Lecture 3
Principles of Functional Programming
Summer 2020

Lists & Structural Induction

One thing leads to another...

Section 1

Another word on pattern matching

Syntax

Recall that we had several ways of pattern matching:

- Lambda expression clauses:

```
val isZeroOrOne : int -> bool
    = fn 0 => true | 1 => true | _ => false
```

- `fun` declaration clauses

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

- `case` expressions

```
fun fact (n:int):int =
  case n of
    0 => 1
  | _ => n * fact(n-1)
```

- `val` declarations

```
val 8 = power 3
```

Allowed patterns

- Constructors

```
fn true => e1 | false => e2
```

- Variable names

```
fn (x:int) => x
```

- Wildcards

```
fn (_ : string) => 2
```

- Tuples of patterns

```
fun foo ((0,0),_) = "a"
  | foo ((_,0),(7,_)) = "b"
  | foo ( _, (8,8)) = "c"
  | foo _ = "d"
```

Not patterns

- Function applications

```
(* Doesn't work *)
val m+n = 2
val (s1 ^ s2) = "hello world"
```

- Non-match-able types

```
(* Doesn't work *)
val (fn x => e) : int -> string = f
```

- Repetitive patterns

```
(* Doesn't work *)
fun equal (m:int,m:int) = true
  | equal _ = false
```

Comparing cases

```
case true of
  true => 1
| b => 2
```

```
case true of
    b => 2
 | true => 1
```

bool casing

Note: the following are equivalent:

```
case b of
  true => e1
| false => e2
```

```
if b then e1 else e2
```

Common error: the "flase" bug

3.0

```
case b of
     flase => 2
   | true => 1
```

int casing

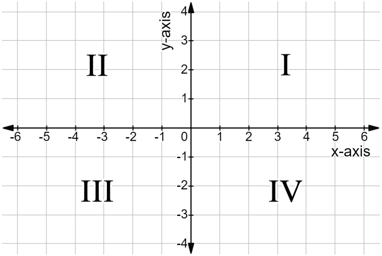
3.1

```
(* REQUIRES: n>=0 *)
fun divByThree (0:int):bool = true
  | divByThree 1 = false
  | divByThree 2 = false
  | divByThree n = divByThree(n-3)
```

```
(* Doesn't work *)
fun abs 0 = 0
  | abs ~n = n
  | abs n = n
```

```
fun abs 0 = 0
  | abs n = if n<0 then ~n else n
```

Quadrants



`quadrant : int * int -> string`
REQUIRES: true
ENSURES: `quadrant(x,y)` evaluates to either `"I"`, `"II"`, `"III"`, `"IV"` or `"boundary"`, if `(x,y)` is in the first, second, third, fourth quadrant, or on one of lines, respectively

Version 1

3.2

```
fun quadrantV1 (m:int,n:int):string =
   if m=0 orelse n=0
   then "boundary"
   else if m>0
        then if n>0
             then "I"
             else "IV"
        else if n<0
             then "II"
             else "III"
```

Version 2

3.3

```
fun quadrantV2 (0,_) = "boundary"
  | quadrantV2 (_,0) = "boundary"
  | quadrantV2 (m:int,n:int):string =
       if m>0
       then if n>0
            then "I"
            else "IV"
       else if n<0
            then "II"
            else "III"
```

The order type

SML has a built-in type to encode orderings, `order`.

- There are three constructors of type `order`:

  `LESS` `EQUAL` `GREATER`

- These are also the only values of this type
- The following values are built-in to SML:

```
val Int.compare
    : int * int -> order
val String.compare
    : string * string -> order
```

3.4

```
fun quadrant (m:int,n:int):string =
  case (Int.compare(m,0),Int.compare(n,0)) of
       (EQUAL, _ ) => "boundary"
     | ( _ , EQUAL) => "boundary"
     | (GREATER,GREATER) => "I"
     | (LESS,GREATER) => "II"
     | (LESS,LESS) => "III"
     | (GREATER,LESS) => "IV"
```

Section 2

Lists

The list type

- For each type codet, there is a type

  `t list`

  of *lists of elements of* `t`
- There are two constructors of type `t list`:
  - `[]: t list`
  - If `x:t` and `xs:t list`, then

    `(x::xs) : t list`
- The values of type `t list` are lists `[x1,x2,...,xn]`, including `[]`. This is just syntactic sugar for `[]` and `::`, however:
  - `[1]:int list` is `1::[]`
  - `["functions","are","values"] : string list` is just `"functions"::"are"::"values"::[]`

```
len : int list -> int
```
REQUIRES: true

ENSURES: `len L` evaluates to the length of `L`

3.5

```
fun len ([] : int list):int = 0
  | len (x::xs) = 1 + len xs

val 5 = len [1,2,3,4,5]
val 2 = len [~5000,19]
val 0 = len []
```

```
(op @) : int list * int list -> int list
```

REQUIRES: true

ENSURES: If `L1` is a list of length m and `L2` is a lsit of length n, then `L1@L2` evaluates to a list of length $m+n$ whose first m elements are the elements of `L1` (in the same order they appear in `L1`) and whose last n elements are the elements of `L2` (in the same order they appear in `L2`)

3.6

```
infix @
fun ([]:int list) @ (L:int list) = L
  | (x::xs) @ L = x::(xs@L)

val [1,2,3,4] = [1,2]@[3,4]
val [1,2] = []@[1,2]
val [1,2] = [1,2]@[]
```

```
rev : int list -> int list
```

REQUIRES: true

ENSURES: `rev L` evaluates to a list containing exactly the elements of `L`, in the opposite order they appeared in `L`

3.7

```
fun rev ([]:int list):int list = []
  | rev (x::xs) = (rev xs)@[x]

val [3,2,1] = rev [1,2,3]
val [] = rev []
```

A claim

I claim that, for all types `t` and all values `L : t list`,

$$\texttt{len(rev L)} \cong \texttt{len L}$$

How do we prove this?

Section 3

Structural Induction

The Principle of Structural Induction on Lists

Let `t` be some type. In order to show that a statement P holds of all values `L : t list`, it suffices to show:

- **(BC)** $P($`[]`$)$ holds
- **(IS)** Assuming $P($`xs`$)$ holds for some `xs : t list` **(IH)**, show for any value `x : t` that $P($`x::xs`$)$ holds

Why does it work? Well, every value of type `t list` is either `[]` or of the form `x::xs` for some `x`,`xs`.

$P($`[]`$)$ implies $P($`[1]`$)$ implies $P($`[4,1]`$)$ implies $P($`[3,4,1]`$)$ implies ...

Example: Totality of `len`

Theorem

For all values `L : int list`, `len L` evaluates to some value

Proof:
BC: `L=[]`
WTS: `len []` evaluates to a value

$$\texttt{len []} \Longrightarrow \texttt{0} \qquad \text{(first clause of len)}$$

IS: `L=x::xs` for some `x:t` and some `xs:t list`
IH: `len xs` evaluates to some value.
WTS: `len (x::xs)` evaluates to a value

$$\begin{aligned} \texttt{len (x::xs)} &\Longrightarrow \texttt{1 + len xs} && \text{(second clause of len)} \\ &\Longrightarrow \texttt{1 + v} && \text{(for some value v, by (IH))} \\ &\Longrightarrow \texttt{v'} && \text{(for some value v')} \end{aligned}$$

Another one

Theorem

For all values $L1 : \texttt{int list}$ *and* $L2 : \texttt{int list}$,

$$\texttt{len(L1@L2)} \cong \texttt{len(L1) + len(L2)}$$

Proof: Left as exercise (hint: induct on `L1`)

Thank you!