor
- It has the same status as `::`
- `Node(Empty, 1, Empty)` : int tree
- `Node, 2, (Empty, Node(Empty, 1, Empty))`

Note that you can use pattern-matching with three constructors just as with lists!

Example 50 (**Height of a tree**)**.**

```
let max(n,m) = if n < m then m else n ;;

(*Def: Empty tree has height 0*)

let rec height (t:'a tree) =                    '
```

Hair Parra 2020 ©

```
    (*can pattern match on newly introduced types*)
    match t with
    |Empty -> 0
    |Node (l,_,r) -> 1+max(height l, height r) (*l:left subtree, r:right subtree*)
```

Example 51 (**Inorder traversal**).

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

- ; → sequencing
- Type: `int tree -> unit`
- `unit` is a special type containing only one element `()`
- NOTE: the ; is an example of `imperative programming`.

Example 52 (**Flatten**).

```
let rec flatten (t: 'a tree) =                    '
  match t with
  |Empty -> []
  |Node(l,v,r) -> v::((flatten l)@(flatten r))
```

Hair Parra 2020 ©

7.3 Expression Trees

Example 53 (Order of operations)**.**
$3 + 4 * 5 \implies 23$ Note the number binding , "order of operations".

Example 54 (**Expression Trees**)**.**
We will create our own toy example of performing simple algebraic operations using trees.

```
(*Implement our own type*)
type exptree =  Const of int
           | Var of char
           | Plus of exptree * exptree
           | Times of exptree * exptree

type binding = char * int
type env = binding list

(*Declare an exception*)
exception NotFound ;;

let rec lookup name (rho:env) =
   match rho with
   |[] -> None
   |(n,v)::e -> if name=n then (Some v) (*Pattern: pair and rest of the list*)
            else lookup name e

let rec eval (e:exptree) (rho:env) =
   match e with
   |Const n -> n
   |Var v -> match (lookup v rho) with
         |None -> raise NotFound
         |Some r -> r

   |Plus (e1,e2) ->
      let v1 = eval e1 rho in
      let v2 = eval e2 rho in
      v1 + v2
   |Times (e1,e2) ->
      let v1 = eval e1 rho in
      let v2 = eval e2 rho in
      v1 * v2
```

Hair Parra 2020 ©

7.4 How to handle situations where we fail to find something we are looking for?

Definition 22 (**Option Type**).
If `'a` is any type and `'a option` is a new type containing all the values of `'a` together with a special value to indcicate that you did not find what you are looking for, then

```
None
Some 17 -> special tag Some
```

You can pattern match on `None` and `Some`

7.5 How to abort a computation?

Definition 23 (**Exception**).

```
exception NotFound : declaring an exception
raise <exception name>
```

You can pattern match on `None` and `Some`

7.6 Aside: Why does inductive definition & Recursive programming go together

Naturally.

8 Lecture 7: 2020-01-28

1. **NOTES**
2. Midterm on 18 February 7-8pm, distributed along 3 rooms.
3. Question 1: Recursive function on lists
4. Question 2: Higher-order programming
5. Question 3: Environment model
6. 60 min.

Hair Parra 2020 ©

8.1 Higher-Order functions

Definition 24 (**Higher-Order Function**).

A function can be input to another function.

Example 55 (Map, Filter).
The following are examples of higher-order functions:

- `map: (a'->b') -> 'a list -> 'b list`
- `filer: ('a -> bool) -> 'a list -> 'a list`

Example 56 (Sumlist).
Let's construct a function that sums all ints from a to b:

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

Example 57 (SumSquares).
What if we want to sum the squares instead?

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

Example 58 (SumCubes).
Now we would like to have cubes!

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

The previous code is repetitive, we want to **abstract** this code!

Example 59 (Sum).
In order to **abstract** the code, instead of creating very specific functions for squares or cubes, we can pass the desired function as a parameter. Denote `lo` and `hi` for the low and high parameters.

```
let rec sum(f,lo,hi) =
   if lo > hi then 0
   else (f lo) + sum(f,lo+1,hi) (*function can be whatever*)
```

Hair Parra 2020 ©

Why should we increment just by 1? Can we generalize it even more?

Example 60 (SumInc)**.**
Here, we want to specify the incrementer `inc` as another functional parameter instead.

```
(*instead of +1, inc is an incrementer function*)
let rec sumInc(f,lo,hi, inc) =
   if lo > hi then 0
   else  (f lo) + sum(f, (inc lo),hi) (*function can be whatever*)
```

```
sumInc:  (int -> int)*int*int*(int->int))->int
```

Example 61 (Product)**.**
What if instead of creating a sum of every thing, we would like to multiply all the elements instead?

```
let rec product(f,lo,hi,inc) =
   if (lo > hi) then 1
   else (f lo) * product(f, (inc lo), hi, inc)
```

But now we've written code similar for sum and product!!. We can abstract even further.

Example 62 (**Accumulator**)**.**
Instead of combining the cumulative results with addition or multiplication, we can instead abstract this as a combiner operation, which is also a parameter.

```
(*comb: combiner, f: , inc:incrementor*)
let acc (comb, f, lo, hi, inc, unit) =
   let rec helper(a, tmp) =
      if (a > hi) then tmp
      else
         helper((inc a), comb(tmp, (f a)))
   in
   helper (lo, unit) (*start at low, and call with unit parameter*)
```

What is the type of this function?

```
val acc :  ('a * 'b -> 'a) * ('c -> 'b) * 'c * 'c * ('c -> 'c) * 'a -> 'a = <fun>
```

Note 5.
The output of a function can also perhaps be a function!

Example 63 (**Derivatives in calculus**)**.**
Recall the limit definition of the derivative:

$$\frac{df}{dx} = \lim_{\epsilon \to 0} \frac{f(x+\epsilon) - f(x)}{\epsilon}$$

Hair Parra 2020 ©

But computers can't take actual limits! Instead, we approximate them with finite differences. Instead we implement

$$\frac{df}{dx} \approx \frac{f(x+dx) - f(x)}{dx}$$

where dx is small.

```
let deriv(f, dx) =
   (*fun is a promitives, produced at runtime*)
   fun x -> ((f (x + .dx)) -. (f x)) /. dx
```

Type: `(float * float) * float -> (float -> float)`

Example 64 (**IterSum**).
Floating point version of what we wrote before

```
let itersSum(f, lo, hi, inc) =
   let rec helper(x, result) =
      if (x > hi) then result
      else
         helper((inc x), (f x) +. result)
   in
   helper(lo, 0)
```

Example 65 (**Integral**).
QUESTION: How can we write an indefinite integral? Just from the definition!

```
let integral (f, lo, hi, dx) =
   let delta x = x +. dx in
      dx *. iterSum(f , (lo +.(dx/.2)), hi, delta)
```

That is, we are writing a **definite integral**

$$\int_{lo}^{hi} f\,dx$$

Figure 13: Signals before and after smoothing

Hair Parra 2020 ©

Example 66 (**Smoother**)**. Signals are usually noisy**. But if we ignore the 'noise', we see that there is some perfectly well behaved function.

```
let smooth f delta =
  fun x ->
    (f (x -. delta)) +. (f x) _. (f (x +. delta)) /. 3.0
```

8.2 Currying (Haskell Curry)

Haskell Curry invented something called **combinatorial logic**

Definition 25 (**Currying**)**.**
The following isomorphism:
`A*B -> C = A -> (B->C)` holds.

Example 67 (**???**)**.**

```
let myadd (m,n) = m + n
val myadd = int * int -> int

myadd 3 -> type error

let myadd2 m n = m + n
val myadd2 = int -> int -> int

myadd2 3 -> fun of type int->int
```

8.3 Pipes

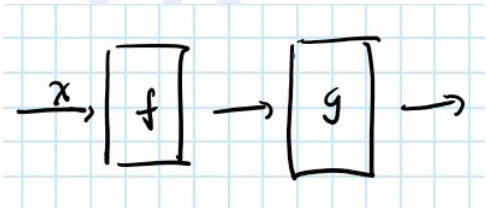

Figure 14: Representation of pipes

Definition 26 (**Pipe**)**.**
A pipe is defined as follows:
`input |> f = f input`
It sends it's input to some function.

Hair Parra 2020 ©

Example 68.

```
(*regular function*)
let inc n = n+1;;
List.map inc [1;2;3] --> [2;3;4]

(*instead we can do this*)
[1;2;3] |> (List.map inc)
```

Example 69 (**List.fold and Pipe**).
We want to add numbers $1, \ldots, 10$, or square all numbers in a list.

```
(1--10) |> (List.fold_left (+) 0) (*outputs 55*)
(1--5) |> (List.map (fun n -> n*n)) |> (List.fold_left (+) 0) (*squares list, adds it*)
```

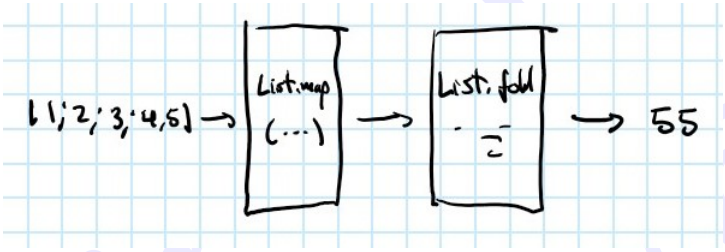

Figure 15: Pipe with map and fold

But how do we define "–" ? (It's supposed to create a list.)

Example 70 (**Function composition**).

```
let compose f g ->
   fun x -> x |> f |> g
```

8.4 How do define infix operators?

Definition 27 (**Infix Operator**).
This produces an enumerator.

```
let (--) lo hi = List.init (hi-lo+1) (fun n ->n + lo)
```

Hair Parra 2020 ©

9 Lecture 8: 2020-01-30

9.1 Binary Search Trees

Definition 28 (**Binary search tree**).
A node n has a subtree L to the left, and everything in it is less than n, everything in a right subtree R is more than n. A binary tree can be defined in OCaml as

```
        (* Binary Search Trees coded from scratch.  *)
type 'key bstree = Empty | Node of 'key * 'key bstree * 'key bstree
```

Example 71 (**Find**).
We want to do the following:

```
let rec find less x (t : 'key bstree) =                    '
  match t with
    | Empty -> false
    | Node(y,left,right) ->
        if (less(x,y)) then find less x left
        else
          if (less(y,x)) then find less x right
        else (* x = y *) true;;
```

1. Less is a function that compares $x < y$
2. If the node is empty, return `false`
3. Else, match thre tree t with a node y with left and right nodes.
4. If $x < y$ then recurse on the left subtree.
5. If $x > y$ then recurse on the right subtree.
6. Else $x = y$, so return true.

Example 72 (**Insert**).

```
let rec insert less x (t: 'key bstree) =                '
  match t with
    | Empty -> Node(x,Empty,Empty)
    | (Node(y,left,right)) as nn ->
        if (less(x,y)) then Node(y,(insert less x left),right)
        else
          if (less(y,x)) then Node(y,left,(insert less x right))
        else (* x = y *) nn;;
```

Hair Parra 2020 ©

1. Define `rec` with parameters `insert`, a function `less` , a value x to insert, and a tree t.
2. Else if t has some node y with left and right, call this nn then
   (a) If $x < y$ then go to insert the node on the left tree
   (b) Else if $x > y$ then Insert on the right subtree
   (c) Else $x = y$ so just return nn

Example 73 (**Delete Min**).

```
exception EmptyTree

let rec deletemin (t: 'key bstree) =                    '
  match t with
    | Empty -> raise EmptyTree
    | Node(y,Empty,right) -> (y,right) (*delete head, return head and right tree*)
    | Node(y,left,right) ->
       let (z,ll) = deletemin(left) in
       (z,Node(y,ll,right));;
```

Example 74 (**Delete Less**).

```
let rec delete less x (t: 'key bstree) =                  '
  match t with
    | Empty -> raise EmptyTree
    | Node(y,left,right) ->
       if (less(x,y)) then Node(y,(delete less x left),right)
       else
         if (less(y,x)) then Node(y,left,(delete less x right))
       else (* x = y *)
         match (left,right) with
           |  (Empty,r) -> r
           |  (l,Empty) -> l
           |  (l,r) ->
              let (z,r1) = deletemin(r) in Node(z,l,r1)
```

9.2 First Order Predicate Calculus

- **FREGE** invented first order predicate calculus; he brought the idea of substitution of **expressions** for **variables**.
- **HILBERT** (1900) Give an effective procedure for solving Diophartive equations; consider the equation below , but insist on particular kinds of solutions (ex. $\mathbb{N}$).

Hair Parra 2020 ©

$$y^2 = x^3 + 7$$
$$x^n + y^n = z^n$$

(Fermat's Last Theorem)

- **KURT GODEL** proved that there are true statements for which no proof is not possible.
- **ALONZO CHURCH** invented the **lambda calculus**. (λ-calculus). He also proved unsolvability of the **halting problem**. (evolved into OCaml).
- **ALAN TURING** Proposed the **Turing Machine**, and also proved the **unsolvability of the halting problem**. He also showed the Turing Machine was **equivalent** in expressive power to the λ-calculus, equivalent to the **RAM machine**, equivalent to **Chomsky Type 4**.

Definition 29 (**Lambda Calculus**)**.**
The lambda Calculus has only 3 ingredients:

- Functions
- Variables
- Applications

Also, there are **no types** (produces inconsistency). Recursion makes it **Turing Complete**.

Example 75 (**Simple Turing Complete Language**)**.**

```
x,y,z, ...
   fun x -> (...)
      f x
```

Example 76 (**Creating a simple programming language?**)**.**

- functions are values
- arithmetic values
- pairs (data structure)
- choice (if-then else)
- recursion

Hair Parra 2020 ©

1. Coding booleans:

```
let tt = fun x -> fun y -> x
let ff = fun x -> fun y -> y

ttMN = (fun y -> M) N = M
ffMN = (fun x -> fun y -> y) MN
     -> (fun y-> y) N -> N
```

2. Coding pair:

```
let makePair = fun x -> fun y
          fun z -> z xy

makePair M N = fun z -> z M N

let ch-fst p = p tt
let ch-snd p = p ff

ch-fst (MakePair M N)
= (fun z -> z M N ) tt
= tt M N = M
```

3. **Church numerals**

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

(*succesor function: takes the chruch numeral, and produce the next one*)
let succ cn = fun x -> (fun x -> (f ((cn f) x)))
```

4. Coding arithmetic

```
(*addition*)
(*get m applications of f and then n applications of f to that*)
let add n m =
  fun f -> (fun x -> ((n f) (((m f) x )))

(*multiplication*)
let tines n m =
  fun f -> (fun x -> fun x -> (n (m f) x))

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

Hair Parra 2020 ©

5. Recursion? Use the `Y combinator`

```
two two two two two two two (stack of two's)
```

Definition 30 (**Y-Combinator**).
`YM -> M (YM) -> M (M (YM))` (fixed point combinator).

Note 6 (**Church Numerals**).

1. Church numerals are the number of times we apply a function.
   Ex. `let two = fun f -> fun x -> f(fx)` is a **composition of functions:** $(f \circ f)(x) \equiv f(f(x))$.

2. More generally,
$$f^n(x) = \underbrace{(f(f(f \dots f}_{n-times}(x)))) = \underbrace{(f \circ f \circ f \cdots \circ f)}_{n-times}(x)$$

3. The definition of **Church numerals** is that
$$nfx = f^n(x)$$
   This implies that , for instance, $2fx$ can represent `two`

4. For the function `add n m`, we want to apply the function $(n + m)$ times, that is
$$f^{(n+m)} = f^n(f^m(x)) = nf(mfx)$$

5. For the function `times n m`, we want to apply a function $(n * m)$ times, that is
$$f^{n*m} = (f^m)^n = n(f^m)x = n(mf)x$$

6. For the function **exp n m**, we want to apply a function (n^m) times, that is $f^{(n^m)}(x)$. Using the definition of Church numerals, $n^m(x)$ can be rewritten as mnx. Thus,
$$f^{(n^m)} = (mn)fx$$

10 Lecture 9: 2020-02-04

10.1 Imperative Programming

Definition 31 (**Imperative Programming**).
In imperative progamming you give orders or **commands**.

Definition 32 (**Command**).

Hair Parra 2020 ©

1. A **command** changed the state of the system.
2. New data type: `mutable storage`.
3. We can now issue a command to update a storage cell.
4. NOTE: **commands** are considered to be special expressions.
5. We need a special "dummy" value:
6. `value : type === () : unit`

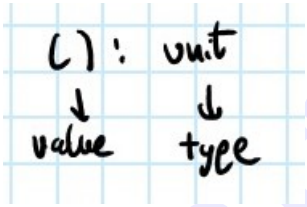

Figure 16: Value-type

Example 77 (**Ref**)**.**

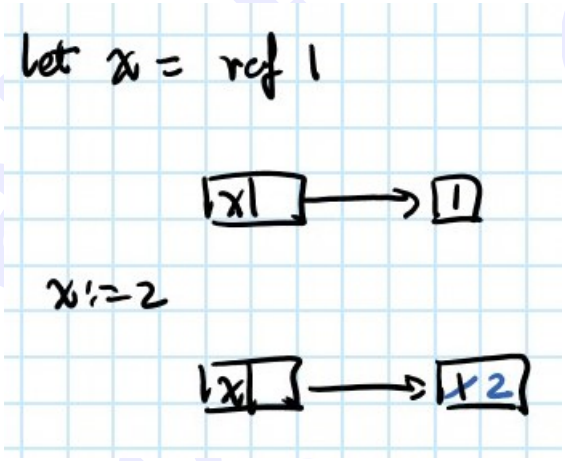

Figure 17: Ref example of assignment.

```
let x = ref 1

val x: int ref = {contents = 1}
```

- A record with one mutable field = *cell* on a location.
- `ref` plays 2 roles:
    1. It is a **type constructor**

2. It is also a **value constructor** (e.g. `ref 1`)

- To update `x := 2`
  1. `x` should be bound to am memory cell
  2. Go and find it, type check: `x` should be a `mutable variable`
  3. Go and change what's inside the celle.
- How do we check what's in the cell?

```
(*Like this? *)
(): unit

(*See what is actually inside*)
!x
- : int 2
```

- " `!` " is called the **dereferencing operator**

Example 78 (**Evaluation rule**)**.**
The following is the process of evaluation for an imperative expression.

- `x` α`-ref`
- `exp` : α

```
x := exp
```

1. Evaluate `x` and make sure it is a location
2. Evaluate `exp` to produce a value `v`
3. Store `v` in the location pointed to by `x`

NOTE:

- Information has been thrown away: old value of `x`
- The programmer now has responsibility for the lifetime of data.
- **Time** has appeared in the picture $\implies$ notion of **Sequence**
- The **sequence operator** in OCaml is " `;` "

Hair Parra 2020 ©

Example 79 (Traditional programming language).
In other languages than OCaml, we may have

```
x:= x + 1
```

- The symbol `x` means 2 different things:
- The LHS stands for a **location**
- The RHS stands for a `value`
- NOTE: In OCaml, `x` **always means a location**

```
(*Increment value of x*)
x:= !x + 1
```

Example 80 (**Equality**).

Do both cells store the same value?

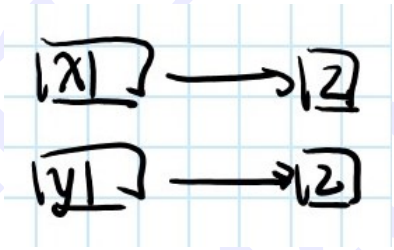

Figure 18: `x` and `y` are pointers to the value 2 in each case.

```
let y  = ref 2
x = y
true
```

- **NOTE**: New notation: `==` means "*do these names stand for the same location?*"

```
let u = ref 6 ;;
let v = ref 7 ;;
!u;;  (*obtain valiue of u*)
7
!v  (*values of v*)
7
u=v  (*values are equal*)
```

Hair Parra 2020 ©

```
true
v:=8
u==v  (*locatins are different*)
false
```

Example 81 (**Aliasing**).

```
let u =ref 7
let v = u
```

- The phenomenon above is called **aliasing**: two names for the same cell

```
!u
7
!v
7
u = v
true

v:8
!u
8
u=v
true

let w = ref 8
u=w
true

u==v (*same location? yes!*)
true
u==w  (*same location? no!*)
false

u := u + 1
TYPE ERROR (*should be u := !u + 1*)
```

Example 82 (**List and dereference**).

```
let l = ref [1;2;3]

(*how to append a 0 at the beginning? *)
l := 0::(!l)
```

Hair Parra 2020 ©

NOTE: `!` is the dereference operator

Definition 33 (**Records with multiple mutable fields**).
Creation syntax: `type point = {mutable x:int, mutable y:int }`

```
type point = {mutable x:int, mutable y:int }

let p = {x=3 ; y = 4}
p.x (*automatic deref*)
value: 3
```

Example 83 (**Move**).

```
type point = {mutable x:int, mutable y:int }

let p = {x=3 ; y = 4}
p.x (*automatic deref*)

(*add a and b to both operators*)
let move (p:point) (a,b) =
   (p.x <- p.x + a) ; (p.y <- p.y + b)
```

NOTE: `<-` is an alternative syntax for updating.

10.1.1 Difference between mutation and functions

Example 84 (**Modify**). We want to alter the value by adding one; i.e., fetch the value and add 1 (an expression), and assign this value to the location n.

```
let modify n =
   (n := !n + 1)
 -: int ref -> unit

 !x
 17
 modify x ;;
 ()
 !x
 18
```

Hair Parra 2020 ©

Example 85 (**IntUpdate**).

```
let incUpdate (n:int) =
   let m ref n in (m:= ! m + 1) ;;
- : int -> unit

let  q = 37 ;;
incUpdate q ;;
();
q ;;
37 (*Why didn''t it get updated?*)
```

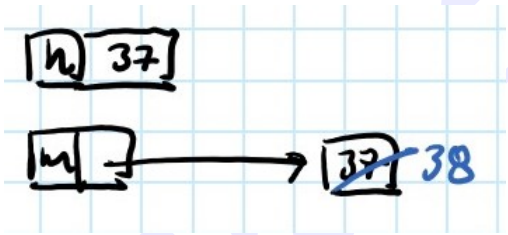

Figure 19: IntUpdate update

NOTE: `incUpdate` is absolutely useless!

Example 86 (**Mash**).

Figure 20: Mash function enviornment and results

To see the intermediate changes, we need to insert print statements.

```
let mash (n:int) =
   let m = ref n in
   (Printf.printf "m is %i " !m);
   (m := !m + 1);
   (Printf.printf "n is %i " n);
   (Printf.printf "m is %i" !m);;
val mash : int -> unit = <fun>

mash 3;;
```

```
m is 3
n is 3
m is 4
- : unit = ()
```

Example 87 (**While loop**).
This is the only time you will see while loops in this class.

```
let foo n =
   let x = ref n in
   while(!x < 10) do
      (Print .... x ) ;
      (x := !x + 1 )
   done

foo 1; (*is sequencing*)
```

- Memory can not be updated
- The programmer is responsible for lifetime of data
- Time enters the computation picture
- Commands appear as a new semantic category

11 Lecture 10: 2020-02-06

11.1 Pointers

How to list lists as records and pointers ? (`pointers = refs`)

- How do we express something that may be a pointer or may be nil ?
- Recall option types:

```
A -> type
A option -> type containing two sorts of values
a: A
Some a: A option
None: A option
```

Hair Parra 2020 ©

Example 88 (**Linked Lists**).
We work with a mutually recursive definition.

```
(*rlist = 'reference to list'*)
type cell = {data: int; next: rlist}
and rlist = cell option ref ;;

let c1 = {data = 1; next = ref None};;
let c2 = {data = 2; next = ref (Some c1)};;
let c3 = {data = 3; next = ref (Some c2)};;
let c5 = {data = 5; next = ref (Some c3)};;

(* This converts an rlist to an ordinary list. *)
let rec displayList (c : rlist) =
  match !c with
    | None -> []
    | Some { data = d; next = l } -> d :: (displayList l);;
val displayList : rlist -> int list = <fun>

let cell2rlist (c:cell):rlist = ref (Some c);;
val cell2rlist : cell -> rlist = <fun>

displayList (cell2rlist c5) ;;
- : int list = [5; 3; 2; 1]
```

Example 89 (**Imperative Reversing a List**).
We want to sort the list **in place**.

```
type cell = {data: int; next: rlist} (*rlist = reference to list*)
and rlist = cell option ref ;;

let reverse (lst: rlist) =
  let rec helper ((l: rlist), (acc: rlist)) =
    match !l with
      | None -> acc
      | Some c when !(c.next) = None ->
           (c.next := !acc; acc := (Some c); acc)
      | Some c ->
           (l := !(c.next); c.next := !acc; acc := (Some c); helper(l,acc))
  in
  (helper(lst, ref None));;
```

- Define the parameters to be a list, `(lst: rlist)`
- Define a recursive helper on with a list and an accumulator.
- NOTE: `l` is a reference! (so need to write with `!l`).

Hair Parra 2020 ©

- Match the value of `c` with`None`, if that's the case, then , reassign `c.next`, let the accumulator be the value of `Some c`, and return the accumulator.
- If the value of `!(c.next)` is not `None`, then `l` gets assigned the value of `c.next`, then `c.text` gets the value of the accumulators, the `acc` gets the value of `Some c`, and then we recurse with `l` and the accumulator.
- We call with the base case.

11.2 Exceptions and backtracking programming

Definition 34 (**Exception**).
Exceptions are modern form of the statement `go to`.

Example 90 (**Quadratic equations**).
Recall the quadratic equations:

$$ax^2 + bx + c = 0$$

Which leads to the `quadratic formula`.

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

```
let solve(a,b,c) =
  let disc = (b *. b -. 4.0 *. a *. c) in
  if  disc < 0.0 || a = 0.0 then
    failwith "The discriminant is negative or a was zero."
  else
    ((-.b +. sqrt(disc))/.(2.0*.a),(-.b -. sqrt(disc))/.(2.0*.a));;
```

Instead, we can define **exceptions**

```
exception Complex;;
exception Linear;;
```

Hair Parra 2020 ©

Example 91 (**Solve_quad**).

```
exception Complex;;
exception Linear;;

let solve_quad(a,b,c) =
  let disc = (b *. b -. 4.0 *. a *. c) in
  if  a = 0.0 then
    raise Linear
  else
    if disc < 0.0
    then raise Complex
    else
      ((-.b +. sqrt(disc))/.(2.0*.a),(-.b -. sqrt(disc))/.(2.0*.a));;
```

11.3 Exception Handling

Example 92 (**Solve quad with Catch**).

```
(*Print on the screen some text using solve_quad*)
let solve_text(a,b,c) = let (r1,r2) = solve_quad(a,b,c) in
                  "Roots are: " ^ (Float.to_string r1)^" and "^(Float.to_string r2);;

 let solve_catch(a,b,c) =
  try
    solve_text(a,b,c)
  with
  | Linear -> "Degenerate linear equation"
  | Complex -> "The roots are complex";;
```

Hair Parra 2020 ©

Example 93 (**Catch**).

```
(*Print on the screen some text using solve_quad*)
let solve_text(a,b,c) = let (r1,r2) = solve_quad(a,b,c) in
                  "Roots are: " ^ (Float.to_string r1)^" and "^(Float.to_string r2);;

 let solve_robust(a,b,c) =
  try
    let disc = (b *. b -. 4.0 *. a *. c) in
    if  a = 0.0 then
      raise Linear
    else
      if disc < 0.0 then
      raise Complex
      else
        ((-.b +. sqrt(disc))/.(2.0*.a),(-.b -. sqrt(disc))/.(2.0*.a))
  with
  | Linear -> (print_string "This is not a quadratic you idiot!");
              (-. c /. b, -. c /. b)
  | Complex -> (print_string "The roots are complex. The real and imaginary parts are:");
              let disc = (b *. b -. 4.0 *. a *. c) in
              let realpart = -. b/.(2.0 *. a) in
              let imagpart = (sqrt(-.disc))/.(2.0 *. a) in
              (realpart,imagpart);;
```

11.3.1 Backtracking

Example 94 (**Backtracking**).

```
exception Change;;

let makeChange coinTypes amt =
  let rec helper((coins: int list), (amt:int)) : int list =
    match (coins, amt) with
    | (_,0) -> [] (**)
    | ([],_) -> raise Change (*Can''t make change with the current coin set*)
    | (coin::rest,amt) -> (*individual coin and rest of coins*)
       try
         if
           (coin > amt)
         then
           helper(rest,amt)
         else
```

Hair Parra 2020 ©

```
          coin::(helper(coins, amt - coin))
        (*handle the exception = backtracking*)
        with
          | Change -> helper(rest,amt)
  in
  try
    let ch = helper(coinTypes,amt) in
    ("Return the following coins: "
     ^ (List.fold_left (fun s -> fun n -> (string_of_int n) ^ " " ^ s) " " ch) ^ "\n")

    with
      | Change -> "Sorry, I cannot make change.\n"
```

- `coins` List with coin types in increasing order
- **amt** is the amount to make change for

11.4 Some tricky list examples

Example 95 (**Partial sum of a list**).

```
[1;3;2;5] -> [1;4;6;11]

let psums lst =
  let rec helper l a =
    match l with
      | [] -> [a] (*done,return acc*)
      | x ::xs -> a :: (helper xs (x + a)) (*update acumulator*)
  in
  match lst with
    | [] -> [0] (*convention*)
    | x :: xs -> helper xs x;;
```

Example 96 (**Smash**).

```
let smash ll = List.fold_left (@) [] ll;;
```

Example 97 (**Interleave**).
Wqnt to produce a list of lists in which 0 has been inserted in every possible position.

```
0 [1;2;3] -> [[0;1;2;3]; [1;0;2;3] ; [1;2;0;3] ; [1;2;3;0]]
```

Hair Parra 2020 ©

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

- In the last list, we want to map the insertion to every single thing.

Example 98 (**Permutation**).

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

12 Lecture 11: 2020-02-06

12.1 Mutability

Definition 35 (**function**).
An association between certain values from some input space to some output space.
It should only be dependent on the input.

Example 99 (**flip0**).

```
let a = ref 0;;
let flip0 () =
   (a := (1 - !a));   (*imperativeky update value of a*)
   (Printf.printf "%i\n" !a);;

val flip0 : unit -> unit = <fun>

flip0 ()
1
flop0 ()
0
```

- `a` gets bound to 0 (by a pointer).
- Input should be of `unit type`. It will define a function.

Hair Parra 2020 ©

- `a` is a **global variable**; i.e. is **not protected!**
- We lost the purity of a mathematical function!
- flio0 does not have exclusive access to `a`

Note 7.

- Want to have the ref value encapsulated to only the "function" can access it.

Example 100 (**flip**).

```
let a = ref 0;;

let flip =
   let c = ref 0 in
   fun () -> c:= (1 - !c);
   (Printf.printf "%i\n" !c);;

let flip2 =
  let c = ref 0 in
  fun () -> (c := 1 - !c);
  (Printf.printf "%i\n" !c);;
```

- Here the variable is **protected!**
- This thing has a **trapped local mutable variable**
- This happens because the left occurs inside the definition of the function .
- The functions `flip1` and `flip2` do not affect each other.

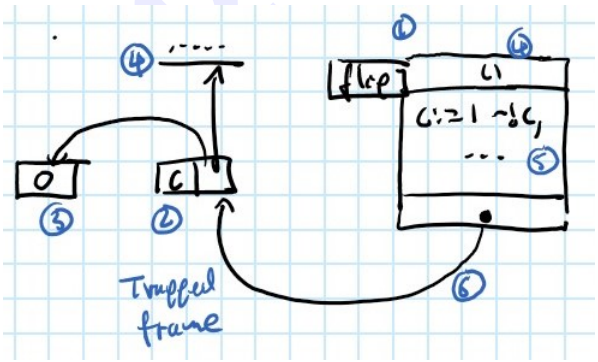

Figure 21: Environment and trapped frame.

Hair Parra 2020 ©

Example 101 (**Make Flipper**).
Instead of creating a filp() each time, we can construct a function that creates flip() 's.

```
let make_flipper = fun () ->
  let c = ref 0 in
  fun () -> (c := 1 - !c);
  (Printf.printf "%i\n" !c);;

val make_flipper : unit -> unit -> unit = <fun>

let flip3 = make_flipper ();;
val flip3 : unit -> unit = <fun>

let flip4 = make_flipper ();;
val flip4 : unit -> unit = <fun>
```

- We use higher-order functions
- It produces a filpper (a function) as an output.
- `make_flipper` is an **object generator**
- `flip1, flip2,...` are **instances**.

Example 102 (**flop :(**).

```
let flop =
  fun () -> let c = ref 0 in (c := 1 - !c);
  (Printf.printf "%i\n" !c);;

flop ();
1

flop() ;
1
```

- DOESN'T WORK!
- The `let` statement is **inside the body of the function**.
- i.e. The value of c is redefined each time to be 1 !

Hair Parra 2020 ©

Example 103 (**Bank Account Generator**).

```
(* Datatype of bank account transactions.*)
type transaction = Withdraw of int
                 | Deposit of int
                 | Checkbalance

type transaction = Withdraw of int | Deposit of int | Checkbalance

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

val make_account : int -> transaction -> unit = <fun>

(* Make some eaccounts *)
let rosie = make_account(6000);;
val rosie : transaction -> unit = <fun>

let marta = make_account(5000);;
val marta : transaction -> unit = <fun>

let prakash = make_account(25) ;;
val prakash : transaction -> unit = <fun>

rosie(Withdraw 500) ;;
Balance is 5500
- : unit = ()

marta(Deposit 500);;
Balance is 5500
- : unit = ()

prakash(Withdraw 1000);;
Insufficient funds.
```

Hair Parra 2020 ©

```
- : unit = ()

prakash(Checkbalance) ;;
Balance is 25
- : unit = ()
```

1. Need different types of transactions, all packed in the same type.
2. `balance` is trapped.
3. Define the body of the function: `fun (t:transaction)`, i.e. argument is a transaction.
4. If it's a **Withdraw**, if the balance is more than m, decrement and print it to the screen. Note `m` is the amount to withdraw.

- The `balance` mutable variable has to be **protected**, it is a trapped mutable frame.
- When creating multiple accounts, they don't interfere with each other's balance. =

Example 104 (**Monitoring**).
Want to monitor how something is behaving.

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


let makeMonitored f =
    let c = ref 0 in (*local but updatable*)
    fun x ->
      match x with
        | Query -> (Numcalls !c)
        | (Normal y) -> ( c := !c + 1; (Ans (f y)));;
val makeMonitored : ('a -> 'b) -> 'a tagged -> 'b answers = <fun>


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

let inc n = n + 1

let moninc = makeMonitored(inc)
```

- This type of function is called a **wrapper**: it takes some other function inside but does not mess up with the code.
- `tagged` and `answers` are polymorphic types.

Hair Parra 2020 ©

13 Lecture 12: 2020-02-06

13.1 Midterm Review

Example 105 (**Question 1: HIgher-order function**).
Write a higher-order function such that .

$$(f n) \to \underbrace{f \ldots f}_{n-times}$$

```
(*question*)
let rec repeated (f,n) = ... <code > ...
'a -> 'a * int -> 'a -> 'a

(*bad solution1*)
let rec repeated (f,n) =
   if n=0 then f
   else f (f, n-1)

(*bad solution2*)
let rec repeated (f,n) =
   if n =0 then f
   else f (repeated (f, n-1))

(*bad solution3*)
let rec repeated (f,n) =
   if n=0 then f
   else
      fun x -> repeated (f, n-1) (f x) (*crucial step*)

(*good solution*)
let rec repeated (f,n) =
   if n=0 then fun x -> x (*apply 0 times means identity! *)
   else
      fun x -> f((repeated(f,n-1)) x)
      (*fun x -> repeated (f, n-1) (f x)* (*crucial step*)
```

- Takes a
- bad solution 1: not even recursive, not doing the right thing, wrong case
- bad solution 2: does recurse, but not right! , doesn't type check! n=0 is also wrong
- bad solution 3: still wrong, (base case), but returning is better!

Hair Parra 2020 ©

- good solution: does type check, correct base case, correct recursion

Example 106 (**Question 2:Data structures and Recursion**).
Represent arrays as **lists of lists**.
A matrix is said to be **proper** if all rows have the same length.
It is **square** if it is proper and the length of each row = number or rows.
Write a program to test whether a list of lists is a square matrix.4

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

```
(*examples of arrays*)
m1 = [[1;2;3];[4;5;6];[7;8;9]] (*kind of like a matrix*)

(*solution*)
let square m =
   match m with
   |[] -> true
   |_ ->
      List.for_all (fun r -> (List.lenfth m) = (List.length r) ) m


(*from scratch solution, wrong but on the right track*)
let square2 m =
   let l = List.length m in
      match m with
      |[] -> true
      |first::rest -> (*wrong! pattern matching outside the recursion!*)
         let rec helper mat =
            if (List.length first) = l then
               helper rest
            else
               false
         in
         helper m
```

- `List.for_all` takes a predicate to each element of the list, returns true if all of them pass the test.
- `r` stands for row.
- It is testing if testing whether each row has the length of the whole list of lists.
- NOTE: Write basic `List.module functions!!!`

Hair Parra 2020 ©

- **second solution:** Kinda right, buf pattern matchinf outside the recursion; goes into an infinite loop!!!
- Below the right solution

```
let square3 m =
   let l = List.length m in
      match m with
      |[] -> true
      |_  ->
         let rec helper mat =
            match mat with
            |first::rest
               if (List.length first) = l then
                  helper rest
               else
                  false
         in
         helper m


let square4 m =
   let l = list.length m in
      let rec helper mat =
         match mat with
         |[] -> true
         |first::rest ->
            if (List.length first) = l then
               helper rest
            else
               false
      in
      helper m
```

Example 107 (**Proper List**)**.**

```
let proper2 m =
   match m with
   |[] -> true
   |first::rest ->
      let l = List.length first in
      .
      .
      .
      (*check every row has the same length*)
```

Hair Parra 2020 ©

Example 108 (**Environment Diagrams**).

```
let x = 1 in
   let  y =x in
      let x =2 in
         let f u = u + x + y in
            let x = 4 in
               f x
```

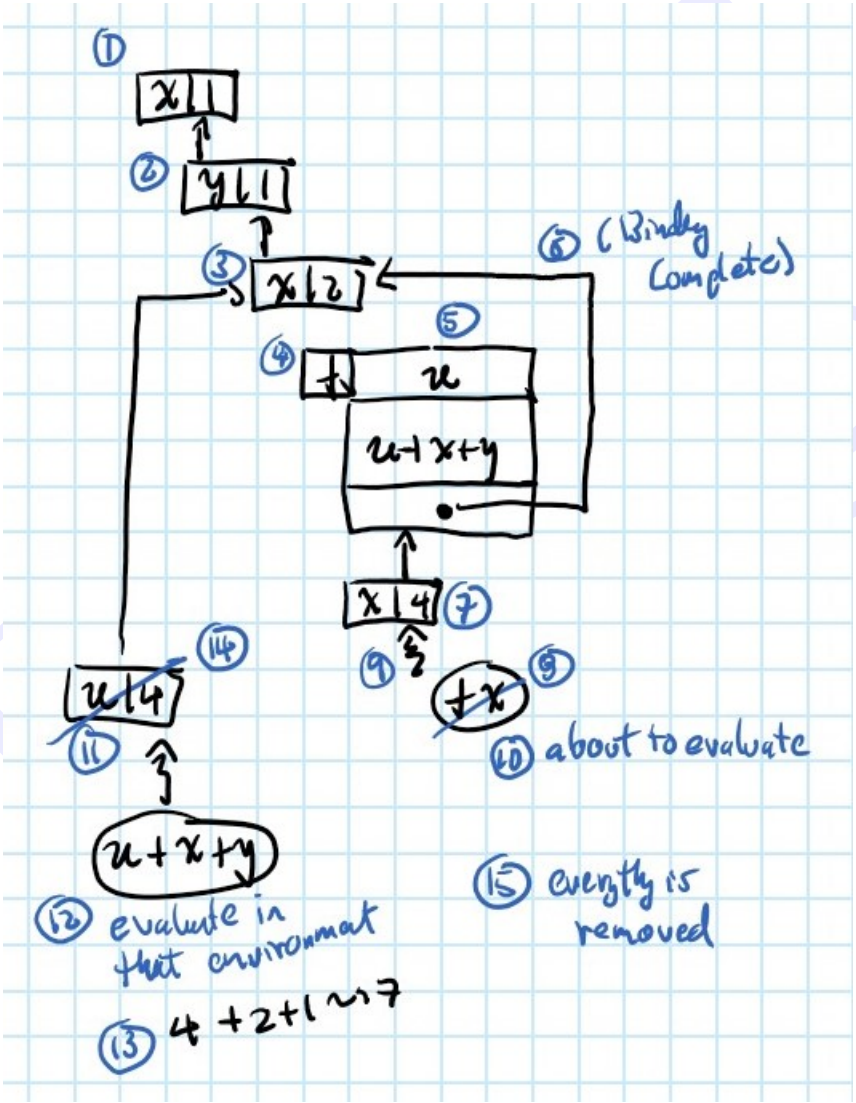

Figure 22: Environment diagram for the expression above

Example 109 (**Environment Diagrams: Trapped frame**).

```
let x = 2 in
   let f = let c=3 in
      fun u -> u+1
   in f x
```

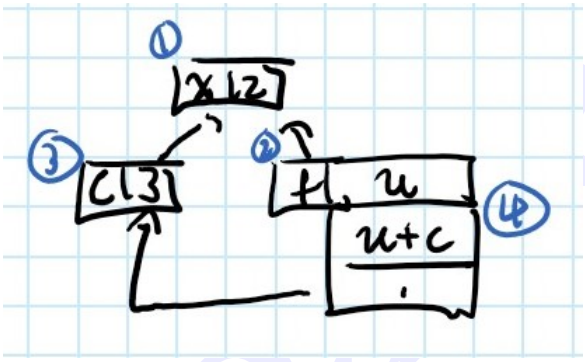

Figure 23: Environment diagram for a trapped frame

Example 110 (**More env diagrams**).

```
let x = 1 in
   let f = fun n -> (let y=x in n+y) in
      let z = (f x) in (f z);;
```

See figure 24

Figure 24: Environment diagram for example 110

Hair Parra 2020 ©

Example 111 (**Proper Matrix**).
A matrix is called **proper** iff all it's rows have the same length. Write a function that checks whether a list of lists is a proper matrix.

```
let proper m =
   match m with
   |[] -> true
   |first::rest ->
      let rec aux (f1, r) =
         let len1 = List.length f1 in
            match r with
            |[] -> true
            |l::ll ->
               if (List.length l) = len1 then
                  aux(l,ll)
               else
                  false
      in
      aux(first, rest)
;;
```

14 Midterm: 2020-02-18

14.1 Question 1

Write a function to remove all consecutive duplicates in a list

```
let rec remove_consec_dup l =
  match l with
  | [] -> []
  | a::[] -> [a]
  | a ::(b :: more) -> if a = b
                       then
                         remove_consec_dup (b :: more)
                       else
                         a:: (remove_consec_dup (b :: more));;
```

14.2 Question 2

Recall the code

Hair Parra 2020 ©

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

let integral((f: float -> float),(lo:float),(hi:float),(dx:float)) =
  let delta (x:float) = x +. dx in
  dx *. iterSum(f,(lo +. (dx/.2.0)), hi, delta)
;;
```

A **convolution** of two functions f and g is a function $\mathbb{R} \to \mathbb{R}$ defined as follows:

$$(f \circledast g)(x) = \int_{-\infty}^{+\infty} f(y)g(y-x)dy$$

Write the above function in OCaml. You can use `min_float` and `max_float` for $-\inf$ and $+\inf$

14.2.1 Solution

```
let convolution f x =
   let delta = 0.000000000000001 in
      f (x:float) ->
         let conv -> fun (y:float) -> (f y) *. (g (y -. x)) in
         `  integral (conv, min_float, max_float, delta )

val convolution: (float -> float) -> (float -> float) -> float -> float = <fun>
```

14.3 Question 3

COnsider the following expression:

```
let x = 1 in
   let y = x in
      let rec f n =
         if n = 0 then 0
         else y + (f(n-x))
      in
      let x = 3 in f x ;;
```

Hair Parra 2020 ©

Questions:

1. What is the value of the first binding of `y` ?
2. Is `y` bound at each recursive call of `f` ?
3. A student who skipped class says: "The vale of this expression is 2 because when you call f with 3, n gets bound to 3, and so `y + (f (n-x)) = 2 + (f (3-3)) = 2`". What is wrong with this logic?
4. What is the correct final value of the expression?

15 Lecture 13: 2020-02-20

This section will talk about the **Formal Definition of Languages and Types: Processing Programming Languages**

15.1 Semantics

Definition 36 (**Semantics**).
Semantics is the **Theory of meaning**.

- In natural language, this is extremely complex and contains a lot of subtle aspects.
- **Mathematical theory**:
  1. **Denotational semantics**
  2. **Operational Semantics**: formal, but it gives step-by-step execution of programs. This id defined by induction on the structure of programs.

Note 8 (**Values and expressions**).
Note that $values \subset expressions$

- 1729 is a **value**
- $10 * 10 * 10 + 9 * 9 * 9$ is an **expression**. **Evaluation** produces $\rightarrow$ 1729

Example 112 (**NanoML**).
We will define a small language using **inductive definitions**.

Hair Parra 2020 ©

- **base cases of the inductive definition:**

```
nanoMl
exp ::== n | true | false | e1+ e2 | e1 * e2 |
   if b then e1 else e2 | b1 and b2 | not b (*types ignored for now*)
```

- Notation: $e \downarrow v$ means that e **evaluates to** v
- 1729 is an expression
- 2, 3 are expressions
- $2 + 3$ is an expression
- $(2 + 3) * 1729$ is an expression
- **Rules are given by induction on structure of expressions**
  - **Base:**
    $n \downarrow n$, $true \downarrow true$, $false \downarrow false$
  - **Inductive step:** We have the following format:

$$\frac{\text{ASSUMPTIONS}}{\text{CONCLUSIONS}}$$

1. **addition**

$$\frac{e_1 \downarrow v_1 \quad e_2 \downarrow v_2}{e_1 + e_2 \downarrow v_1 + v_2}$$

2. **boolean expression evaluation**

$$\frac{b \downarrow true \quad e_1 \downarrow v_1}{(\text{if } b \text{ then } e_1 \text{ else } e_2) \downarrow v_1}$$

$$\frac{b \downarrow false \quad e_2 \downarrow v_2}{(\text{if } b \text{ then } e_1 \text{ else } e_2) \downarrow v_2}$$

Theorem 15.1. *Every expression in NanoML terminates.*

Theorem 15.2. *If $e \downarrow v_1$ and $e \downarrow v_2$ then $v_1 = v_2$*

Example 113 (**MicroML**).
We now extend the `NanoML` language.

```
nanoMl
exp ::== n | true | false | e1+ e2 | e1 * e2 |
   if b then e1 else e2 | b1 and b2 | not b

microML = nanoML +
     x | let x = e1 in e2
```

Hair Parra 2020 ©

- Consider the expression:

$$let\ x = e_1\ in\ e_2$$

- In the above expression, `x` is bound to the **value** of `e1`. This binding is **only valid** during the evaluation of `e2`.
- `e2` is in the **scope** of this binding.
- NOTE: `x` can occur in `e2` (it probably should), but if we look at `e2` in isolation, it will appear that `x` is an unbound variable. We call this a **free variable**.

Definition 37 (**Free variables in MicroML**)**.**

- Define the set of **free variables** in an expression as

```
FV: Expressions -> Set_of_vars
```

$FV(n) = \varnothing$
$FV(e_1 + e_2) = FV(e_1) \cup FV(e_2)$
$\vdots$
$FV(b_1 \text{ and } b_2) = FV(b_1) \cup FV(b_2)$
$FV(x) = \{x\}$ (where x is a free variable)
$FV(let\ x = e_1\ in\ e_2) = FV(e_1) \cup \{FV(e_2) - \{x\}\}$

- Example:

$$FV(\text{ if } b \text{ then } e_1 \text{ else } e_2) = FV(b) \cup FV(e_1) \cup FV(e_2)$$

Definition 38 (**Free & bound variables: Quantifiers**)**.**

$$\underbrace{\forall x \exists y\ \underbrace{\phi(x,y)}_{x,y \text{ are free}}}_{\text{No free variables}}$$

Example 114 (**Bound variables and binders**)**.**

$$\int_{-\infty}^{+\infty} e^{-x^2} dx = \int_{-\infty}^{+\infty} e^{-y^2} dy = \int_{-\infty}^{+\infty} e^{-z^2} dz = \sqrt{\pi}$$

Here, dx is a **binder**, note that we can replace the variable without changing the expression's meaning.

Hair Parra 2020 ©

Example 115 (**???**).

```
FV( let x = 2 in
      let y = x +1 in
        x + y)
```

$$FV(let\ y = x + 1\ in\ x + y)$$
$$= FV(x + 1) \cup \{FV(x + y) - \{y\}\}$$
$$= \{x\} \cup \{x\} = \{x\}$$

$$FV(2) \cup (FV(\underline{let\ y = x + 1\ in\ x + y}) - \{x\})$$
$$= \{x\} - \{x\} = \varnothing$$

- At top-level, there are no free variables, but inside these there are free variables that got bound.
- A term with no free variables is called a **close term**.

Now consider the following example:

Example 116 (**Substitution in environment**).

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

- First that happens: `let x =2 in x * (x+1)`

Definition 39 (**Substitution**).

$[e/x]e'$ means "replace free occurrences of x in e' with e"

e.g. $[x + 1/y](y + y) \to (x + 1) + (x + 1)$

Example 117 (**Some substitution examples**).
$[e/x]\ n \to n$
$[e/x]\ x \to e$
$[e/x]\ (e_1 + e_2) \to ([e/x]e_1) + ([e/x]e_2)$
$[e/x]$ (let $u = e_1$ in e_2) $\to$ let $u = ([e/x]e_1)$ in $([e/x]e_2)$ (DANGER!!)

Hair Parra 2020 ©

- u may occur free in e.
- If we substitute e for x in e_1 the free u becomes **captured**.
- To avoid capture, we rename local bound variables by using a fresh name. (Generating fresh names)

Example 118 (Generating a fresh name)**.**

$$[u+5/z](let\ u = x + 17\ in\ u + z)$$

$$let\ u = x + 17\ in\ u + (\underbrace{u}_{\text{CAPTURED}} + 5)$$

i.e., we rename the inner `u` to a new fresh name

Definition 40 (**Semantics of let**)**.**

$$\frac{e_1 \downarrow v_1 \quad [v_1/x]e_2 \downarrow v_2}{(\text{let } x = e_1 \text{ in } e_2) \downarrow v_2}$$

Theorem 15.3. *All terms in* `microML` *terminate.*

Theorem 15.4. *No matter in which order you perform substitution, you get the same answer.*

Definition 41 (**Mini-ML**)**.**
We extend the `Micro-ML language`

- `MiniML = MicroML + fun x -> e | e1 e2`
- `fun x -> e` is a value even if `e` is not.
- `fun` is a binder
  $FV(fun\ x \to e) = FV(e) - \{x\}$
  $(fun\ x \to e) \downarrow (fun\ x \to e)$

$$\frac{e_1 \downarrow (fun\ x \to e) \quad e_2 \downarrow v \quad [v/x]e \downarrow v'}{(e_1\ \ e_2) \downarrow v'}$$

because:

Hair Parra 2020 ©

$$
\begin{array}{c}
(e_1 \;\; e_2) \\
\downarrow (fun\ x \to e)v \\
= [v/x]e \\
\downarrow v'
\end{array}
$$

This completes the semantics of a higher-order functional language.

16 Lecture 14: 2020-02-25

16.1 Types

Definition 42 (**Types**).

- Types classify expressions & values.
- A type is a **collection**.
- Types are used to guarantee proper computational behaviour.
  e.g. `3 + "foo"` shold prevent things like this.

Definition 43 (**Language of types**).

- **Basic types:** `int | float | bool | string | char| unit...` (all of these are disjoint)
- Later, we can have expressions with different types: **polymorphism**. → allows **type inference**.
- **New types** are built using **type constructors** (which are not a type, but things that build a type!).

1. `*` (*product constructor*): allows to build **pairs**.

```
* : product
(1729, false): int * bool

(int * bool) * string
((17, true),"bar")
```

Hair Parra 2020 ©

The previous is **isomorphic**:
`(int*bool)*string` $\simeq$ `(int*(bool*string))`

2. `list` (*list constructor*) : allows to build ordered collections of a certain type
   - `(int*bool) list` is a type.
   - `int list list` is a type
3. `->` is the *function space constructor*
   `(int -> int)` is a type ; `(int -> int) -> (bool -> bool)`
4. `|` is the *sum type*.

Definition 44 (**Typing rule**)**.**

- The following is a rule.

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

- **Matching rule**

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

Then we have,

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

$$\frac{2 : int \quad 3 : int}{2 + 3 : int}$$

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

We can then understand **pattern matching** as follows:

```
match(x,y) with (17,false)
```

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

- **Axioms**

$$\frac{}{0 : int} \quad \frac{}{1 : int} \cdots \frac{}{n : int}$$

Hair Parra 2020 ©

- Expressions also get types (see the typing rules):

$$\frac{x : int \quad y : int}{x + y : int} \qquad \frac{x : int \quad y : int}{x = y : bool}$$

  We have to give specific rules to the comparisons

$$\frac{u : string \quad v : string}{u = v : bool}$$

- `If-then-else`

$$\frac{e_1 : \tau \quad e_2 : \tau \quad b : bool}{(if \;\; b \;\; then \;\; e_1 \;\; else \; e_2) : \tau}$$

  Note! The boolean only gets evaluated at runtime.
  NOTE: At checking time, nothing is computed.At running time, things are computed.

- **List constructor**

$$\frac{e : \tau \quad \ell : \tau - list}{e :: \ell : \tau - list}$$

  NOTE: The type system does not tell you about **exceptions!**

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

- Typing rules for **functions** (most important type constructor: $\rightarrow$)
  Binding happens when functions are applied

Definition 45 (**Typing assignments**)**.**
Typing rules for **functions** (most important type constructor: $\rightarrow$)
Binding happens when functions are applied. For this, we need **variables**.
For today, assume that variables get a type by **declarations:**

$$x : \tau$$

Definition 46 (**Typing judgements**)**.**
A **type judgement** is of the form

$$\Gamma \vdash e : \tau$$

where

- Γ is a "list" of variable declarations.
- $\vdash$ (called *turnstile*) is the judgement

Hair Parra 2020 ©

- $e : \tau$ is a **type assignment**

Definition 47 (**Typing rules**)**.**

- **If-then-else rule:**

$$\frac{\Gamma \vdash e : bool \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash (if\ e\ then\ e_1\ else\ e_2) : \tau}$$

We can apply the rule above

$$\frac{x : int\ ,\ y : int \quad \vdash x = y : bool \quad x : int, y : int \vdash x : int}{(x : int, y : int \vdash if \quad x = y \quad then\ x \quad else\ y) : int}$$

- **Axiom**

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

This says: if $x : \tau$ is one of the assumptions in Γ , we can conclude $x : \tau$.

- **Rule for `let`**

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \ (let \quad x = e_1 \quad in \quad e_2) : \tau_2} \quad , x \notin FV(\Gamma)$$

Above the second term, we have a list of assumptions, and an **additional declaration:** $x : \tau_1$

Here is an example; let

$$\frac{x : int \vdash x : int \quad x : int \vdash 2 : int}{x : int \vdash x + 2 : int}$$

$$\frac{x : int, y : int \vdash x : int \quad x : int, y : int \vdash y : int}{x : int, y : int \vdash x + y : int}$$

$$\frac{}{\vdash 5 : int}$$

$$\frac{\dfrac{x : int \vdash x : int \quad x : int \vdash 2 : int}{x : int \vdash x + 2 : int} \quad \dfrac{x : int, y : int \vdash x : int \quad x : int, y : int \vdash y : int}{x : int, y : int \vdash x + y : int}}{x : int \vdash (let\ y = x + 2\ in\ x + y) : int}$$

$$\frac{\dfrac{}{\vdash 5 : int} \quad \dfrac{\dfrac{x : int \vdash x : int \quad x : int \vdash 2 : int}{x : int \vdash x + 2 : int} \quad \dfrac{x : int, y : int \vdash x : int \quad x : int, y : int \vdash y : int}{x : int, y : int \vdash x + y : int}}{x : int \vdash (let\ y = x + 2\ in\ x + y) : int}}{\vdash (let\ x = 5\ in\ (let\ y = x + 2\ inx + y)) : int}$$

Hair Parra 2020 ©

- **Arrow constructor:** $\rightarrow$

$$\tau_1 \rightarrow \tau_2$$

So how do we create these types?
`fun x -> ...`

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash (fun\ x \rightarrow e) : \tau_1 \rightarrow \tau_2}$$

The arguments you feed to a function, the type must match to that function

$$\frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}$$

(This second rule means that if you have a function with $e_1 : \tau_1 \rightarrow \tau_2$ and $e_2 : \tau_1$ then the evaluation of e_1 with argument e_2 is of type τ_2).

Example 119 (**Examples**).

- Example:

$$\frac{x : int \vdash x : int}{\vdash\ fun\ x : int \rightarrow x : int \rightarrow int}$$

Here, we could drop the type annotation in fun.

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash\ fun\ x : \tau_1 \rightarrow \tau_2}$$

- Example:

Suppose the following:

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

$$\Gamma := x : int, y : int, u : int, f : int \rightarrow int$$

Then we have:

$$\frac{\dfrac{\Gamma \vdash u : int \quad \Gamma \vdash x : int}{\Gamma \vdash u + x : int}}{\Gamma \vdash (fun\ u \rightarrow u + x) : int \rightarrow int}$$

Hair Parra 2020 ©

$$\frac{\Gamma \vdash f : int \to int \quad \Gamma \vdash y : int}{\Gamma \vdash f\ y\ : int}$$

$$\frac{y : int, f : int \to int, x : int \vdash f\ y : int \quad \Gamma \vdash 2 : int}{f : int \to int, x : int \vdash (let\ y = 2\ in\ f\ y) : int}$$

$$\vdots$$

$$\frac{}{x : int \vdash (let\ f = fun\ u \to u + x\ \ in\ let\ y = 2\ in\ f\ y)\ : int}$$

17 Lecture 15: 2020-02-27

17.1 Polymorphism

Definition 48 (**Polymorphism**)**.**
The same piece of code can have many shapes/types.

Example 120 (**Identity**)**.**
This code works uniformily across many different types.

```
fun x -> x
```

We capture this uniformily by introducing **type variables**.

Example 121 (**Identity**)**.**

```
fun x -> x
```

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

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

The best way to capture all these valid possible types is to say

$$\forall \alpha\ \ \alpha \to \alpha$$

Hair Parra 2020 ©

- The universal quantifier is supressed, implictly undestood.
- This produces
  `fun x -> x:  'a -> 'a`

Definition 49 (**Formal polymorphism typing rules**)**.**

$$\tau ::== int|bool|\ldots|\tau * \tau|\tau - list|\tau \to \tau|\alpha$$

where α is a **type variable**.

$$(\alpha \to \alpha) \to (\alpha \to \alpha)$$

The above stands for a **family of variables**.

e.g. We can **instantiate** α as int.

$$(int \to int) \to (int \to int)$$

e.g We can **instantiate** α as $\beta - list$

$$(\beta - list \to \beta - list) \to (\beta - list \to \beta - list)$$

NOTE: A **ground instance** has no type variables.

Definition 50 (**Substitution**)**.**
Notation:

$$[\tau/\alpha]\tau'$$

means: "replace occurrences of α in τ' with τ.

e.g.

$$[\beta - list/\alpha]\ \alpha \to \alpha = (\beta - list) \to (\beta - list)$$

Rules

1) $[\tau/\alpha]\alpha = \tau$
2) $[\tau/\alpha]\beta = \beta \ \ (\beta \neq \alpha)$
3) $[\tau/\alpha]int = int$

Hair Parra 2020 ©

4) $[\tau/\alpha]bool = bool$
5) $[\tau/\alpha]\tau_1 * \tau_2 = ([\tau/\alpha]\tau_1) * ([\tau/\alpha]\tau_2)$
6) $[\tau/\alpha]\tau_1 \to \tau_2 = ([\tau/\alpha]\tau_1) \to ([\tau/\alpha]\tau_2)$

The following rule captures **polymorphism**

$$\frac{\Gamma \vdash e : \tau}{[\tau'/\alpha]\Gamma \vdash e : [\tau'/\alpha]\tau}$$

Theorem 17.1 (**Fundamental Theorem**). *For every expression e, there is a **unique** type τ, possibly containing type variables such that every valid type for e is obtained by an appropriate substitution of τ. We say that τ is a (or the) **principal type** for e.*

17.2 Type Inference

Example 122 (**Type of map function**).

```
map f lst =
   match lst with
   |[] -> []
   |x::xs -> (f x) :: (map f xs)
```

How can we figure out the type of this?

- Let $f : \alpha$, $lst : \beta$
- $\beta = \gamma - list$
- What is the return type? = It is returning some kind of list!
- $\delta : \eta - list$
- $\implies x : \gamma \;\; xs : \gamma - list$
- $\implies \alpha = \gamma \to \eta$
- $(fx) : \eta$
- $\implies map : (\gamma \to \eta) \to \gamma - list \to \eta - list$
- `('a -> 'b)-> 'a list -> 'b list`

Example 123 (**PRACTICE FOR THE FINAL**).
Derive the types for

Hair Parra 2020 ©

1. append
2. reverse
3. fold-left

17.3 Type Constraints

Definition 51 (**Type Constraints Inference**).

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

- Γ is a set of variable declarations
- e is an expression
- τ is a polymorphic type
- C is a constraint.
- The expression e will have type τ is the constraints are satisfied.

$$\frac{}{\Gamma \vdash n : int / \varnothing} \quad \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau / \varnothing}$$

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

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

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

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

Functions

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

Hair Parra 2020 ©

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

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

Lists

$$\frac{\Gamma \vdash e_1 : \tau_1/C_1 \quad \Gamma \vdash e_2 : \tau_2/C_2}{\Gamma \vdash e_1 :: e_2 : \tau_2/C_1 \cup C_2 \cup \{\tau_2 = \tau_1 - list\}}$$

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

Example 124 (**Function Type Constraints**).

$$\frac{}{x : \alpha \vdash \alpha/\varnothing}$$

$$\frac{}{x : \alpha \vdash 1 : int/\varnothing}$$

$$\frac{\dfrac{}{x : \alpha \vdash \alpha/\varnothing} \quad \dfrac{}{x : \alpha \vdash 1 : int/\varnothing}}{x : \alpha \vdash x + 1 : int/\{\alpha = int\}}$$

$$\frac{\dfrac{\dfrac{}{x : \alpha \vdash \alpha/\varnothing} \quad \dfrac{}{x : \alpha \vdash 1 : int/\varnothing}}{x : \alpha \vdash x + 1 : int/\{\alpha = int\}}}{\vdash (fun\ x \to x + 1) : \alpha \to int/\{\alpha = int\}}$$

Solution: $\alpha = int$, so we get

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

Example 125 (**Append**).

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

Hair Parra 2020 ©

- $l_1 : \alpha \quad l_2 : \beta$
- From the match: $\implies \alpha = \gamma - list$ (γ-fresh)
- $x : \alpha \implies$ return type is $\gamma - list$
- $\implies l_2 : \gamma - list$
- $\therefore \gamma - list * \gamma - list \to \gamma - list$

Example 126 (**Reverse**).

```
1. let rec reverse l =
2.   let rec helper(l,acc) =
3.     match l with
4.     |[] -> acc
5.     |x::xs -> helper(xs, x::acc)
6.   in
7. helper(l,[]) (*Call with list and empty list acc*)
```

- Le't s first analyze the type of `helper`
- $(\ell, acc) : \lambda * \alpha$
- $\lambda : \beta\text{-}list$ (From match)
- $\implies x : \beta \ , \ xs : \beta\text{-}list$ (from matching)
- $\ell : \lambda \iff xs : \lambda$
- but! $xs : \beta\text{-}list \implies \ell : \beta\text{-}list$
- $acc : \alpha \iff (x :: acc) : \alpha$(From line 2 and 5)
- $x : \beta \implies (x :: acc) : \beta\text{-}list$
- $\implies acc : \beta\text{-}list$
- $\implies$ `helper`: $(\beta\text{-}list \to \beta\text{-}list) \to \beta\text{-}list$
- So `reverse`: $\beta\text{-}list$

Example 127 (**Fold-left**).
Type: $(\alpha \to \beta \to \alpha) \to \alpha \to \beta\text{-}list \to \alpha$
`fold left` f a $[x_1, x_2, \ldots, x_n] = (f \ldots (f(f\ a\ x_1)\ x_2) \ldots x_n)$

Hair Parra 2020 ©

```
1. let sec fold_left f acc l =
2.   match l with
3.   |[] -> acc            (*done, ret accumulator*)
4.   |x::xs -> fold_left f (f acc x) xs  (*apply formula along the list *)
```

1. Let $f : \phi$, $acc : \alpha$, $\ell : \lambda$
2. $\implies \lambda : \beta\text{-}list$ (From the match in line 4)
3. $\implies x : \beta$, $xs : \beta\text{-}list$ (From match in line 4)
4. $\implies (f\ acc\ x) : \tau \iff acc : \tau$ (From match in line 4 and line 1)
5. $acc : \alpha, x : \beta \implies f : \alpha \to \beta \to \tau$
6. $\implies f : \alpha \to \beta \to \alpha$ (From 1) and 4))
7. $\implies fold\_left : \Phi \to \alpha$ (From line 3)
8. **type:** $(\alpha \to \beta \to \alpha) \to \alpha \to \beta\text{-}list \to \alpha$ (From line 4 and the previous)

Example 128 (**Type for the church numeral 2**).

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

- $f : \alpha\ \ x : \beta\ \ \ f\ x : \gamma$
- $\implies \alpha = \beta \to \gamma$
- $f\ (f\ x)$ says input type for f is γ
- $\implies \gamma = \beta$
- $\therefore f : \beta \to \beta$
- Type for the whole thing:
- $(\beta \to \beta) \to \beta \to \beta$

Example 129 (**Unsolvable type**).

```
let badfun = fun f -> f f
```

Hair Parra 2020 ©

- $f : \alpha$
- $f\ f$ says $f : \alpha \to \beta$
- $\implies \alpha = \alpha \to \beta$
- This equation cannot be solved!!

17.4 Unification

Example 130 (Solving constraints).
Consider the constraints $\{\alpha = int \to \beta, \beta = \beta_1 * bool, \beta_1 = int\}$ can be solved by

- $\alpha = int \to (int * bool)$
- $\beta = int * bool \quad \beta_1 = int$

Similarly, for $\{\alpha_1 \to \alpha_2 = int \to \beta\ ,\ \beta = int \to \alpha_1\}$ can be solved by

- $\alpha_1 = int \quad \beta = int \to int = \alpha_2$

Definition 52 (**Unification**).
It is an algorithm to solve **type constraints**.

- Suppose σ is a substitution $[\tau/\alpha]$. Let $[\sigma]\tau'$: carry out σ on τ'.
- If τ_1 and τ_2 are expressions and σ is a substitution on all the type variables (so σ could look like $[\tau_1/\alpha_1, \tau_2/\alpha_2, \dots]$) such that $[\sigma]\tau_1 = [\sigma]\tau_2$ (where the equality sign means identity), we say that $\tau_1 \& \tau_2$ are **unifiable**, and σ is the **unifier** .
- **Unification algorithm**

$$\{C_1, C_2, \dots, C_n | int = int\} \implies \{C_1, C_2, \dots, C_n\}$$

$$\{C_1, C_2, \dots, C_n | bool = bool\} \implies \{C_1, C_2, \dots, C_n\}$$

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

$$\{C_1, C_2, \dots, C_n, | \tau_1\text{-}list = \tau_2\text{-}list\} \implies \{C_1, C_2, \dots, C_n, \tau_1 = \tau_2\}$$

$$\{C_1, C_2, \dots, C_n, (\tau_1 * \tau_2) = (\tau_1' * \tau_2')\} \implies \{C_1, C_2, \dots, C_n, \tau_1 = \tau_1', \tau_2 = \tau_2'\}$$

Hair Parra 2020 ©

- **Restriction:** We will not allow the constraint $\alpha = \tau$ where $\alpha \in FV(\tau)$.

Example 131 (**Not unifiable constraints**). If we allow $\alpha = int \to \alpha$, for example,this would lead to $\alpha = int \to (int \to \dots)$, a never-ending expression.
Before we introduce a constraint of the form $\alpha = \tau$, we will **check** if α occurs in τ : this is poetically called an **occurs check**.

- Consider: $\{\alpha_1 \to \alpha_2 = int \to \beta, \beta = \alpha_2 \to \alpha_2\}$
- $\implies \{\alpha_1 = int, \beta = \alpha_2, \beta = \alpha_2 \to \alpha_2\}$
- $\implies \{[\alpha_2/\beta]\alpha_1 = [\alpha_2/\beta]int\ ,\ [\alpha_2/\beta]\beta = [\alpha_2/\beta(\alpha_2 \to \alpha_2)\}$
- $\implies \alpha_1 = int, \alpha_2 = \alpha_2 \to \alpha_2$ **occurs-check fails**
- $\implies$ **not unifiable**

18 Lecture 16: 2020-03-10

18.1 Compilers

Definition 53 (**Parsing**).
Parsing is the action of converting a characters string into a structured object. Identifying different substrings separated by some criterion is called **tokenization**.

Definition 54 (**Syntax**).
The study of the grammatical structures.

- **Parse trees** represent the formation of structure.
- A **Grammar** will generate valid structures.
- In programming languages, it specifies what it means to have a valid program.

Definition 55 (**Context-Free Grammars**).
A **context-free grammar** is of the form where

- Σ (called the **alphabet**) is a set of symbols
- Σ^*: is the set of all finite sequences of symbols including the empty sequence. (NOTE: ϵ represents the *empty string*).

A **language** is a subset of Σ^*

Hair Parra 2020 ©

Example 132 (**A small alphabet**).

- Let $\Sigma = \{a, b\}$
- Let $\Sigma^* = \{\epsilon, a, b, aa, ab, ba, \dots\}$

NOTE: The alphabet is finite , but Σ^* is infinite.

Example 133 (**Example language 1**). Let L consist of all words with equally many a's and b's.

- $abba \in L$
- $ba \in L$

Definition 56 (**Context-Free Grammars (Formal definition)**).
A **context-free grammar** is a 4-tuple, $G = (V, \Sigma, R, S)$, where

- Σ or T (called the **alphabet**) is a set of symbols, also called **terminals**
- NT is a set of symbols called **non-terminals**.
- $S \in NT$ is a special symbol called the **start symbol**.
- R is a set of **rules** called **productions**, such that for $A \in NT$

$$A \to (\Sigma \cup NT)^*$$

We use the start symbol and keep generating strings until all non-terminals disappear.

Example 134 (**A small language with rules**).

- Let $\Sigma = \{a, b\}$
- Let $NT = \{S\}$
- Let

$$R = \begin{cases} 1. S \to \epsilon \\ 2. S \to SS \\ 3. S \to a\ Sb \\ 4. S \to b\ Sa \end{cases}$$

Then, we can generate, e.g. $S \to b\ Sa \to b\ aSb\ a \to ba\ ba$

Hair Parra 2020 ©

Example 135 (**Arithmetic Expressions**).

- Let $\Sigma = \{numbers, +, *, (,)\}$ (14 non-terminals)
- Let $NT = \{< EXP >\}$
- Let

$$R = \begin{cases} 1. < EXP > \rightarrow (< EXP >) | < EXP > + < EXP > \\ 2. < EXP > * < EXP > | numbers \end{cases}$$

NOTE: This grammar is **ambiguous**; we can generate the same string with two different parse trees.

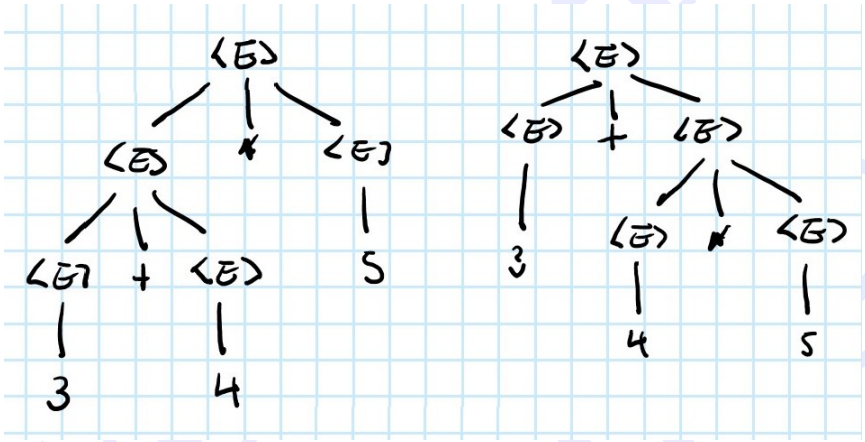

Figure 25: These two derivation trees generate the same string, indicating that the grammar is ambiguous

An **unambiguous** grammar looks like

- Let $\Sigma = \{numbers, +, *, (,)\}$ (14 non-terminals)
- Let $NT = \{< N >, < D >, < E >, < F >, < T >\}$, where $< E >$ is the start symbol
- Let

$$R = \begin{cases} 1. < E > \rightarrow < E > + < T > | < T > \\ 2. < T > \rightarrow < T > * < F > | < F > \\ 3. < F > \rightarrow < N > | (< E >) \\ 4. < N > \rightarrow < N >< D > | < D > \\ 5. < D > \rightarrow 0|1|2|3|4|5|6|7|8|9 \end{cases}$$

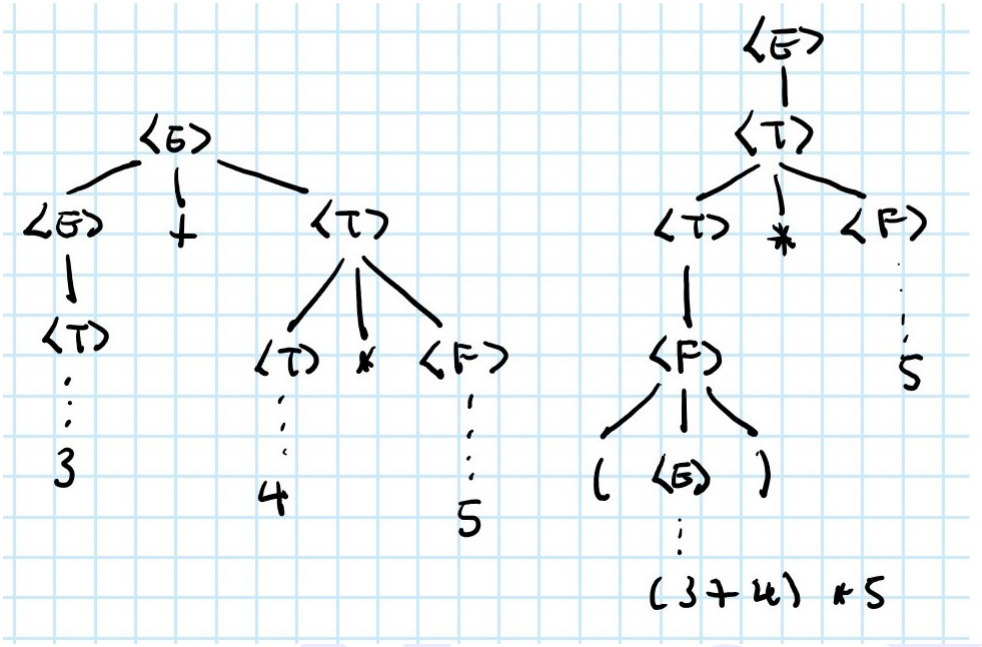

Figure 26: The grammar is designed so that every expression can be generate in a **unique way**

18.2 Parsers

Definition 57 (**Syntax Graphs (for algebraic expressions)**).

- Let $\Sigma = \{+, *, A, B, C, \ldots, Z, (,)\}$
- Let $NT = \{< EXP >, < TERM >, < PRIMARY >\}$
- Let

$$R = \begin{cases} 1. < EXP > \longrightarrow < TERM > \{+ < TERM >\} \\ 2. < TERM > \longrightarrow < P > \{* < P >\} \\ 3. < P > \longrightarrow \ A|B|C| \ldots |Z| \ (< EXP >) \end{cases}$$

- Then the graph is

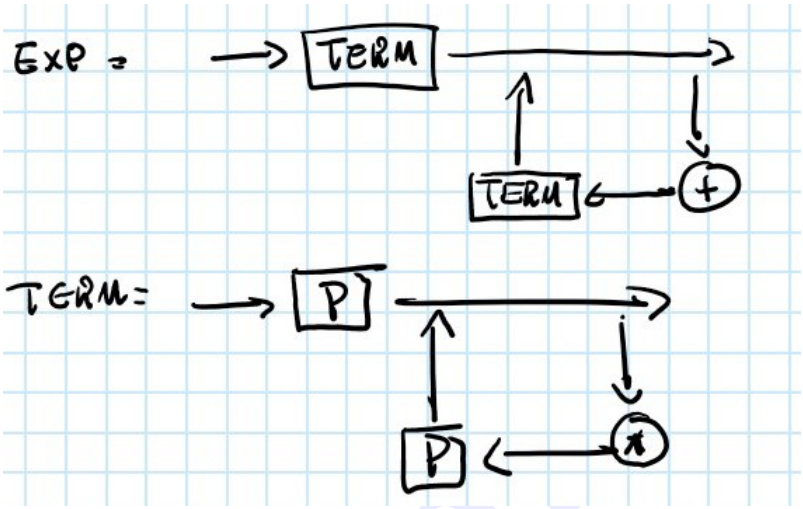

Figure 27: Diagrams for $< EXP >$ and $< TERM >$

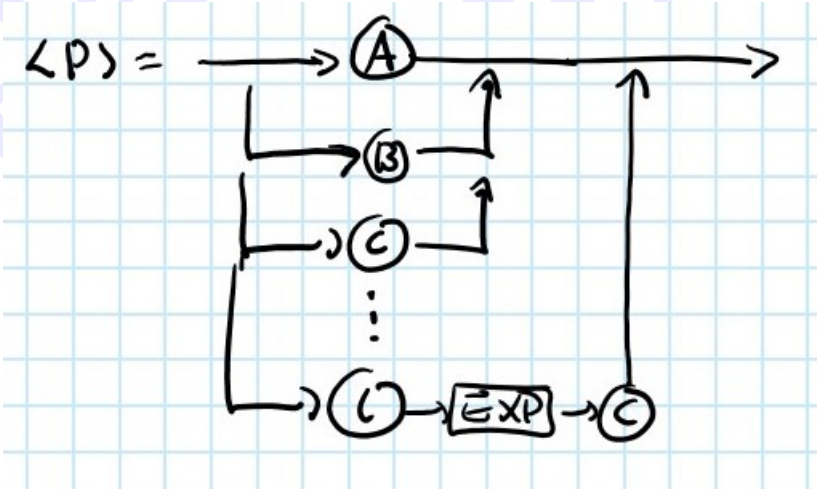

Figure 28: Diagram for $< P >$

Hair Parra ©

Definition 58 (**Expression Trees**).

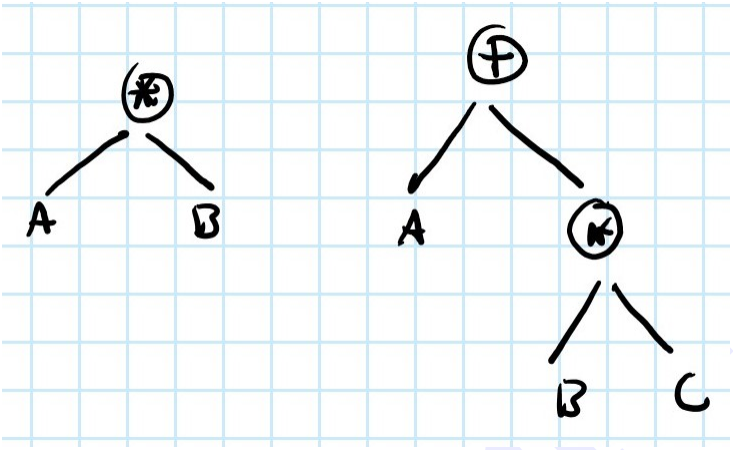

Figure 29: Expression trees

Definition 59 (**The structure of a compiler**).

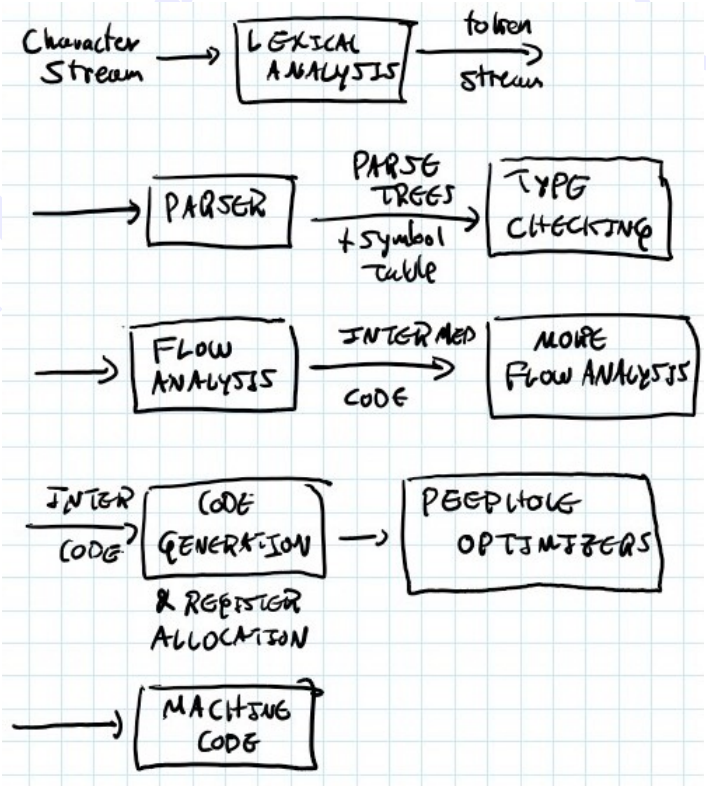

Figure 30: Compiler workflow structure

19 Lecture 16: 2020-03-12

19.1 Compiling to Machine Code

- Programs are symbolic data - **Turing**
- Early computer was made by Charles Babbage
- First **Turing-complete** programming language was maed by Ada Lorelace.

We will explore form high-level language to a lower -level language (i.e. **compilation**).
"OCaml is made for YOU"

Definition 60 (**Reading for Interpreters and Compilers**)**.**

- **Interpreters**: reads line by line.
- **Compilers**: Takes the whole program and feed it to the hardware.

We have the model

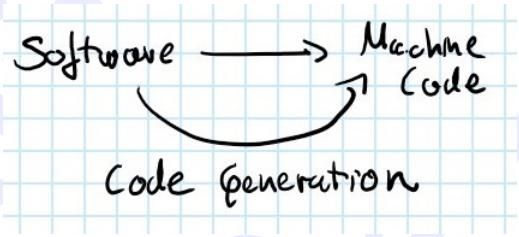

Figure 31: Example

We'll do an example on this:

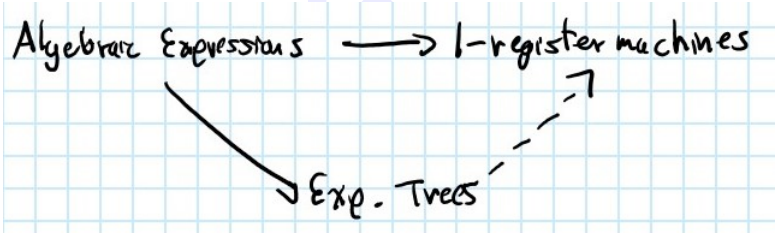

Figure 32: Example

Definition 61 (**Register Machine**)**.**
A register machine consists of a **locus of computation** + **memory**.
Let:

Hair Parra ©2020

- `X`: variable name $\rightarrow$ memory location.
- `1,2,3,...`: memory locations

Then we have the operations

1. `LOAD X`: puts the value in memory location `X` into the accumulator
2. `STORE X`: puts the value in the accumulator into location named by `X`.
3. `STORE n`: stores the value in the acc into the location memory n.
4. `ADD X` , `ADD n`
5. `MUL X` , `MUL n`

Example 136 (**Some examples**).

- `A*B+C`

$$\begin{cases} LOAD\ A & ||A \\ MUL\ B & ||A*B \\ ADD\ C & ||A*B+C \end{cases}$$

- `A*B+C*D`

$$\begin{cases} LOAD\ A \\ MUL\ B \\ STORE\ 1 \\ LOAD\ C \\ MUL\ D \\ ADD\ 1 \end{cases}$$

- `A+B*C`

$$\begin{cases} LOAD\ A \\ STORE\ 1 \\ LOAD\ B & x \\ MUL\ C \\ ADD\ 1 \end{cases}$$

Note 9.
The expression tree gives the order to compute by.

Definition 62 (**Codegen**).

Hair Parra 2020 ©

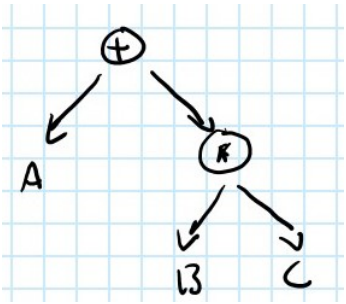

Figure 33: Expression tree for `A+B*C`

- **Codegen** will have to be recursive, but with some imperative features.
- Codegen needs contextual information (ex. need to know was doing addition at point y); this will be an extra parameter to it.
- Codegen has **2 tags**: `(context tag, tree)`
- **context tags** are:

$$\begin{cases} = \text{Starting afresh} \\ *\text{Doing a toplevel MUL} \\ +\text{Doing a toplevel ADD} \end{cases} = \begin{cases} codegen(=, \textcircled{A}) \to Load\ A \\ codegen(+, \textcircled{A}) \to Add\ A \\ codegen(*, \textcircled{A}) \to Mul\ A \end{cases}$$

Example 137 (**Codegen example**).

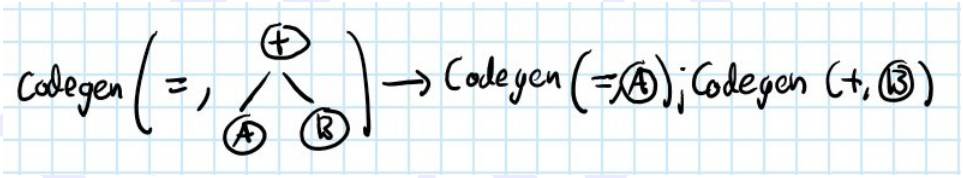

Figure 34: Generation of the small tree in Codegen

In general

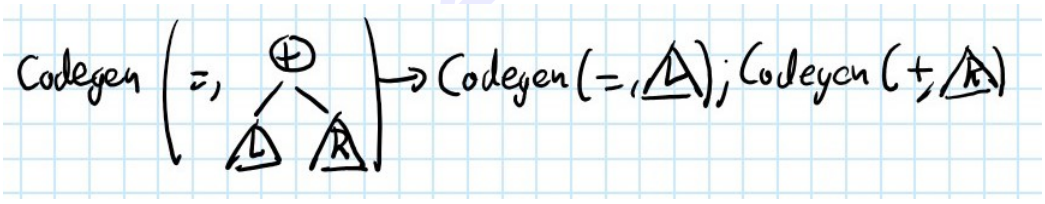

Figure 35: Generalization of previous example: the subtrees are evaluated in the same fashion

Note the following codegen parsing:

The above...

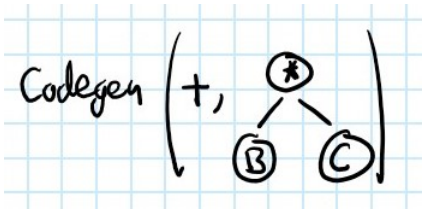

Figure 36: Intermediate step for computation

- Needs temp storage
- Maintains a **global updatable variable**.
- **temp store**: gives the location just used.

Hair Parra 2020 ©

Example 138 (**Codegen computation algorithm**).

Consider the following diagram:

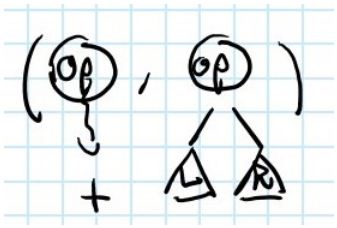

Figure 37: Intermediate step for computation

Algorithm 1 Codegen Computation Algorithm

PROCEDURE:

$tempstore := tempstore + 1$
$print("store", tempstore)$
$codegen(=, \triangle L)$
$codegen(op, \triangle R)$

if $op='+'$ **then**
  $print("ADD", tempstore)$
else if $op='*'$ **then**
  $print("MULT", tempstore)$
else
  ERROR
end if
$tempstore := tempstore - 1$

Hair Parra 2020 ©

Example 139 (**Tracing a recursive call**).
Consider the expression tree in 38

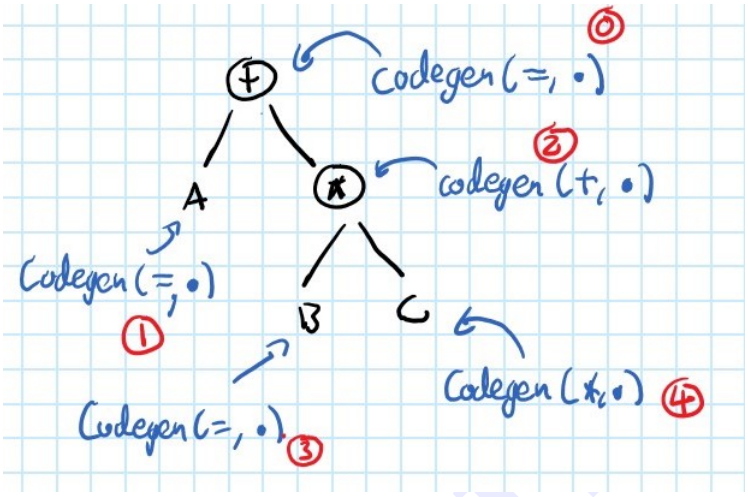

Figure 38: Tracing the recursive calls in evaluation

The procedure is:

1. `LOAD A` (load contents form location pointed by A)
2. `STORE 1` (store in register #1)
3. `LOAD B` (load contents from location pointer by B)
4. `MUL C` (multiply contents in B with contents in C)
5. `ADD 1` (load and add to content in register #1 to the previous)

Hair Parra 2020 ©

Example 140 (**A larger example**).

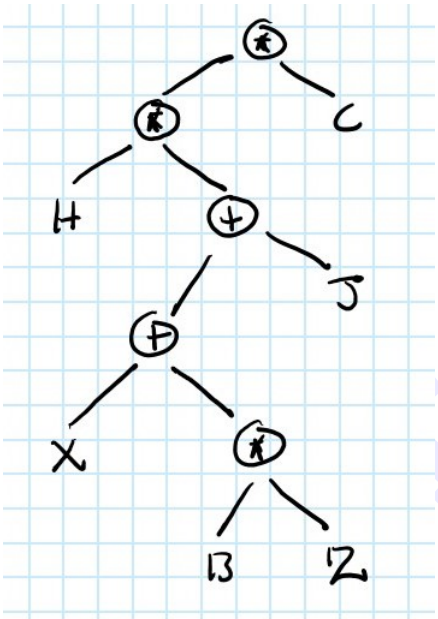

Figure 39: A rather messy tree

1. `LOAD H`
2. `STORE 1`
3. `LOAD X`
4. `STORE 2`
5. `LOAD B`
6. `MUL Z`
7. `ADD 2`
8. `ADD J`
9. `MUL 1`
10. `MUL C`

These are **10 steps, 20 memory cells!** .Instead, we can re-organize the tree for faster computation!

1. `LOAD B`

Hair Parra 2020 ©

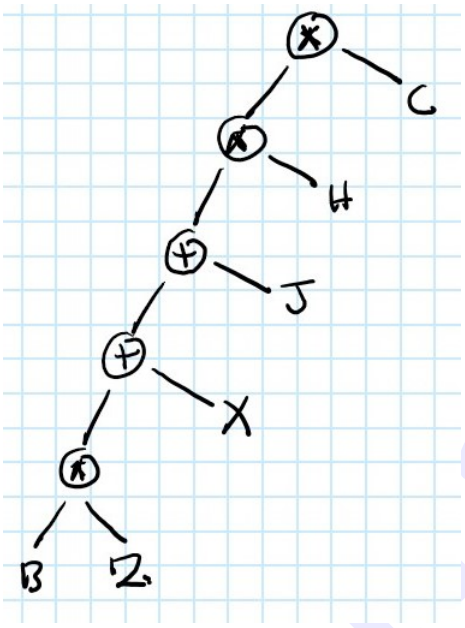

Figure 40: Re-arranged computation of the tree

2. `MUL Z`
3. `ADD X`
4. `ADD J`
5. `MUL H`
6. `MUL C`

6 steps, 0 memory cells!

This one takes

20 Lecture 16: 2019-03-25

20.1 Infinite lists

Example 141 (**Lists and circular lists**).

- Recall the familiar linked-list. which can grow **dynamically** (as opposed to an array with fixed length)

Hair Parra 2020 ©

Figure 41: A linked list

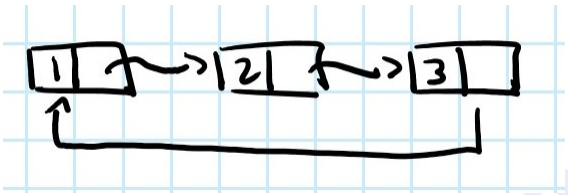

Figure 42: A circular linked list. In a sense, this represents the **infinite list** `[1;2;3;1;2;3;1;2;...]` However, it is no more than the same information repeated endlessly.

- Also consider a **circular linked list** (figure 44)
- These infinite lists can be manipulated, but perhaps don't want to print them.
- More interesting lists: primes

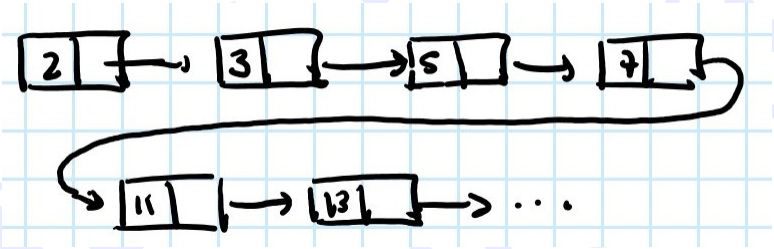

Figure 43: This list is actually infinite: cannot be represented with a circular pointer.

- Cannot put this list in memory!
- Instead, we will use **lazy evaluation**.

Hair Parra 2024

20.2 Streams

Definition 63 (**Lazy Evaluation**).

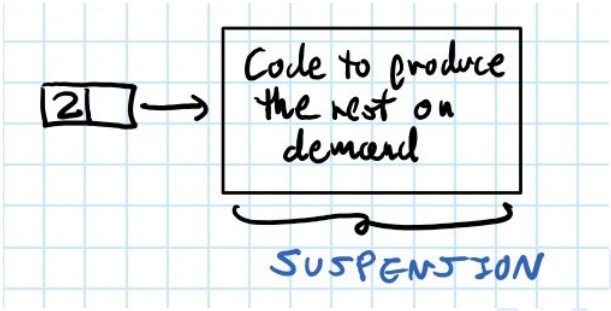

Figure 44: Representation of lazy evaluation

- The code "inside the box" is called **suspended** ,**promise** or more technically, a **thunk**(i.e. code that is not running).
- We also need a way to "wake it up!".
- We suspend a computation by wrapping it in a function.
- Wake up the code by applying the suspended code.

As an aside an interesting fact, consider the proof that there are infinitely many primes.

Theorem 20.1. *There are infinitely many primes.*

Proof. Assume there were finitely many primes $p_1, \ldots, p_k$. Consider the number

$$E = p_1 * p_2 * \cdots * p_k + 1$$

Note that E is **not a prime**, since it's bigger than all the (finitely many) primes. However, by the Fundamental theorem of arithmetic, every number has a unique factorization into primes, and therefore has to have a prime divisor. But none of the primes $p_1, \ldots, p_k$ can divide E by construction.
$\Longrightarrow$ So there must be some other primte that divides it. $\lightning$ $\therefore$ There must be infinitely many primes.

■

Definition 64 (**Streams**).

Hair Parra 2020 ©

Streams are objects that could be infinite or finite. These make use of the Lazy Evaluation principle. . Code for implementing lazy streams in OCaml. This is a shortcut that does not explicitly introduce delay and force. It is based on using thunks to delay evaluation. [1]

- The stream type below allows finite and infinite streams.
- Finite streams end with an end-of-stream marker that is written as `Eos`.
- An infinite stream does not end so it will not have an end-of-stream marker.
- **Type**

```
type 'a stream = Eos | StrCons of 'a * (unit -> 'a stream);;
```

- This is a **polymorphic type** variable. (Even streams of streams!)
- `StrCons` is a **stream constructor**: It's a pair (element of type `'a`, another stream!)
- Note that the second part of the constructor is a **wrapped** stream! `(unit -> 'a stream)`. This allows to control the stream , and only make it work on demand.

Example 142 (**Deconstructors for streams**).

```
let hdStr (s: 'a stream) : 'a =
  match s with
  | Eos -> failwith "headless stream"
  | StrCons (x,_) -> x;;

let tlStr (s : 'a stream) : 'a stream =
  match s with
  | Eos -> failwith "empty stream"
  | StrCons (x, t) -> t ();;
```

- `hdStr` returns the head of the stream.
- `tlStr` returns the tail of the tream (the rest)
- For `tlStr`, notice we return `t ()` to actually return a stream!

Example 143 (**Stream to list**).
Convert the first n elements of a stream into a list, useful to display a part of a stream.

[1] Part of the code for streams is adapted from code found at a Cornell University web site.

Hair Parra 2020 ©

```
let rec listify (s : 'a stream) (n: int) : 'a list =
  if n <= 0 then []
  else
    match s with
    | Eos -> []
    | _ -> (hdStr s) :: listify (tlStr s) (n - 1);;
```

Logic: Simply go extracting the elements of the stream and parsing them into a list, n-steps, recursively,

Example 144 (**N-th stream element**).

```
let rec nthStr (s : 'a stream) (n : int) : 'a =
  if n = 0 then hdStr s else nthStr (tlStr s) (n - 1);;
```

Example 145 (**Make a stream from a list**).

```
let from_list (l : 'a list) : 'a stream =
   List.fold_right (fun x s -> StrCons (x, fun () -> s)) l Eos;;
```

- Takes a list and produces a stream.
- Recall the definition of `fold_right` :
  `fold_right` f $[x_1, x_2, \ldots, x_n]$ b $= (f \ldots (f(f\ x_n\ b)\ x_{n-1}) \ldots x_1)$

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

- not the same as `fold_left` !

  `fold-left` f a $[x_1, x_2, \ldots, x_n] = (f \ldots (f(f\ a\ x_1)\ x_2) \ldots x_n)$
  Takes a function, an accumulator and a list.

```
let sec fold_left f acc l =
   match l with
   |[] -> acc                          (*done, ret accumulator*)
   |x::xs -> fold_left f (f acc x) xs  (*process along the list*)
```

Example of `from_list` :

```
let stramed = from_list [1;1;2;3;5;8;13] ;;
val stramed : int stream = StrCons (1, <fun>)
```

Hair Parra 2020 ©

```
listify stramed 3 ;;
- : int list = [1; 1; 2]
```

Example 146 (**Various streams**).

```
let rec ones = StrCons (1, fun () -> ones);; (*stream of ones*)
let rec nums_from n = StrCons(n, fun () -> nums_from (n + 1));; (*seq by 1 from n*)
let nats = nums_from 0;; (*Natural numbers*)
```

- The first function creates an "infinite list" of ones.
- The second one creates a sequence of natural numbers starting from n.
  NOTE: This is not a stream! It is a **stream generator**.
- The third line uses the previous function and actually generates a stream representing $\mathbb{N}$

Example

```
ones ;;
- : int stream = StrCons (1, <fun>)

listify ones 10 ;;
- : int list = [1; 1; 1; 1; 1; 1; 1; 1; 1; 1]
```

Example 147 (**Map for streams**).

```
let rec mapStr (f : 'a -> 'b) (s : 'a stream) : 'b stream =
  match s with
  | Eos -> Eos
  | _ -> StrCons (f (hdStr s), fun () -> mapStr f (tlStr s));;
```

Example 148 (**Squares**).

```
let squares = mapStr (fun n -> n * n) nats;;
val squares : int stream = StrCons (0, <fun>)
```

Example:

```
listify squares 20 ;;
- : int list =
[0; 1; 4; 9; 16; 25; 36; 49; 64; 81; 100; 121; 144; 169; 196; 225; 256; 289;
 324; 361]
```

Hair Parra 2020 ©

Example 149 (**Filter for streams**).

```
let rec filterStr (test : 'a -> bool) (s : 'a stream) : 'a stream =          '
  match s with
  | Eos -> Eos
  | StrCons (x, g) ->
      if (test x) then StrCons (x, fun () -> filterStr test (g ()))
      else filterStr test (g ());;
```

Example:

```
filterStr (fun x -> x mod 2 = 0) squares ;;
- : int stream = StrCons (0, <fun>)

listify fsq 6 ;;
- : int list = [0; 4; 16; 36; 64; 100]
```

Example 150 (**Map2 for streams**).

```
let rec map2Str (f: 'a -> 'b -> 'c)
            (s : 'a stream) (t : 'b stream) : 'c stream =
  match (s, t) with
  | (Eos, Eos) -> Eos
  | (StrCons (x, g), StrCons (y, h)) ->
      StrCons (f x y, fun () -> map2Str f (g ()) (h ()))
  | _ -> failwith "map2";;
```

Example 151 (**Fibonacci numbers**).

```
let fibStr : int stream =
  let rec fibgen (a: int) (b: int) =
    StrCons (a, fun () -> fibgen b (a + b))
  in
  fibgen 1 1;;
```

Example 152 (**Sift**).

```
let sift (p : int) : int stream -> int stream =
  filterStr (fun n -> n mod p <> 0);;
```

- Helper method to enumerate all "interesting numbers". So we identify the first number, then delete all the other numbers, etc.
- i.e. : Remove all multiples of a number from a stream.

Hair Parra 2020 ©

Example:

```
let sss = sift 2 nats ;;
val sss : int stream = StrCons (1, <fun>)

listify sss 10 ;;
- : int list = [1; 3; 5; 7; 9; 11; 13; 15; 17; 19]
```

Example 153 (**sieve of Eratosthene**).

```
let rec sieve (s : int stream) : int stream =
  match s with
  | Eos -> Eos
  | StrCons (p, g) -> (*match with head and tail *)
      StrCons (p, fun () -> sieve (sift p (g ())));;
```

- The idea is as follows:
- Take a stream, match it : if it's EOS, do nothing, return EOS
- Else, match with the head and tail, and generate a new stream: the head stays as is (first number we want to filter from the stream!) and then do the following:
- `sift p (g ())` will filter that number from the stream... but only once! So need to recurse again and call `sieve( sift p (g ()))`.
- Now need to make this back into a stream, so apply `fun() ->` to it.

Example 154 (**Primes**).

```
let primes = sieve (nums_from 2);;
```

- Just take the previous function, and apply it to 2! $\therefore$ **BOOM! We have all the primes**.

Example 155 (**Zip**).

```
let rec zip (s : 'a stream) (t : 'a stream) : 'a stream =          '
  match s with
  | Eos -> t
  | StrCons (x, g) -> StrCons (x, fun () -> zip t (g ()));;
```

- Takes two streams, and produces a new stream with items from the first stream, then the second, etc. interlaced.

Hair Parra 2020 ©

- The streams may be finite or infinite.

Example

```
let genn (n:int) =
  let rec nums = StrCons (n, fun() -> nums)
  in
  nums ;;
val genn : int -> int stream = <fun>

let fives = genn 5 ;;
val fives : int stream = StrCons (5, <fun>)

listify fives 5 ;;
- : int list = [5; 5; 5; 5; 5]

listify ones 5 ;;
- : int list = [1; 1; 1; 1; 1]

let onesfives = zip ones fives ;;
val onesfives : int stream = StrCons (1, <fun>)

listify onesfives 10 ;;
- : int list = [1; 5; 1; 5; 1; 5; 1; 5; 1; 5]
```

Example 156 (**Foo**).

Zip both the primes and the fibonacci numbers.

```
  let foo = zip fibStr primes;;
```

Example 157 (**Unzip**).

```
  let rec unzip (s : 'a stream) : 'a stream * 'a stream =                    '
    match (listify s 2) with
    | [] -> (Eos, Eos)
    | [x] -> (StrCons (x, fun () -> Eos), Eos)
    | x :: y :: _ ->
      let t = tlStr (tlStr s) in
      (StrCons (x, fun () -> fst (unzip t)), StrCons (y, fun () -> snd (unzip t)));;
```

Note 10 (**Merge**).

- **Merge:** Combine both streams into a single stream, but make some choice about which one to take first. Will put them in a certain order.
- **Demonic merging:** Throw a coin and pick

Hair Parra 2020 ©

- **Fair marge:** It could take all elements from one strean and ignore the other. Whatever you do or have, it will guarantee that everything that shows up gets serviced. **NOTE:** Can only be done with **timestops**.
- **Infinity fair merge**: Works well if both streams are infinity.
- **Angelic merge:** Like above, but will not crash if one of the streams is finite'instead it will focus its attention on the remining stream.

Example 158 (**Adding two streams together**).

```
let addStr (s1:int stream) (s2:int stream) =
   map2Str (fun x -> fun y -> x + y) s1 s2;;
```

- Recall the **map2Str** function!

Example 159 (**Partial sums**).

```
let rec partial_sums (s : int stream) =
  match s with
  | Eos -> Eos
  | StrCons (h,t) ->
      StrCons (h, fun () -> (addStr (t ()) (partial_sums s)));;
```

- third line: This uses a concept called **coinduction** and **corecursion**

Example 160 (**Alternative definition of** $\mathbb{N}$).

```
let altnats = partial_sums ones;;
let triangular = partial_sums nats;; (*nats is above*)
```

Example 161 (**Pascal's triangle**).
Pascal's triangle as an infinite stream of infinite streams (see figure 45)

```
let rec pascal =
   StrCons(ones, (fun () -> (*first stream is a bunch of ones! *)
      (mapStr partial_sums pascal)));; (*use co-recursion with the whole pascal again*)
let row7 = nthStr pascal 7;;
listify row7 10;;
```

- Try to understand this with figure 45.

Example 162 (**Infinite decimal expansion of any base**).

Hair Parra 2020 ©

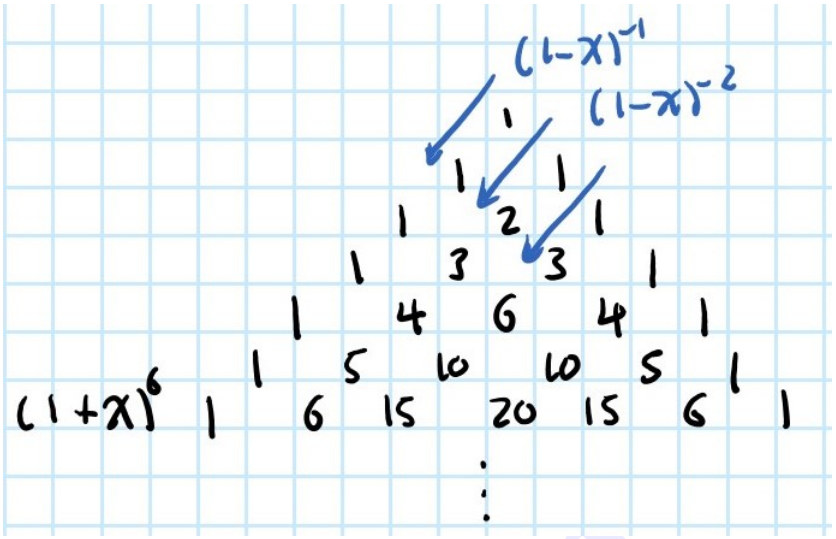

Figure 45: Unorthodox presentation of Pascal's triangle: the arrows shot the coefficients of the given expansion $(1-x)^{-n}$, where for each row being $n+1$. Each of these streams if also the partial sums of the previous stream at that point!

```
let rec expand num den radix =
  StrCons (((num * radix) / den), fun () ->
   (expand ((num * radix) mod den) den radix));;
```

Example 163 (**Twin prime search**).

```
let search_twins (s: int stream) =
  let rec helper (current: int) (str: int stream) =
    match str with
    | Eos -> Eos
    | StrCons (h, t) ->
       let next = h in
       if (next = current + 2) then
         StrCons((current,next), fun () -> helper next (t ()))
       else
         helper next (t ())
  in
  match s with
  | Eos -> Eos
  | StrCons(h,t) -> helper h (t ());;

 let twin_primes = search_twins primes;;
 listify twin_primes 10;;
```

Example 164 (**Infinite decimal expansion of any base**).

Hair Parra 2020 ©

```
let rec expand num den radix =
  StrCons (((num * radix) / den), fun () ->
   (expand ((num * radix) mod den) den radix));;
```

Example 165 (**Stream solution to the Hamming-Dijkstra problem.**).

```
let rec ordered_merge (s1: int stream) (s2: int stream) =
  match s1 with
  | Eos -> s2
  | StrCons(h1,t1) ->
     match s2 with
     | Eos -> s1
     | StrCons(h2, t2) ->
        if h1 = h2 then
          StrCons(h1, fun () -> ordered_merge (t1 ()) (t2 ()))
        else
          if h1 < h2 then
            StrCons(h1, fun () -> ordered_merge (t1 ()) s2)
          else
            StrCons(h2, fun () -> ordered_merge s1 (t2 ()));;
```

Example 166 (**Scale a stream**).

```
let scale_stream n s = mapStr (fun x -> n * x) s;;
```

Example 167 (**???**).

```
let rec ham = StrCons(1,
                  fun () ->
                  ordered_merge (scale_stream 2 ham)
                    (ordered_merge (scale_stream 3 ham) (scale_stream 5 ham)));;
```

21 Lecture 17: 2020-03-23

21.1 Subtyping

Definition 65 (**Subtyping**).
Subtyping introduces a new relation between types.

- Denoted by $\triangleleft$

Hair Para 2020 ©

- For two types τ_1 and τ_2

$$\tau_1 \triangleleft \tau_2$$

means " τ_1 is a **subtype** of τ_2" $\iff$ *any context that needs a value of type* τ_2 *will be content with a value of type* τ_1

- Another interpretation: τ_1 has every feature that τ_2 has and possibly more.
- We will be working with mxonomorphic types since polymorphic types is more subtle.

Definition 66 (**Set definition of Subtyping**)**.**

- A **type** is a collection of values sharing some common structural features.
- A **subtype** is a subset of a type.
- CAUTION! This definition is too simple-minded.

Example 168 (**Subtypes (set definition) examples**)**.**

- A dog is a subtype of the type of mammals.
- **ISA-relationship**.

Definition 67 (**Contracts**)**.**
We can think of a type as a **contract**.

- $x : \tau$ *promises* certain structural computation properties of x. This is called a **contract**.
- Ex.: $x : int$ tells us $x + 1$ makes sense.
- NOTE: A subtype may content more properties.

Example 169 (**Implicit and Explicit Coercion**)**.**

$$int \triangleleft float \implies 2 + 3,14$$

makes sense.
Easy to put into a language with run-time checking of types, but not so easy when assigning types at compile time. Ocaml has *explicit functions* to convert int to float.

Hair Parra 2020 ©

21.1.1 Record types

Example 170 (**Types, subtypes and interfaces**).

- Consider a type called **Person**, which is a **record type** with fields:
  1. `name:String`
  2. `SIN:integer`
  3. `age:integer`
- Also consider two types: **Student** and **Employee**. They have same fields as person.
- **Student** also has :
  - `studentid:integer`
  - `program:String`
- **Employee** also has:
  - `titile:String`
  - `salary:float?`

  Consider the method:

```
public static Person method(Person inputPerson){
// Some method definitions
return somePersonObject;
}
```

  This method **promises** to be of type **Person**, and also takes an input that must be of type **Person**. The method should be perfectly happy if it gets a value of type **Employee**.
- In Java, we definte the type **Person** as an **interface**, and definte classes **Student** and **Employee** that *implement* the interface **Person**.
- It is equally what happens if we define a class called **Person** and define classes **Student** and **Employee** that *extend* the class **Person**

Definition 68 (**Java Inheritance (short def.)**).
In Java, **inheritance** is when we define a class as an extension of another class. (to be discussed later)

Example 171 (**Subtype containment relation**).
Note the following relation between **Student** and **Person**: An individual **Student** record is bigger than an individual **Person** record; the collection of all student records is smaller than the collection of all person records. Note that we have

Hair Parra 2020 ©

Student $\triangleleft$ **person** and **Employee** $\triangleleft$ **Person**

but

Student $\not\triangleleft$**Employee** and **Employee** $\not\triangleleft$**Student**

However, consider an entity **TA**, who is both a **Student** and a **Employee**. (See figure 46)

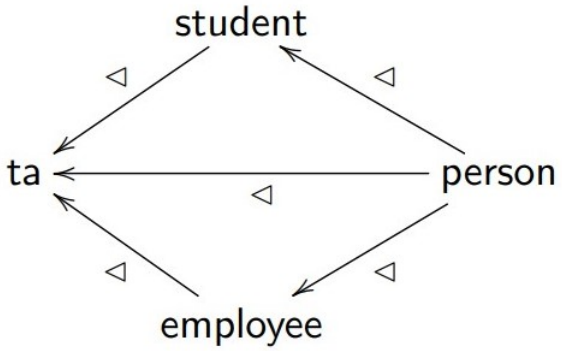

Figure 46: Subtyping diamond

NOTE! This kind of situation is banned in Java, which has inheritance mixed with pure subtyping, but not in C++.

Definition 69 (**Subtyping rules**)**.**

- **Derivation rule:** Suppose that $A \triangleleft B$, then

$$\frac{\Gamma \vdash e : A \quad A \triangleleft B}{\Gamma \vdash e : B}$$

  This says: if you can derive that e has type A in context $Gamma$ and if $A \triangleleft B$, then you can deduce that e has type B. *implies* any function(method) expecting an input of type B will be happy to receive e as input.

- **Basic rules**

$$\frac{}{T \triangleleft T} \qquad \frac{S \triangleleft R \quad R \triangleleft T}{S \triangleleft T}$$

- **Type constructors rule for tuples**

$$\frac{S_1 \triangleleft T_1 \quad S_2 \triangleleft T_2}{S_1 * S_2 \triangleleft T_1 * T_2}$$

Hair Parra 2020 ©

- **Lists**

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

NOTE: The two previous constructors *preserve* subtypying. These are called **covariant**

- **Depth subtyping (for records)**: Assumption: the field names are exactly the same.

$$\frac{S_1 \lhd T_1 \quad S_2 \lhd T_1 \quad \dots \quad S_k \lhd T_k}{\{x_1 : S_1; \dots; x_k : S_k\} \lhd \{x_1 : T_1; \dots; x_k : T_k\}}$$

NOTE: This is also a covariant rule: subtyping is preserved.

- **Derived subtypes with different fields**

$$\frac{k \leq n}{\{x_1 : T_1; \dots; x_n : T_n\} \lhd \{x_1 : T_1; \dots; x_k : T_k\}}$$

NOTE: Consider the example where subtyping between record types and the field names are *not* exactly the same.
NOTE2: If the $\leq$ is flipped, the relation gets flipped: this is called **contravariance**.

Definition 70 (**Function Types**)**.**
In OOP, functions are called *methods*, but we will call them functions here. Here, think of a **type declaration** as a **constract**.

- Declaring $f : T_1 \to T_2$ means "if f is given input of type T_1, it will deliver output of type T_2 ".

- Supposes such function has been declared, and we are given an **implementation** of type $S_1 \to S_2$, then

$$\frac{\Gamma \vdash f : S_1 \to S_2 \quad T_1 \lhd S_1 \quad S2 \lhd T_2}{\Gamma \vdash f : T_1 \to T_2}$$
$$\Longleftrightarrow$$
$$\frac{T_1 \lhd S_1 \quad S_2 \lhd T2}{(S_1 \to S_2) \lhd (T_1 \to T_2)}$$

NOTE: This is neither covariant nor contravariant, but **mixed variance**.

- The first rule is saying: if we have a function f implemented in such a way that $f : S_1 \to S_2$ but the declaration said that f should have type $T_1 \to T_2$. However, we know that $T_1 \lhd S_1$ and $T_2 \lhd S_2$, so this is fine! (Think about why).

Example 172 (**G flat**)**.**
Suppose we have a language called G flat with types `int, real` and `complex`. Assume that

$$\texttt{int} \lhd \texttt{real} \lhd \texttt{complex}$$

Hair Parra 2020 ©

- Declare a function f to tbe of type `real` $\to$ `int`.
- **Question:** Can we safely use functions of type `complex` $\to$ `real` ? NO!
- f expects input of type `real`, so since `real` $\triangleleft$ `complex`, it will accept `complex` as input. However it will output a value of type `real`, and since f has a promised output of type `int`, then this is not possible, as `int` $\triangleleft$ `real` but `real` $\not\triangleleft$ `int` (aka real number is not an intteger!).
- How about `int` $\to$ `int` ? Nope.
- How about `complex` $\to$ `int` ? YES! This works as `int` $\triangleleft$ `complex` (by transitivity).

Hair Parra 2020 ©

22 Lecture 18: 2020-04-01

22.1 Inheritance and Subtyping in Java

22.1.1 Inheritance

One of the fundamental advanced in the **object-oriented paradigm** is the abiity to *reuse* code. Often, you will find yourself coding a small variation of something youhad coded before. The new code ia new class, but looks like an older class except for a couple of methods. It is to handle situations like this that we have **inheritance**.

Example 173 (**Basic Inheritance example**).
Consider the following basic class to store an integer with some basic methods:

```
class myInt {
   // Instance variable
   private int n;

   // Constructor
   public myInt(int n){
      this.n = n;
   }

   // Instance methods
   public int getval(){
      return n;
   }

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

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

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

Now imagine that we decide to have "integers" made out of complex numbers. These numbers are called **gaussian integers** used by the great mathematician Gauss for his work in number theory. We might want to extend out ordinary integers to deal with these gaussian integers.

Hair Parra 2020 ©

- The keyword `extends` in the class declaration tells the system that you want all the code that worked for class `myInt` to be present in the class `gaussInt`.
- We say that `gaussInt` **inherits** from `myInt`.
- We also say that `myInt` is the **superclass** and that `gaussInt` is the **subclass**.

```
class gaussInt extends myInt {

   // Instance variable
   private int m; // Represents the imaginary part

   /* We do not need the real part that is already present
      because we have inherited all the data and methods
      of myInt. Thus the private int n is also present
      in every instance of a gaussInt. */

   // Constructor
   public gaussInt(int x, int y){
      super(x); // Special keyword to call the superclass constructor
      this.m = y;
   }

   // Instance methods
   public void show(){ //This method is overriden from the superclass
      System.out.println(
      "realpart is: " + this.getval() + " imagpart is: " + m);
   }

   public int realpart(){
      return getval(); // inherited from myInt
   }

   public int imagpart(){
      return this.m;
   }

   /*The method getval is defined in the superclass.
     It is not defined here, but it is inherited by
     this class so we can use it. */

   // This is an overloaded method
   public gaussInt add(gaussInt z){
      int newrealpart = z.realpart() + this.realpart();
      int newimagpart = z.imagpart() + this.imagpart();
      return new gaussInt(newrealpart, newimagpart);
   }
```

Hair Parra 2020 ©

```
   public static void main(String[] args){
      gaussInt kreimhilde = new gaussInt(3,4);
      kreimhilde.show(); // Will use the overriden method
      kreimhilde.increment(2); // Will use method from myInt
      kreimhilde.show();
   }
} // class gaussInt
```

- The word `super` invokes the superclass constructor.
- You can think that an instance of `gaussInt` *contains* an instance of myInt.
- The method show is **overriden**, i.e. , contains a new definition for the same method name.
- The method `add` is **overloaded**: this means that the arguments expected are different, and the method behaves similarly in principle but differnt in accordance to its arguments.
- How does the system knows which one to use? It looks at the **types** of the actual arguments and decides which to use. So if you want to add an ordinary `myInt` to a `gaussInt`, then the `add` method of the superclass is used.

22.1.2 Subtyping and Interfaces

One of the features of object-oriented languages is a sophisticated type system that offers a possibility called **subtype polymorphism**.
NOTE: **subtyping** $\neq$ **inheritance** !

Definition 71 (**Subtype polymorphism**)**.**

- A type system classifies values into collections that reflect some structural or computational similarity.
- In Java we have **subtype polymorphism**, which is based on the idea that there may be a relation between types called the **subtyping relation**.
- Recall: Type A is a **subtype** of type B if *whenever* the context requires an element of type B it can accept an element of type A. Denoted $A \triangleleft B$
- **NOTE1:** In Java, **subtyping** occurs automatically when you have **inheritance**, *this does not mean that subtyping and inheritance are the same thing.*
- **NOTE2:** You can also have instances of subtyping without any inheritance, this happens when you **implement** an **interface**.

Hair Parra 2020 ©

Example 174 (**Java subtyiping and inheritance**).

- Some build-in instances: $int \triangleleft float$
- User-defined: If we declare a class B to be an **extension** of class A, we will have:
  1. B inherits all the methods and attributes of A
  2. $B \triangleleft A$
- e.g. Suppose we have the method `foo` below, you can call it passing B as an argument.
- If we extend class B with class C , then we have

$$C \triangleleft B \triangleleft A$$

- Now, if extend A with another class D, then we will have $D \triangleleft A$, but there will be no subtyping relation between B and D or C and D.
- A method expecting an object of type A will accept object of type B, C or D.
- **Code illustration:**

```
class A {
   // constructor
   public A(){
   }

   // some class definitions
}

class B extends A {
   // constructor
   public B() {
      super(); // call class A constructor
   }

   // some class definitions
}

class C extends B {
   // constructor
   public C() {
      super(); // calls class B constructor
   }

   // some class deinitions
}
```

Hair Parra 2020 ©

```
class D extends A {
   // constructor
   public D() {
      super(); // call class A constructor
   }
}

class test {

   public static void foo(A objA){
      // some method definitions
   }

   public static void foo2(B objB){
      // some method definitions
   }

   public static void foo3(D obj D){
      // some method definitions
   }

   public static void main(String[] args){
      objA = new A();
      objB = new B(); // B inherits from A
      objC = new C(); // C inherits from B, and also from A
      objD = new D(); // D inherits from A only

      foo(objA); // legal
      foo(objB); // legal: B is a subtype of A
      foo(objC); // legal: C is a subtype of A by transitivity
      foo2(objA); // illegal, A is NOT a subtype of B!
      foo2(objB); // legal
      foo2(objC); // legal, C is a subtype of B
      foo2(objD); // illegal, D is NOT a subtype of B!
      foo3(objA); // illegal, A is NOT a subtype of D
      foo3(objB); // illegal
      foo3(objC); // illegal
      foo3(objD); // illegal
   }
}
```

Hair Parra 2020 ©

22.1.3 Interfaces

Definition 72 (**Java Interface**).
An **interface** is a declaration of collection of method names - *without any method bodies* -

and perhaps some constants. They describe a (sub)set of methods that a class might have.

- There is no code in an interface, hence there is nothing to inherit.
- A concrete class may be declared to **implement** an interface. This is really a subtyping declaration!
- When we say that a class P `implements` interface I, we have that $P \triangleleft I$. However, there is no inheritance!
- However, P must have a method of the same name and type as every method name in the interface I.

NOTE: Java interfaces allow **multiple subtyping**, not **multiple inheritance**, as opposed to C++, which really has.

Example 175 (**Another inheritance example**).
This part is a straight copy-paste from Prakash's notes, available at `https://www.cs.mcgill.ca/~prakash/Courses/302/Notes/sub_and_inh_java.pdf`. See footnote. [2]

22.1.4 Subtypes and Typechecking in Java

Definition 73 (**Rules for Method up and Type Checking**).
There are two phases:

1. **Compile time:** This is when **type checking** is done.
2. **Run time:** This is when **method lookup** happens.

[2]Here is an example of the use of inheritance. Imagine that you have written code to draw the graph of a mathematical function. You want this to be abstracted on the function. You do not want to write code to plot a graph of the *sine* function and another different − but almost identical − piece of code to plot a graph of the *exp* function. You want the function - a piece of code − to be a *parameter*. We can do this with objects because objects can be passed around like any other piece of data but yet they carry code. What kind of object is a mathematical function? It expects a `double` argument and returns a `double` result. There might be other methods associated with function objects but the plot method does not care. It only cares that there is a method − called, say, `y` − such that `f.y(<input>)` returns a double. So instead of having to know all about the details of the class of mathematical functions we just define an interface plottable with the one method `y` in it with the appropriate type. When we define our mathematical function objects we can make them as complicated as we please as long as we declare that they implement `plottable` and ensure that they really do have the method `y`. If we have another type of object − say `fin-data` − for financial data we would expect to define a totally different class with no relation to mathematical function objects. However, fin-data could also have a method y and be declared to implement plottable. Then our plot method works on totally unrelated classes. We have achieves generality for our plotting method through subtype polymorphism in a very general situation. Of course, we could have done this with inheritance too but it would be silly.

Hair Parra 2020 ©

NOTE: Compile time is *before* run time.

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

Example 176 (**gaussInt**).
Consider the following code:

```
class myInt {
   // Instance variable
   private int n;

   // Constructor
   public myInt(int n){
      this.n = n;
   }

   // Instance methods
   public int getval(){
      return n;
   }

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

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

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

Hair Parra 2020 ©

```
class gaussInt extends myInt {

   // Instance variable
   private int m; // Represents the imaginary part

   // Constructor
   public gaussInt(int x, int y){
      super(x); // Special keyword to call the superclass constructor
      this.m = y;
   }

   // Instance methods
   public void show(){ //This method is overriden from the superclass
      System.out.println(
      "realpart is: " + this.getval() + " imagpart is: " + m);
   }

   public int realpart(){
      return getval(); // inherited from myInt
   }

   public int imagpart(){
      return this.m;
   }

   // This is an overloaded method
   public gaussInt add(gaussInt z){
      int newrealpart = z.realpart() + this.realpart();
      int newimagpart = z.imagpart() + this.imagpart();
      return new gaussInt(newrealpart, newimagpart);
   }

     public static void main(String[] args){

        System.out.println("Kreimhilde");
        gaussInt kreimhilde = new gaussInt(3,4);
        kreimhilde.show(); // Will use the overriden method
        kreimhilde.increment(2); // Will use method from myInt
        kreimhilde.show();


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

Hair Parra 2020 ©

```
        // different types

        System.out.println("the value of z is: " );
        z.show();
        System.out.println("the value of b is: " );
        b.show();
        // which method will be used?
        // Even if the object z references b with a different type,
        // the method show() from z will be used!

        myInt d = b.add(b); // this does type check
        System.out.print("the value of d is: ");
        d.show();
        // w = z.add(b); // will not type check
        // w = b.add(z); // will not type check
        w = ( (gaussInt) b).add(z); // this does typecheck
        System.out.println("the value of w is: " ) ;
        w.show(); // no value!
        myInt c = z.add(a); // will this typecheck? -> Yes! gets casted into myInt
        System.out.print("the value of c is: " );
        c.show(); // this is now a myInt

    }

} // class gaussInt
```

What can we say about the variables:

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

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

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

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

Hair Parra 2020 ©

- z is declared to be of type `gaussInt`.
- It passes the type checker as there is a `show` method defined in the `gaussInt` class.
- At run time it uses the `show` method of `gaussInt`.

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

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

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

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

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

> 6
```

- b is declared to be of type `myInt`.
- The type checker checks to see whethere there is an `add` method in the `myInt` class.
- YES! It takes a `myInt` object and returns a `myInt` object.
- At run time $b's$ actual type is `gaussInt` the run-time system checks to see if there is an `add` method in the `gaussInt` class which matches the type that it was told by the type-checker. There are two methods: one that takes a `myInt` and returns a `myInt` (inherited from the `myInt` class). The other takes a `gaussInt` and returns a `gaussInt`; this is the method explicitly defined in the `gaussInt` class.
- However, this method does not match what the type-checker told the run-time system to expect.
- WHICH `add` METHOD TO USE?
- *"when there is overloading, it is resolved by typechecking"* $\implies$ the method which takes an object of t

Hair Parra 2020 ©

- Thus, `b.add(b)` returns a `myInt` object, and therefore NOW, the actualy type of `d` is `myInt`.

```
// w = z.add(b); // (i) will not type check
// w = b.add(z); // (ii) will not type check
```

Tese will not typecheck

1. (i)
   - z is declared to be of the type `gaussInt`.
   - There are two methods in the `gaussInt` class, the one that takes a `myInt` object and returns a `myInt` object is used.
   - Why? Once again overloading is resolved by typechecking; since b is declared to be a `myInt` object it will pick the `add` method that it inherited.
   - z is a `gaussInt` which is a subtype of `myInt` and hence is added to b and returns a `myInt`. w is declared to be a `gaussInt`. Since `myInt` is not a subtype of `gaussInt` the assignment statement will not accept this for the right hand side, and hence would cause an error.
2. (ii)
   - b is decalred to be of type `myInt`.
   - The type checker checks if there is an `add` method in the `myInt` class there is one which expects a `myInt` object and returns a `myInt` object .
   - z is a `gaussInt` and since `gaussInt` is a subtype of `myInt`, b is added to z to produce a `myInt` object.
   - w is declared to be a `gaussInt`, so the assignment fails since `myInt` is not a subtype of `gaussInt`

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

- This will typecheck since w is a `gaussInt` and the right side has a `gaussInt` too.
- It now has to choose between two possible `add` methods.
- To resolve the overloaidn it uses declared types; z has declared type `gaussInt`.
- Thus when it resolved the overloading of the `add` method it figures out to use the `gaussInt` to `gaussInt` version.

Hair Parra 2020 ©

23 Copyright

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.

Hair Parra 2020 ©

References

[1] Prakash Panangaden "COMP 302: Programming Languages and Paradigms." McGill University Winter 2020

Hair Parra 2020 ©