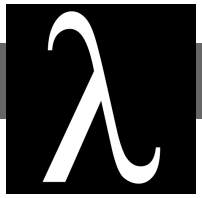

Contents

1 Induction and Recursion

We will learn about pattern matching, recursive thinking, and induction.

1.1 Pattern Matching

Pattern matching is when we take an evaluated expression and attempt to match it against a pattern. There are five types of patterns – *variables*, *tuples*, *constructors*, *constants*, and *wildcards*.

Declarations use pattern matching to ***extract*** values from an expression and ***bind*** them to variables—this is particularly useful when extracting from ***tuples*** or ***constructors***.

```
val <pattern> = <expression>
```

Let expressions make temporary declarations, whose bindings can be used within the scope of the expression/declaration.

```
let
  <declarations>
in
  <expression>
end
```

Casing and Clauses. `case` expressions[1] and functions can handle expressions differently based on what clause[2] pattern they match to.

```
case <expr> of          | fn <ptrn> => <expr>  | fun <name> <ptrn> = <expr>
  <ptrn> => <expr>      |  | <ptrn> => <expr>  |   | <name> <ptrn> = <expr>
| <ptrn> => <expr>      |  | <ptrn> => <expr>  |   | <name> <ptrn> = <expr>
```

To match exhaustively with all possible remaining patterns, we can use ***wildcards***.

[1] When in doubt, wrap your entire case expression with parentheses. Without them, SML might get confused by nested cases—a common source of bugs and type errors.

[2] Clauses are numbered by the patterns matched: `(case ... of clause 1 | clause 2 | clause 3)`

1.2 Thinking Recursively

It can be helpful to think about implementing `recursive functions` as **proofs by induction**:

1. The `base cases` should be the same as the **base cases of your proof**.
2. The `recursive cases` should be the same as the **induction step**.
3. The `recursive calls` mirror the **inductive hypothesis** since it's being used to solve a smaller part of the problem.

By "applying the **inductive hypothesis**", we can assume a `recursive call` will satisfy the function's spec, and then construct the recursive case based on that assumption.

1.3 Induction Proofs

The principle of induction states that, in order to prove a statement about an inductive type (e.g., a integer), it suffices to prove:

Task 1.1.

1. **Base Case(s):** the non-recursive part(s) of your code. When trying to prove something of the form for all $n \in \mathbb{N}, P(n)$ for a property P, this is often $P(\_\_\_\_\_)$.
2. **Inductive Step:** the recursive parts of your code, assuming that the property holds for smaller cases.
   - Simple Induction over $\mathbb{N}$:
   - Strong Induction over $\mathbb{N}$:

Solution 1.

1. **Base Case:** $P(0)$
2. **Inductive Cases:**
   - For simple induction: If $P(k)$ (IH), then $P(k+1)$.
   - For strong induction: If $P(k)$ for all k s.t. $0 \leq k < n$ (IH), then $P(n)$.

Check out the 15-150 Proof Guidelines for additional proof-writing guidance.

Task 1.2.

Let's prove that two exponentiation functions are equivalent:

```
fun slow (0 : int) : int = 1
  | slow (1 : int) : int = 2
  | slow (n : int) : int = slow (n-1) + 2 * slow (n-2)

fun exp2 0 =
  | exp2 n =
```

Note: In this course, you can assume that built-in math operators (e.g. `+`, `-`) behave as intended without additional justification.

Theorem. For all $\mathtt{n} \in \mathbb{N}$, `slow n` $\cong$ `exp2 n`.

Proof. We prove by _____ induction on `n`.

□

Solution 2.

Proof. We will prove by strong induction on `n`.

Base Case 1. `n = 0`

LHS:

$$\texttt{slow 0} \cong \texttt{1} \qquad \text{(clause 1 of } \texttt{slow}\text{)}$$

RHS:

$$\texttt{exp2 0} \cong \texttt{1} \qquad \text{(clause 1 of } \texttt{exp2}\text{)}$$

Base Case 2. `n = 1`

LHS:

$$\texttt{slow 1} \cong \texttt{2} \qquad \text{(clause 2 of } \texttt{slow}\text{)}$$

RHS:

$$\begin{aligned} \texttt{exp2 1} &\cong \texttt{2 * exp2 (1-1)} && \text{(clause 2 of } \texttt{exp2}\text{)} \\ &\cong \texttt{2 * exp2 0} && \text{(math)} \\ &\cong \texttt{2 * 1} && \text{(clause 1 of } \texttt{exp2}\text{)} \\ &\cong \texttt{2} && \text{(math)} \end{aligned}$$

Induction Step. `n > 1`

Induction Hypothesis: Assume for all i such that $0 \leq i < n$, $\texttt{slow i} \cong \texttt{exp2 i}$.

We prove $\texttt{slow n} \cong \texttt{exp2 n}$.

LHS:

$$\begin{aligned} \texttt{slow n} &\cong \texttt{slow (n-1) + 2 * slow (n-2)} && \text{(clause 3 of } \texttt{slow}\text{)} \\ &\cong \texttt{exp2 (n-1) + 2 * exp2 (n-2)} && \text{(IH)} \end{aligned}$$

RHS:

$$\begin{aligned} \texttt{exp2 n} &\cong \texttt{2 * exp2 (n-1)} && \text{(clause 2 of } \texttt{exp2}\text{)} \\ &\cong \texttt{exp2 (n-1) + exp2 (n-1)} && \text{(math)} \\ &\cong \texttt{exp2 (n-1) + 2 * exp2 (n-2)} && \text{(clause 2 of } \texttt{exp2}\text{)} \end{aligned}$$

Thus, the claim is proven. □

Note: In this course, proof steps that deal with the behavior or valuability of applying built-in math operators (e.g. `+`, `-`), can be made without lemmas or additional justification.

2 Recursion on the Naturals

In this lab, we will write several recursive functions over the natural numbers.

Your functions should case on the argument using two branches:

1. The first branch specifies the base case when the argument is zero.
2. The second branch specifies the induction case when the argument is greater than zero. This branch will include a recursive application of the function to an argument that is one less.

You will practice two different ways of implementing these functions:

- Using a case expression:

```
fun f (x : int) : int =
 (case x of
    0 => (* base case *)
  | _ => ... f (x - 1) ...)
```

- Using clauses:

```
fun f (0 : int) : int = (* base case *)
  | f (x : int) : int = ... f (x - 1) ...
```

2.1 Summorial

Task 2.1. (Recommended)

Define the `summ` function and its spec in `code/summorial/summ.sml` such that for `n` ≥ 0, `summ n` equals the sum of the natural numbers from `0` to `n`.

$$\texttt{summ n} \cong \sum_{i=0}^{n} i$$

Constraint: Make sure your function is ***recursive***. You may not used the closed-form formula.

- What should the type of `summ` be?
- As you write the body of the `summ` function, attempt to justify its correctness to yourself[3].

For practice, try writing two versions of this function: one using a case expression and the other using clauses.

Solution 1.

```
(* summ : int -> int
 * REQUIRES: n >= 0
 * ENSURES: summ n returns the sum of the integers 0 to n. *)
fun summ (n : int) : int =
    (case n of
      0 => 0
    | n => n + summ(n - 1))
```

[3] Think recursively!

```
(* summ :
 * REQUIRES:
 * ENSURES: *)
fun summ (0 : int) : int = 0
  | summ n = n + summ(n - 1)
```

Task 2.2. (Recommended)

Write a few test cases based on your code for `summ`. Consider different cases and cover them!

Solution 2.

```
val () = Test.int ("summ 0", 0, summ 0)
val () = Test.int ("summ 4", 10, summ 4)
val () = Test.int ("summ 5", 15, summ 5)
```

3 Another Pattern of Recursion

3.1 Fairly WhatParity

> The Fairly OddParents met another family with Fairly EvenParents but they can't tell who's who! Write these functions to help them figure out who is a Fairly OddParent and who is a Fairly EvenParent.

We want to define a function `evenP : int -> bool` which takes a natural number `n` and evaluates to `true` if `n` is even, and `false` otherwise. However, we're in a recursive mood, so we want to define `evenP` recursively!

To define this function on all natural numbers, it suffices to give cases for

- `0`
- `1`
- $n > 1$, using a recursive call on $n - 2$

Therefore, the `case` expression in the body of `evenP` has three branches rather than two. The first two branches give the base cases, and the third branch includes a recursive application of the function to the natural number that is two less than the argument:

```
fun evenP (x : int) : bool =
 (case x of
    0 => true
  | 1 => false
  | _ => evenP (x - 2))
```

Convince yourself that this function behaves as expected, and that it will return a value for all natural number inputs.

Task 3.1.

Write a few test cases based on the code for `evenP`. Try to consider different cases and cover them!

Solution 1.

```
10 val () = Test.bool ("evenP 0", true, evenP 0)
11 val () = Test.bool ("evenP 1", false, evenP 1)
12 val () = Test.bool ("evenP 12", true, evenP 12)
13 val () = Test.bool ("evenP 27", false, evenP 27)
```

We will now define the `oddP` function using this pattern.

Task 3.2.

In `code/divisibility/divisibility.sml`, define the function meeting the following specification.

`oddP : int -> bool`

REQUIRES: $n \geq 0$

ENSURES: `oddP n` $\Longrightarrow$ `true` if `n` is odd and evaluates to `false` otherwise.

Constraint: Do not call `evenP` or `mod` in the definition of `oddP`.

Solution 2.

```
18 fun oddP (0 : int) : bool = false
19   | oddP 1 = true
20   | oddP n = oddP (n - 2)
```

Task 3.3.

Write a few test s based on your code for `oddP`. Try to consider different cases and cover them!

Solution 3.

```
23 val () = Test.bool ("oddP 0", false, oddP 0)
24 val () = Test.bool ("oddP 1", true, oddP 1)
25 val () = Test.bool ("oddP 6", false, oddP 6)
26 val () = Test.bool ("oddP 37", true, oddP 37)
```

Task 3.4.

Using an inductive proof, prove that, for all natural numbers `n`,

$$\texttt{oddP n} \cong \texttt{not (evenP n)}$$

where `not : bool -> bool` is defined as:

```
fun not (true  : bool) : bool = false
  | not (false : bool) : bool = true
```

Pay particular attention to how the recursive structure of `oddP` dictates the structure of your proof.

Solution 4.

We will use this definition of `oddP`:

```
fun oddP (0 : int):bool = false
  | oddP (1 : int):bool = true
  | oddP (n : int):bool = oddP(n-2)
```

Theorem. For all natural numbers `n`, `oddP n` $\cong$ `not (evenP n)`.

Proof. By weak induction on `n`.

BC - 0: `n` $=$ `0`. Showing:

LHS:

$$\texttt{oddP 0} \cong \texttt{false} \qquad \text{(Clause 1 of oddP)}$$

RHS:

$$\begin{aligned} \texttt{not (evenP 0)} &\cong \texttt{not (true)} && \text{(Case 1 of evenP)} \\ &\cong \texttt{false} && \text{(defn of not)} \end{aligned}$$

BC - 1: `n` $=$ `1`. Showing:

LHS:

$$\texttt{oddP 1} \cong \texttt{true} \qquad \text{(Clause 2 of oddP)}$$

RHS:

$$\begin{aligned} \texttt{not (evenP 1)} &\cong \texttt{not (false)} && \text{(Case 2 of evenP)} \\ &\cong \texttt{true} \end{aligned}$$

IS: `n` $=$ `k+2` for some natural number `k`.
IH: Assume that `oddP k` $\cong$ `not(evenP k)`.
Want to show: `oddP(k+2)` $\cong$ `not(evenP(k+2))`.
Showing:

LHS:

$$\begin{aligned} \texttt{oddP(k+2)} &\cong \texttt{oddP ((k+2)-2)} && \text{(Clause 3 of oddP)} \\ &\cong \texttt{oddP k} && \text{(math)} \\ &\cong \texttt{not (evenP k)} && \text{(IH)} \end{aligned}$$

RHS:

$$\begin{aligned} \texttt{not (evenP (k + 2))} &\cong \texttt{not (evenP ((k + 2) - 2))} && \text{(Case 3 of evenP)} \\ &\cong \texttt{not (evenP k)} && \text{(math)} \end{aligned}$$

By induction, the theorem holds for all natural numbers `n`. □

3.2 Name a more iconic trio... I'll wait

After restoring order to the Fairly OddParents, you are contacted by Alvin and the Chipmunks! They request a function that quickly determines whether a quantity of acorns can be evenly

split amongst the three of them.

Next, you will define a function, `divisibleByThree : int -> bool`, that lives up to its name.

Task 3.5.

`divisibleByThree : int -> bool`

REQUIRES: `n` ≥ 0

ENSURES: `divisibleByThree n` $\Longrightarrow$ `true` if `n` is a multiple of 3 and to `false` otherwise.

Constraint: Do not use the SML `mod` operator for this task.

Solution 5.

```
(* Base cases for 0, 1, 2. Use recursive case for n >= 3      *)
(*                                                            *)
(* We have three base cases corresponding to each distinct    *)
(* value mod 3 and an inductive case for 3 plus another       *)
(* natural number.                                            *)
```

```
fun divisibleByThree (0 : int) : bool = true
  | divisibleByThree 1 = false
  | divisibleByThree 2 = false
  | divisibleByThree n = divisibleByThree (n - 3)
```

```
val () = Test.bool ("divisibleByThree 0", true, divisibleByThree 0)
val () = Test.bool ("divisibleByThree 1", false, divisibleByThree 1)
val () = Test.bool ("divisibleByThree 2", false, divisibleByThree 2)
val () = Test.bool ("divisibleByThree 6", true, divisibleByThree 6)
val () = Test.bool ("divisibleByThree 7", false, divisibleByThree 7)
val () = Test.bool ("divisibleByThree 11", false, divisibleByThree 11)
```

4 GCD

4.1 Euclid's Algorithm

Euclid's algorithm for computing the greatest common divisor of two numbers rests on the following two observations:

1. The GCD of two numbers $m > n$ does not change if m is replaced by $m - n$.
2. The GCD of any number and 0 is that number.

Task 4.1.

Using these two facts, implement a recursive GCD function, in `code/gcd/gcd.sml`,

> `GCD : int * int -> int`
>
> REQUIRES: `m` ≥ 0, `n` ≥ 0
>
> ENSURES: `GCD (m,n)` $\Longrightarrow$ the GCD of `m` and `n`

Solution 1.

```
4  fun GCD (m : int, 0) : int = m
5    | GCD (0, n : int) : int = n
6    | GCD (m : int, n : int) : int =
7          if m > n then GCD (m - n, n) else GCD (m, n - m)
```

Task 4.2.

Why do we restrict our inputs to be nonnegative in the REQUIRES?

Solution 2.

If we did not, then naively applying the above algorithm to negative inputs could loop forever. For example, consider `GCD (4,~2)`.

Task 4.3.

Write a few test cases based on your code for `GCD`. Try to consider different cases and cover them!

Solution 3.

```
10 val () = Test.int ("GCD (0, 0)", 0, GCD (0, 0))
11 val () = Test.int ("GCD (0, 23)", 23, GCD (0, 23))
12 val () = Test.int ("GCD (14, 21)", 7, GCD (14, 21))
```

5 Divide and... Mod

Suppose we wanted to implement division in SML (don't worry; you'll do addition and multiplication on the homework). You'll notice that the output of division is a little tricky - often, when one number does not divide another, we want to return more than one thing. If we wanted the answer to 8 divided by 3, we would want to know both that 3 goes into 8 twice and that it leaves a remainder of 2.

Fortunately, this is very straightforward to do! Just as we can write functions that take in a pair (tuple), we can write functions that evaluate to a pair of results.

The algorithm is fairly simple. Given that we are computing $\frac{n}{d}$: subtract d from n until n is less than d, at which point n is the remainder, and the number of total subtractions is the quotient.

Task 5.1. (Recommended)

In `code/divmod/divmod.sml`, write the function:

`divmod : int * int -> int * int`

REQUIRES: $n \geq 0$, $d > 0$

ENSURES: `divmod (n, d)` $\Longrightarrow$ `(q,r)` such that $qd + r = n$ and $0 \leq r < d$

Solution 1.

```
5  fun divmod (n : int, d : int) : int * int =
6    if n < d
7      then (0, n)
8      else
9        let
10         val (q, r) = divmod (n - d, d)
11       in
12         (q + 1, r)
13       end
14
15 val () = Test.int_int ("divmod (4,4)", (1,0), divmod (4,4))
16 val () = Test.int_int ("divmod (4,5)", (0,4), divmod (4,5))
17 val () = Test.int_int ("divmod (25,7)", (3,4), divmod (25,7))
18 val () = Test.int_int ("divmod (6,3)", (2,0), divmod (6,3))
```

```
5  fun divmod (n : int, d : int) : int * int =
6    if n < d
7    then (0, n)
8    else (fn (q, r) => (q+1, r)) (divmod (n-d, d))
```

Recall that you are not responsible for the behavior of your function when it is applied to an argument that does not meet the precondition.

Constraint:

- Integer division and modulus are built in to SML (`div` and `mod`), but ***you may not use them for this problem***. The point is to practice recursively computing a pair.
- You ***may not*** make more than one recursive call to `divmod` in the recursive case of your

function.

- You ***may not*** write/use a helper function to implement this.

6 List Practice

To get practice working with lists (and how to write recursive list functions), we will be implementing several basic list functions.

For each of the following, in `code/list-utils/utils.sml` write the type annotation, REQUIRES, ENSURES, implement the function, and write test cases. Make sure that your test cases adequately test both the base case and the recursive case!

Task 6.1. (Recommended)

Specify and implement a function `addToEach` which takes an int `n` and an int list `L` and adds `n` to every element of `L`:

```
val () = Test.int_list_eq ("add 1", [2,3,4], addToEach (1,[1,2,3]))
val () = Test.int_list_eq ("empty", [], addToEach (2,[]))
val () = Test.int_list_eq ("add 0", [3,3], addToEach (0,[3,3]))
```

Solution 1.

```
(* addToEach : int * int list -> int list
 * REQUIRES: true
 * ENSURES: addToEach(x,L) ==> L' where L' is the list formed
 *          by adding x to every element of L
 *)
fun addToEach (x:int,[]:int list):int list = []
  | addToEach (x, y::ys) = (x+y)::addToEach(x,ys)
```

```
val () = Test.int_list_eq ("add 1", [2,3,4], addToEach (1,[1,2,3]))
val () = Test.int_list_eq ("empty", [], addToEach (2,[]))
val () = Test.int_list_eq ("add 0", [3,3], addToEach (0,[3,3]))
```

Task 6.2. (Recommended)

Specify and write a function `mult` which takes an int list `L` and multiplies together all the elements of `L`:

```
val () = Test.int ("empty", 1, mult [])
val () = Test.int ("zero", 0, mult [1,2,3,4,5,0])
val () = Test.int ("1*2*3*1", 6, mult [1,2,3,1])
```

Solution 2.

```
(* mult : int list -> int
 * REQUIRES: true
 * ENSURES: mult L ==> the product of the elements of L
 *)
fun mult ([]:int list):int = 1
  | mult (x::xs) = x * (mult xs)
```

```
val () = Test.int ("empty", 1, mult [])
val () = Test.int ("zero", 0, mult [1,2,3,4,5,0])
val () = Test.int ("1*2*3*1", 6, mult [1,2,3,1])
```

Task 6.3.

Specify and write a function `trues` which takes a `bool list` `L` and removes all the `false`'s from `L`:

```
val () = Test.bool_list_eq ("base case", [], trues [])
val () = Test.bool_list_eq ("all false", [], trues [false, false,
    false])
val () = Test.bool_list_eq ("TFT", [true,true], trues [true, false,
    true])
```

Solution 3.

```
(* trues : bool list -> bool list
 * REQUIRES: true
 * ENSURES: trues L ==> L' where L' is just L without false's
 *)
fun trues ([] : bool list):bool list = []
  | trues (true::xs) = true :: trues xs
  | trues (false::xs) = trues xs
```

```
val () = Test.bool_list_eq ("base case", [], trues [])
val () = Test.bool_list_eq ("all false", [], trues [false, false, false])
val () = Test.bool_list_eq ("TFT", [true,true], trues [true, false, true])
```

Task 6.4.

Specify and write a function `trues'` which takes an `(int * bool) list` and removes all instances of `(x,false)` from the list and returns an `int list` with all the true elements in the original order:

```
val () = Test.general_eq ("base case", [], trues' [])
val () = Test.general_eq ("all false", [], trues' [(1,false),(6,
    false)])
val () = Test.general_eq ("TFTFT", [1,3,5], trues' [(1,true),(2,
    false),(3,true),(4,false),(5,true)])
```

Solution 4.

```
(* trues' : (int * bool) list -> int list
 * REQUIRES: true
 * ENSURES: trues' L ==> L', where L' is a list of all those
 *          integers n such that (n, true) is in L
 *)
fun trues' ([] : (int*bool) list):int list = []
  | trues' ((n,false)::xs) = trues' xs
  | trues' ((n,true)::xs) = n::trues' xs
```

```
val () = Test.general_eq ("base case", [], trues' [])
val () = Test.general_eq ("all false", [], trues' [(1,false),(6,false)])
val () = Test.general_eq ("TFTFT", [1,3,5], trues' [(1,true),(2,false),(3,true)
    ,(4,false),(5,true)])
```

Task 6.5.

Look at the `take` function and the `bake` function. Fill in their specifications (part of it has been done for you).
Note: Though the functions look similar, they behave differently.

Solution 5.

```
(* take : int * int list -> int list *  int list
 * REQUIRES: 0 <= i <= length L
 * ENSURES: take (i, L) = a pair of lists (A, B)
 * such that length A = i, and A @ B = L
 *)
```

```
(* bake : int * int list -> int list *  int list
 * REQUIRES: 0 <= i <= length L
 * ENSURES:  bake (i, L) = a pair of lists (A, B)
 * such that A contains i elements, all of which
 * are equal to the first element of L, and B = L
 *)
```

7 A def*in*ition

7.1 Definitions

What does it mean for a value `x` to be "in" a list `L`? We know what it means intuitively, but if we want to use this notion in function specifications and be able to prove things about this notion, we need to define it formally. In our definition, it might help to think about how we might write proofs using the definition. For our purposes, here is a way to define "`x` is in `L`":

Definition. For all types `t` and values `x : t`,

1. for all values `L : t list`, `x` is in `x::L`, and
2. if `x` is in `L : t list`, then for all `y : t`, `x` is in `y::L`.

We may say "`x` is in `L`" if and only if we can derive that fact from the above rules. For example, we know that `2` is in the list `[1,2,1]` because by part 1 of the above definition, `2` is in `2::[1]`, and therefore by part 2 of the above definition, `2` is in `1::2::[1]`.

This form of a definition is called an *inductive* definition, that is, a definition which make reference to itself. The advantage of such a definition is that we could translate it fairly readily into recursive SML code, and then write proofs by induction using our definition, as you will see later. But before you proceed, take a moment to convince yourself that this definition indeed encodes the intuitive notion of what it means for an element to be 'in' a list.

7.2 Justification

The main advantage of writing our definition recursively is that it is naturally conducive to writing corresponding SML functions. To see this, we'll write a function `isIn` which takes an `int` and an `int list` and returns whether the given integer is in the given list:

> `isIn : int * int list -> bool`
>
> REQUIRES: true
>
> ENSURES: `isIn (x,L)` $\Longrightarrow$ `true` if `x` is in `L`, and `isIn (x,L)` $\Longrightarrow$ `false` otherwise

```
fun isIn (x : int, [] : int list) : bool = false
  | isIn (x, y::ys) = (x=y) orelse isIn (x,ys)
```

We want to prove that this function satisfies its specification. We will use the following lemma:

Lemma. For all types `t` and all values `x: t`, `x` is not in a list of length 0. [4]

And we want to prove the theorem below:

Theorem. For all values `x : int` and all values `L : int list`, `isIn(x,L)` $\Longrightarrow$ `true` if `x` is in `L`, and `isIn(x,L)` $\Longrightarrow$ `false` otherwise

[4] And here's a fun exercise: try to prove this lemma!

Proof. [*We have left (A), (B), (C), and (D) blank for you to fill on the next page.*]

Let `x : int` be an arbitrary value. We will proceed by simple induction on the length of `L`.

Base Case: Let `L = []`. Since `L` has length 0, by **Lemma**, `x` is not in `L`. Furthermore, `isIn(x,[])` $\Longrightarrow$ `false` by clause 1 of `isIn`. Therefore, the base case holds, as desired.

Induction Hypothesis: Let `ys : int list` have length n. Assume that `isIn(x,ys)` $\Longrightarrow$ `true` if `x` is in a list of length n, and `isIn(x,ys)` $\Longrightarrow$ `false` otherwise.

Inductive step. Let `L = y::ys`, for some `y: int`. Therefore, `L` has length $n+1$. We want to show that the **Theorem** holds for `isIn(x,y::ys)`

- Case 1: `x` is in `y::ys`

  In this case, we want to show that `isIn(x,y::ys)` $\Longrightarrow$ `true`. Since we assumed `x` is in `y::ys`, we can case on whether `x=y` or `x<>y`.

  - Case 1a: `x = y`

    `isIn(x,y::ys)` $\Longrightarrow$ `(x=y) orelse isIn(x,ys)` (Clause 2 of `isIn`)
    $\Longrightarrow$ `true orelse isIn(x,ys)` (assumed `x=y`)
    $\Longrightarrow$ `true` (evaluation of `orelse`)

  - Case 1b: `x <> y`

    `isIn(x,y::ys)` $\Longrightarrow$ `(x=y) orelse isIn(x,ys)` (A)
    $\Longrightarrow$ `false orelse isIn(x,ys)` (assumed `x<>y`)
    $\Longrightarrow$ `isIn(x,ys)` (evaluation of `orelse`)
    $\Longrightarrow$ `true` (IH, and `x` is in `ys`)

  We have shown that in Case 1 `isIn(x,y::ys)` $\Longrightarrow$ `true` as desired.

- Case 2: `x` is not in `y::ys`

  In this case, we want to show that `isIn(x,y::ys)` $\Longrightarrow$ `false`. If `x` is not in `y::ys`, then by the negation of the definition, `x` is not equal to `y` and `x` is not in `ys`. Then,

  `isIn(x,y::ys)` $\Longrightarrow$ `(x=y) orelse isIn(x,ys)` (A)
  $\Longrightarrow$ `false orelse isIn(x,ys)` (B)
  $\Longrightarrow$ `isIn(x,ys)` (C)
  $\Longrightarrow$ `false` (D)

  We have shown that in Case 2 `isIn(x,y::ys)` $\Longrightarrow$ `false` as desired.

By simple induction, the theorem holds for all `L : int list` for the fixed `x : int`.

Since our choice of `x : int` was arbitrary, the theorem holds for all values `x : int` and all values `L : int list`, as desired. □

Task 7.1.

What justification should be given for the step labeled (A)?

Solution 1.

Clause 2 of `isIn`

Task 7.2.

What justification should be given for the step labeled (B)?

Solution 2.

Assumed: `x<>y`

Task 7.3.

What justification should be given for the step labeled (C)?

Solution 3.

evaluation of `orelse`

Task 7.4.

What justification should be given for the step labeled (D)?

Solution 4.

IH, and assumption that `x` is not in `ys`