COMP3031: Functional Programming with ML

Qiong Luo

HKUST

Functional Programming (FP)

- A program implements a mapping from input values to output values.
- In imperative programming, this mapping is achieved indirectly by commands that read inputs, manipulate them, and write outputs.
- In FP, it is achieved directly: a program in FP is a function.
- Instead of primitive actions, in FP we have primitive functions. Instead of control constructs, in FP we have function composition rules.
- FP has been used as: (a) a convenient setting for studying concepts such as values and types; (b) a technique for language description; (c) a programming style in its own right (focus of this course).

Pure FP

- Basic principle:

  The value of an expression depends only on the values of its subexpressions, if any. Example: the value of $f(e_1, ..., e_n)$ depends only on that of e_1,..., e_n.

- Side effect:

  A function f has side effects if f makes changes to some variables and these changes persist after f has returned.

- The basic principle of pure FP rules out side effects. Why?

Features of FP

- FP is simple because of its following feature:

Implicit storage management.

Storage is allocated as necessary by built-in operators on data. Storage that becomes inaccessible is automatically deallocated.

- FP is powerful because of its following feature:

Functions are first-order values.

Functions have the same status as any other values. A function can be the value of an expression, it can be passed as an argument, and it can be put in a data structure.

FP Languages

- Begins with LISP (LISt Processing, McCarthy 58).
- Lisp family: Lisp, MacLisp, Scheme, CommonLisp,...
- Other FP languages: ML, Haskel, Miranda, ...
- We study ML (more specifically, SML) because it is small.

SML (Standard Meta Language)

SML (/usr/local/sml/bin/sml) supports:

- *Static Scope*: All identifier references resolved prior to execution.
- *Strong Typing*: Every expression has a type which can be determined prior to execution. (C++ is not. e.g. virtual function)
- *Polymorphism*: functions, data types (similar to template in C++).
- *Type-Safe Exception Mechanism*: to handle unusual situations arising at run-time. e.g. division-by-zero.
- *Abstract Data Types*: type = data + functions (as in OOP).
- *Modules*: an ML program = a set of interdependent modules glued together using *functors*.

An SML Demo

```
Standard ML of New Jersey v110.54 ...
- "Hello World!";     (* user input *)
val it = "Hello World!" : string
                      (* response from SML *)
- fun sq(x)=x*x;
val sq = fn : int -> int
- a;
stdIn:20.1 Error: unbound variable or constructor: a
- val a=3;
val a = 3 : int
- sq(a);
val it = 9 : int
-
```

5 Basic Types, 3 Composite Types

TYPE	SYMBOL	EXAMPLE	OPERATIONS/TYPES
unit	()	()	—
boolean	bool	true, false	not, andalso, orelse
integer	int	2, 0, 87	~, +, -, *, div, mod
real	real	1.3, 3E2	~, +, -, *, /
string	string	"hello"	^
tuple	(...)	(1, "yes", 2.5)	int*string*real
list	[...]	[3, 8, 1, 9]	int list
record	{...}	{ID="007", age=51}	{ID:string,age:int}

Basic Types

- **unit** is similar to **void** in C. It is used
  - whenever an expression has no value.
  - when a function is to have no arguments.
- The boolean operators **andalso** and **orelse** perform short-circuit evaluations: i.e.
  E_1 **andalso** E_2 $\Rightarrow$ will NOT evaluate E_2 if E_1 is **false**.
  E_1 **orelse** E_2 $\Rightarrow$ will NOT evaluate E_2 if E_1 is **true**.
- Negative **int** or **real** values are denoted using the unary operator $\sim$ instead of the usual minus sign.
- Integer division uses **div** and **mod**, and real number division uses /.
  NO implicit coercion!

Example: int/real

```
- ();
val it = () : unit

- 5 + 13;
val it = 18 : int

- ~5 + 13;
val it = 8 : int

- floor(123.6);
val it = 123 : int

- floor(~123.6);
val it = ~124 : int
```

Example: Type Checking in int/real

```
- 5/6;
stdIn:10.1-10.4 Error: operator and operand don't agree
  operator domain: real * real
  operand:         int * int
  in expression: 5 / 6

- real(5)/6;
stdIn:1.1-1.10 Error: operator and operand don't agree
  operator domain: real * real
  operand:         real * int
  in expression: real 5 / 6


- real(5)/real(6);
val it = 0.833333333333 : real
```

Example: String

```
- "Hong"^" "^"Kong";
val it = "Hong Kong" : string

- size "Hong Kong";
val it = 9 : int

- size "Hong"^" "^"Kong";
stdIn:69.1-69.23 Error:
  operator and operand don't agree [tycon mismatch]
  operator domain: string * string
  operand:         int * string
  in expression:   size "Hong" ^ " "

- size("Hong"^" "^"Kong");
val it = 9 : int
```

Example: Boolean Expression

- "if <bool-exp> then <then-exp> else <else-exp>" always come together; and its value is that of <then-exp> if <bool-exp> is true, otherwise that of <else-exp>.
- <then-exp> and <else-exp> must match in their types.

```
- if 2=3 then "don't worry" else "be happy";
val it = "be happy" : string
- if "don't worry"="be happy" then 1 else 2;
val it = 2 : int
- if 2=3 then "don't worry" else 4;
stdIn:16.1-16.33 Error: types of if branches do not agree
  then branch: string
  else branch: int
  in expression:
    if 2 = 3 then "don't worry" else 4
```

Composite Type: Tuple

- n-tuple: $(e_1, e_2, \ldots, e_n)$. The n items may be of mixed types.
- 2 n-tuples are equal if their corresponding components are equal.
- Items in a tuple are ordered, and "#k" selects the kth item.

```
- (4, true, "cat");
 val it = (4,true,"cat") : int * bool * string
 - (if 3=8 then "X" else "Y", 9.5/0.5, 5 div 2);
 val it = ("Y",19.0,2) : string * real * int
 - (14 mod 3, not false) = (1+1, true);
 val it = true : bool

 - #2("for", "your", "info");
 val it = "your" : string
```

List

- Empty list: **nil** or [ ];
- **nil** : 'a list ⇒ a polymorphic object.
- $[e_1, e_2, \ldots, e_n]$ is an abbreviation for $e_1{::}e_2{::}\ldots{::}e_n{::}$**nil**.
- :: is the list *constructor* pronounced as "cons".
- :: is an infix operator which is right associative.
- <new-list> = <item>::<list>.

```
1::2::3::nil = 1::(2::(3::nil))
             = 1::(2::[3])
             = 1::[2,3]
             = [1,2,3]
```

- Equality on 2 lists is item-by-item.

List Operators

- cons operator: :: : ‘a item $*$ ‘a list $\rightarrow$ ‘a list
- head operator: $hd()$: ‘a list $\rightarrow$ ‘a item
- tail operator: $tl()$: ‘a list $\rightarrow$ ‘a list
- append operator: @ : ‘a list $*$ ‘a list $\rightarrow$ ‘a list

```
- hd([1,2,3,4]);
val it = 1 : int

- tl([1,2,3,4]);
val it = [2,3,4] : int list

- hd([1,2,3,4])::tl([1,2,3,4]);
val it = [1,2,3,4] : int list

- [5,6]@tl([1,2,3,4]);
val it = [5,6,2,3,4] : int list
```

Record

- similar to **struct** in C.
- Syntax: { $label_1 = E_1$, $label_2 = E_2$, ... }
- Order does NOT matter since the fields are labelled.
- Tuples are actually short-hands for records.
  $(E_1, E_2, E_3) = \{ 1{=}E_1, 2{=}E_2, 3{=}E_3 \}$

```
- {name="bird", age=5, dead=true};
val it =  {age=5,dead=true,name="bird"}
        : {age:int, dead:bool, name:string}
- {name="bird", age=5, dead=true}
  = {age=5, dead=true,name="bird"};
val it = true : bool
```

Identifiers

BNF (Baccus-Naur Form) for alphanumeric identifiers:

```
<Id>            ::= <First_Char><Other_Chars>
<First_Char>    ::= [A-Z]|[a-z]|'
<Other_Chars>   ::= <empty>|<Other_Char><Other_Chars>
<Other_Char>    ::= [A-Z]|[a-z]|[0-9]|['_]
```

BNF for symbolic identifiers:

```
<Id>           ::= <S_Char>|<S_Char><Id>
<S_Char>       ::= [+-/*<>=!@#%^'~\$?:]
```

- '<Other_Char> are *alpha* variables ONLY used for data types.
- Symbolic identifiers should be used for user-defined operators.

Identifiers: Value Binding

Syntax: **val** < *identifier* > **=** < *expression* >**;**

```
- val a_df = 3+2;  (* similar to const int a_df = 3+2; in C++
val a_df = 5 : int

- val a'a = "Albert"^"Einstein";
val a'a = "AlbertEinstein" : string

- val a1b2 = 2;
val a1b2 = 2 : int

- val +++$$$ = 9*3; (* may hold integral value *)
val +++$$$ = 27 : int

- +++$$$ + +++$$$;  (* Though you don't want to do that *)
val it = 54 : int
```

Declaration: let Statement

let

$$\textbf{val} <\text{1st-identifier}> = < E_1 >;$$
$$\textbf{val} <\text{2nd-identifier}> = < E_2 >;$$
$$\ldots$$

in

$$<\text{expression}>$$

end

- The semicolons at the end of each **val** statements is optional.
- similar to Declaration of local variables in C++

Let Example

```
- let
    val x = 3
    val y = 5
  in
    x*x + 3*y
  end;
 val it = 24 : int
```

let: val Example

```
- val z =
      let
        val x = 3
        val y = 5
      in
        x*x + 3*y
      end;
val z = 24 : int
```

- The statement written in one single line:

```
val z = let val x = 3  val y = 5 in x*x + 3*y end;
```

- To avoid too many **val** statements in the **let**-part, one may use tuples to group all identifiers as follows:

```
val z = let val (x, y) = (3, 5) in x*x + 3*y end;
```

Nested let Example

```
- let val x = 3.0  val y = 5.0 in
      let val a = x+y  val b = x-y in
          let val f = a*b*x  val g = a/b/y  in  f/g  end
      end
end;
```

What is the output? val it $= 60.0$: real

```
f/g =>  (a*b*x)/(a/b/y) =>
((x+y)*(x-y)*x)/((x+y)/(x-y)/y) =>
8.0*(-2.0)*3.0 / 8.0/-2.0/5.0 =>  -48.0 / -0.8
```

Pattern Matching

- Pattern matching with tuples

```
- val (left, right) = ("Einstein", 4);
val left = "Einstein" : string
val right = 4 : int
```

- Pattern matching with lists

```
- val x::y = [5,6,7,8];   (* [5,6,7,8] = 5::[6,7,8] *)
val x = 5 : int
val y = [6,7,8] : int list
```

- Pattern matching with records

```
- val {flag=y,count=x} = {count=2,flag=true};
val x = 2 : int
val y = true : bool
```

Pattern Matching: Wildcard Pattern

The wildcard pattern "_" (underscore symbol) may be used for terms that you don't care in pattern matching.

```
-  val (left,_) = ("Einstein", 4);
val left = "Einstein" : string

- val _::a = [1,2,3];
val a = [2,3] : int list

- val x::_::z = [[1,2],[3,4],[7,9],[0,0]];
val x = [1,2] : int list
val z = [[7,9],[0,0]] : int list list
```

Pattern Matching: Bug

Identifiers cannot duplicate in different parts of one pattern.

```
- val (x, x::y) = (3, [3,4,5]);
stdIn:1.1-287.4 Error: duplicate variable in pattern(s): x

- val (x, x) = (3,3);
stdIn:1.1-279.7 Error: duplicate variable in pattern(s): x
```

Functions: It is "fun"

- Syntax: **fun** <identifier> (<parameter-list>) = <expression>;
- Parameter passing method: Call-By-Value.

```
- fun square(x) = x*x;
val square = fn : int -> int

 - fun square x = x*x; (* parentheses are optional *)
val square = fn : int -> int
- square 4;
val it = 16 : int

 - fun first (x,y) = x;
val first = fn : 'a * 'b -> 'a
first (3, "man") => val it = 3 : int
first ("man",3) => val it = "man" : string
```

Type of Functions

Each identifier, variable or function, has a type.
Function : <domain type> → <range type>

- Argument type may be explicitly specified with :< *type* >. e.g. A function whose input is a **real** number and which returns a **real** number:

  ```
  - fun f_square(x: real) = x*x;
  val f_square = fn : real -> real
  - fun f_square(x):real = x*x;     (* Another way *)
  ```

- Types can be polymorphic - a dynamic type determined only at run time:

  ```
  first (3, "man") => val it = 3 : int
  first ("man",3) => val it = "man" : string
  ```

Scope

In functions, identifiers with the same name are resolved using the static lexical scope rule.

```
fun weird(x: real) =
    let val x = x*x
        val x = x*x
    in  x*x*x end;

- weird 2.0;
```

What is the result? 4096.0

```
x*x*x =>  (x*x)*(x*x)*(x*x) =>
((x*x)*(x*x))*((x*x)*(x*x))*((x*x)*(x*x)) =>
((2.0*2.0)*(2.0*2.0))*((2.0*2.0)*(2.0*2.0))*((2.0*2.0)*(2.0*2.
```

More Complex Functions

- Defined with boolean expressions.

```
- fun greater(x,y) = if x > y then x else y;

- fun factorial x = if x = 0
=     then 1   (* Initial ''='' is continuation symbol *)
=     else x*factorial(x-1);
```

- Defined by enumerating **ALL** cases with pattern matching ($\Rightarrow$ more readable).

```
- fun factorial 0 = 1
  |   factorial x = x * factorial(x-1);
```

Functions: Bug

When functions are defined by case analysis, SML issues a warning or an error if

- Not all cases are covered.

```
- fun myhead(head::tail) = head;
stdIn:266.1-266.30 Warning: match nonexhaustive
          head :: tail => ...
val myhead = fn : 'a list -> 'a
```

- A case is redundant because of earlier cases.

```
- fun nonsense(_) = 3 | nonsense(0) = 5;
stdIn:275.1-275.47 Error: match redundant
          _ => ...
    -->   0 => ...
```

Deducing Types

- ML is strongly typed - every expression has a type, and type checking is always done at "compile" time.
- SML interpreter will infer the type of an expression automatically.
- Some simple rules are:
  - The types of arithmetic operators are built-in, and no coercion is done.
  - In a conditional expression, the expression itself and the sub-expressions follow the then and else must be of the same type.
  - The return type of a function is the same as the type of the expression that defines the function.

An Example

```
fun comb(n,m) =
      if m=0 orelse m=n then 1
      else comb(n-1,m) + comb(n-1,m-1);
```

```
m=0 => m must be an int
m=n => n must be an int
the 'then' part is an int => the 'if' expression is an int

val comb = fn : int * int -> int
```

Higher-Order Functions (I)

Functions taking functions as arguments:

```
- fun square x = x*x; fun twice x = 2*x;

- fun apply5 f = f 5;
 val apply5 = fn : (int -> 'a) -> 'a
- apply5 square;   val it = 25 : int

- fun apply f x = f(twice(x));
 val apply = fn : (int -> 'a) -> int -> 'a
- apply square 3;   val it = 36 : int

- fun first x y = x;
 val first = fn : 'a -> 'b -> 'a
- first 2 "hello";  val it = 2 : int
```

Higher-Order Functions (I)(cont'd)

- Function application is left-associative.
  Thus, (first x y) = ((first x) y).
- Operator $\rightarrow$ is right-associative.
  Thus, 'a $\rightarrow$ 'b $\rightarrow$ 'a = 'a $\rightarrow$ ('b $\rightarrow$ 'a).
- i.e. first() has domain type = 'a, range = 'b $\rightarrow$ 'a.
- i.e. first() takes an 'a value and returns another function which takes a 'b value and returns an 'a value.

Deducing the Types

```
fun first x y = x
fn first: 'a -> 'b -> 'c
'a = 'c ==> fn first: 'a -> 'b -> 'a

fun foo f x = f x
fn foo: 'a -> 'b -> 'c
f x ==> the type of f is 'b -> 'd
'c is the type of f x ==> c' = 'd
fn foo: ('b -> 'd) -> 'b -> 'd
fn foo: ('a -> 'b) -> 'a -> 'b

fun foo f x = x f
fn foo: 'a -> 'b -> 'c
x f ==> the type of x is 'a -> 'd
'c is the type of x f ==> c' = 'd
fn foo: 'a -> ('a -> 'd) -> 'd
```

Higher-Order Functions (II)

Functions returning function:

```
- fun sq_or_twice  x = if x > 0 then square else twice;
val sq_or_twice = fn : int -> int -> int

- (sq_or_twice 2) 5;
val it = 25 : int

- sq_or_twice 2;
val it = fn : int -> int
```

Functions on List: Examples

- In general, a function on list must deal with the 2 cases:
  - [ ] or nil
  - head::tail

```
- fun len([]) = 0  |  len(x::tail) = 1 + len(tail);
- fun sum([]) = 0  |  sum(x::tail) = x + sum(tail);
- fun mean L = sum L div len L;
- mean [1,2,3];
val it = 2 : int
```

```
- fun append([], L2) = L2
  |   append(x::tail, L2) = x::append(tail, L2);
- append([3,5], [9,8,7]);
val it = [3,5,9,8,7] : int list
```

List Function: map

- The built-in **map**() has 2 arguments: a function *f()* and a list.
- It applies function *f()* to each element of the list.

fun map *f* [ ] = [ ]
| **map** *f* (head::tail) = (*f* head)::(**map** *f* tail);

  - Type of list: 'a list
  - Type of *f*: 'a → 'b
  - Type of **map**: ('a → 'b) → 'a list → 'b list

map: Examples

```
- fun odd x = (x mod 2) = 1;
val odd = fn : int -> bool

- map odd [1,2,3];
val it = [true,false,true] : bool list

- map odd;   What is the result?
val it = fn : int list -> bool list

- map;
val it = fn : ('a -> 'b) -> 'a list -> 'b list
```

List Function: filter

- **filter** applies a boolean test function to each element of a list, removing the element should the test fail.

fun filter f [ ] = [ ]
| filter f (head::tail) = if (f head)
then head::(filter f tail)
else (filter f tail);

```
- filter;
val it = fn : ('a -> bool) -> 'a list -> 'a list
- filter odd;
val it = fn : int list -> int list
- filter odd [1,2,3,4,5];
val it = [1,3,5] : int list
```

List Function: reduce

- **reduce** accumulates a result from a list.

fun reduce f [ ] v = v
| reduce f (head::tail) v = f (head, reduce f tail v);

```
- fun add (x,y) = x+y;

- reduce add [1,2,3,4,5] 0;
val it = 15 : int

- reduce;
val it = fn : ('a * 'b -> 'b) -> 'a list -> 'b -> 'b

- reduce add;
val it = fn : int list -> int -> int

- reduce add [1,2,3,4,5];
val it = fn : int -> int
```

List Function: Example

```
- fun reverse_([], L2) = L2
  |   reverse_(x::tail, L2) = reverse_(tail, x::L2);
- fun reverse L = reverse_(L, []);

- reverse ["D","O","G"];
val it = ["G","O","D"] : string list
```

The same function defined using “let”:

```
fun reverse L =
    let fun rev_([], L2) = L2
          |   rev_(x::tail, L2) = rev_(tail, x::L2)
    in  rev_(L, []) end;
```

- *rev*: 'a list → 'a list, is SML's built-in operator to do that.

Anonymous Functions

Syntax: **fn** <formal parameter> $\Rightarrow$ <body>

- An anonymous function is a function without a name.
- Used when only a locally defined function is needed.

```
- map (fn x => x*x) [2,3,4];
val it = [4,9,16] : int list

- map (fn (x,_) => x) [(1,2), (3,4), (5,6)];
val it = [1,3,5] : int list
```

Composite Functions

Given: f: 'b $\rightarrow$ 'c and g: 'a $\rightarrow$ 'b .

Define a new function: $h(x) = f \circ g(x) \equiv f(g(x))$: 'a $\rightarrow$ 'c.

i.e first apply function g() to an input x of 'a type, returning a value of 'b type, which is then piped into function f() to give the final result of 'c type.

```
- fun square x = x*x;      fun twice x = 2*x;
val square = fn : int -> int
val twice = fn : int -> int

- val sq_twice = square o twice; (* Use val NOT fun *)
val sq_twice = fn : int -> int

- sq_twice 3;
val it = 36 : int
```

Creating New Infix Operators

Left-associative: **infix** <precedence-level> <operator identifier>.
Right-associative: **infixr** <precedence-level> <operator identifier>.

- If omitted, <precedence-level> is taken as 0 — the lowest level.

PRECEDENCE	OPERATORS	ASSOCIATIVITY	COMMENTS
3	o	—	function composition
	:=	—	assignment
4	=, <>, <, >, ≤, ≥	left	relational operators
5	::	right	list constructor
	@	right	list concatenation
6	+, -	left	add/subtract
	^	left	string concatenation
7	*, /, div, mod	left	multiply/divide

New Operator (cont'd)

- Create the function

```
- fun **(a,0) = 1 | **(a,b) = a * **(a,b-1);
val ** = fn : int * int -> int
```

- Test it:

```
- **(2,5);
val it = 32 : int
```

- Declare it as a left associative operator:

```
- infix 7 **;
infix 7 **
- 4 + 2**5 - 6;
val it = 30 : int
```

```
- 2**3**2;
val it = 64 : int
```

- Or declare it as a right associative operator:

```
- infixr 7 **;
infixr 7 **
- 2**3**2;
val it = 512 : int
```

Defining New Datatype

Syntax: **datatype** <type-name>
= <1st-constructor> | <2nd-constructor> | ...

- A simple example:

```
datatype Primary_Lights = red | green | blue;

- red;
val it = red : Primary_Lights
```

- similar to enumeration in C++

enum Primary_Lights = { red, green, blue };

Constructors of Datatype

- More complex objects can be constructed too. e.g.

```
datatype Money = nomoney       fun amount nomoney = 0
   | coin of int                 | amount(coin(x)) = x
   | note10 of int               | amount(note10(x)) = 10*x
   | note100 of int              | amount(note100(x)) = 100*x
   | check of string*int;        | amount(check(bank,x)) = x;

- amount (note100(2));
val it = 200 : int
```

- Money has 5 constructors: nomoney as a constant constructor, coin(int), note10(int), note100(int), and check(string, int).
- Any function on Money should deal with all five cases, one for each constructor.

Recursive Datatype

```
- datatype expr = constant of int
                | variable of string
                | sum of expr * expr
                | product of expr * expr;
- val zero = constant 0; val one = constant 1;

- fun D x (constant _) = zero
  | D (variable w) (variable z) = if w = z then one else zero
  | D x (sum(e1, e2)) = sum(D x e1, D x e2)
  | D x (product(e1, e2)) =
          let val term1 = product(D x e1, e2)
              val term2 = product(e1, D x e2)
          in sum(term1, term2) end;
val D = fn : expr -> expr -> expr
```

- expr has 4 constructors: constant(int), variable(string), sum(expr, expr), product(expr, expr).

Polymorphic Datatype: Binary Tree Example

```
datatype 'a tree =
    empty_tree | leaf of 'a | node of 'a tree*'a tree;
```

- The 'a tree has 3 constructors: empty_tree (constant constructor), leaf('a), and node('a tree, 'a tree).

```
- fun leafcount(empty_tree) = 0
    | leafcount(leaf(x)) = 1
    | leafcount(node(L,R)) = leafcount(L) + leafcount(R);
val leafcount = fn : 'a tree -> int

- val x = node(node(leaf(1), leaf(2)), leaf(3));
val x = node (node (leaf #,leaf #),leaf 3) : int tree

- leafcount x;
val it = 3 : int
```

Impure FP: Ref-Variables, Assignments

- Reference variable points to a value (similar to indirect addressing):
  val <identifier> = **ref** <expression>.
- Assignment: <identifier> **:=** <expression>
- Dereference: **!**<identifier>

```
- val x = ref(2+3);              |  - val y = ref 9;
val x = ref 5 : int ref          |  val y = ref 9 : int ref
                                 |
- x := 9;                        |  - !x = !y;
val it = () : unit               |  val it = true : bool
                                 |
- x;                             |  - x = y;
val it = ref 9 : int ref         |   val it = false : bool

- !x;
val it = 9 : int
```

Value Binding and Environment

- The phrase: "**val** x = 17" is called a value binding; the variable x is bound to the value 17.
- When an identifier is declared by a value binding, a *new* identifier is "created" — it has nothing whatever to do with any previously declared identifier of the same name.
- Once an identifier is bound to a value, there is no way to change that value.
- Environment: the current set of ordered pairs (identifier, value) that are visible.

Environment: Example

- val x = 17;

val x = 17 : int

- val y = x;

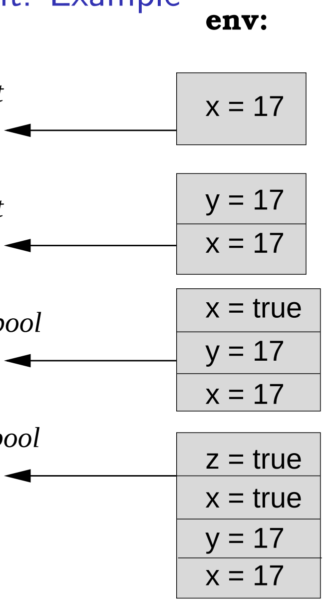
val y = 17 : int

- val x = true;

val x = true : bool

- val z = x;

val z = true : bool



Assignment and Side Effects

- val x = ref 0; *val x = ref 0 : int ref*	**state: { (x, 0) }**
- x := 17; *val it = () : unit*	**state: { (x, 17) }**
- val y = x; *val y = ref 17 : int ref*	**state: { (x, 17), (y, 17) }**
- x := 9; *val it = () : unit*	**state: { (x, 9), (y, 9) }**
- val z = x; *val z = ref 9 : int ref*	**state: { (x, 9), (y, 9), (z, 9) }**

- Notice how the assignment x := 9 produces the side-effects such that not only x's derefenced value is changed, but also y's.

Summary: Functional Programming Languages

Features of functional programming languages:

- Functions are first-class objects: They may be (1) passed as arguments (2) returned as results (3) stored in variables. (simiar to variables in imperative languages.)
- Basic mode of computation: construction and application of functions. (as opposed to assignments in imperative languages.)
- Principal control mechanism: recursive function applications. (in place of for-/while-loops in imperative languages.)
- Free from side-effects (for "pure" FPL).