## Contents

1 **Induction and Recursion**                        2  
   1.1 Pattern Matching .................................... 2  
   1.2 Thinking Recursively ................................ 3  
   1.3 Induction Proofs ..................................... 3  

2 **Recursion on the Naturals**                     4  
   2.1 Summorial ............................................. 4  
      Task 1. ................................................... 4  
      Task 2. ................................................... 4  

3 **Another Pattern of Recursion**                  5  
   3.1 Fairly WhatParents .................................. 5  
      Task 3. ................................................... 5  
      Task 4. ................................................... 5  
      Task 5. ................................................... 5  
      Task 6. ................................................... 6  
   3.2 The Bees Threes ...................................... 6  
      Task 7. ................................................... 6  

4 **GCD**                                            7  
   4.1 Euclid’s Algorithm .................................. 7  
      Task 8. ................................................... 7  
      Task 9. ................................................... 7  
      Task 10. .................................................. 7  

5 **Divide and... Mod**                             8  
   Task 11. ................................................... 8
1 Induction and Recursion

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

1.1 Pattern Matching

Pattern matching is when we take a value and attempt to match it against a pattern. There are five types of patterns – variables, tuples, constructors, constants, and wildcards – each with many different uses.

**Declarations** We often use pattern matching to create declarations, especially when binding variables and extracting values from tuples or constructors.

\[
\text{val } \langle\text{pattern}\rangle = \langle\text{expression}\rangle
\]

We use `wildcards` when we have no need to store a value.

**Casing and Clauses** We can pattern match on constants and constructors to consider different patterns in multiple clauses of a `case` expression or function.

\[
\text{case } \langle\text{expression}\rangle \text{ of}
\quad \langle\text{pattern}\rangle \Rightarrow \langle\text{expression}\rangle
\quad | \langle\text{pattern}\rangle \Rightarrow \langle\text{expression}\rangle
\quad \text{fn } \langle\text{pattern}\rangle \Rightarrow \langle\text{expression}\rangle
\quad | \langle\text{pattern}\rangle \Rightarrow \langle\text{expression}\rangle
\quad \text{fun } \langle\text{identifier}\rangle \langle\text{pattern}\rangle = \langle\text{expression}\rangle
\quad | \langle\text{identifier}\rangle \langle\text{pattern}\rangle = \langle\text{expression}\rangle
\]

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

**Testing** To test code, we can pattern match on constants.
1.2 Thinking Recursively

We can break down recursion into two main parts. We have the base case(s), and the recursive case(s). When constructing recursive functions, we consider the recursive call and assume that the function works on a smaller input, then use that to construct the correct function behavior.

1.3 Induction Proofs

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

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: 

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 $\mathbb{N}$:

Let’s prove a lemma about the following piece of code:

```haskell
fun repeat (0 : int, k : int) : int list = []
| repeat (n : int, k : int) : int list = k :: repeat (n - 1, k)
```

This function takes $(n,k)$ and returns a list of $n$ copies of $k$. You’ll prove part of the correctness of this function on your homework, but we’ll prove that it evaluates to a value for all $k$ and all nonnegative $n$.

**Theorem.** For all natural numbers $n$ and all values $k : \text{int}$, $\text{repeat} \ (n,k)$ evaluates to a value.

**Proof.** We prove by induction on $n$. □
2 Recursion on the Naturals

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

The bodies of the first of these functions will follow the basic pattern of recursion that we discussed in lecture. They will consist of a case expression on the argument that has two branches. The first branch will specify the base case when the argument is zero. The second branch will specify the induction case when the argument is greater than zero. The induction case will include a recursive application of the function to an argument that is one less.

- 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 1.

To illustrate this structure, 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.

\[
\text{summ } n \equiv \sum_{i=0}^{n} i
\]

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.

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

Task 2.

Write a few test cases based on your code for summ. Try to consider different cases and cover them!
3 Another Pattern of Recursion

3.1 Fairly WhatParents

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 \( \text{evenP} : \text{int} \rightarrow \text{bool} \) which takes a natural number \( n \) and evaluates to either \( \text{true} \) if \( n \) is even, and \( \text{false} \) otherwise. However, we’re in a recursive mood, so we want to define \( \text{evenP} \) 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 \texttt{case} expression in the body of \( \text{evenP} \) 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:

\[
\text{fun evenP (x : int) : bool =}
\text{case x of}
\text{ 0 => true}
\text{| 1 => false}
\text{| _ => evenP (x - 2)}
\]

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

\textbf{Task 3.}

Write a few test cases based on your code for \( \text{evenP} \). Try to consider different cases and cover them!

We will now define the \( \text{oddP} \) function using this pattern.

\textbf{Task 4.}

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

\[
\text{oddP : int -> bool}
\]

\texttt{REQUIRES:} \( n \geq 0 \)

\texttt{ENSURES:} \( \text{oddP n} \Rightarrow \text{true} \) if \( n \) is odd and evaluates to \( \text{false} \) otherwise.

\textbf{Constraint: Do not call evenP or mod in the definition of oddP.}

\textbf{Task 5.}
Write a few test cases based on your code for evenP. Try to consider different cases and cover them!

**Task 6.**

Write a proof by induction on the natural numbers that, for all natural numbers n,

\[ \text{oddP } n \equiv \text{not (evenP } n) \]

where \(\text{not : bool} \rightarrow \text{bool} \) is defined as:

\[
\text{fun not (true : bool) : bool = false} \\
| \text{not (false : bool) : bool = true}
\]

Pay particular attention to how the pattern of recursion you used for oddP dictates the pattern of induction you need to prove this.

### 3.2 The Bees Threes

Satisfied with our ability to check even- and odd-ness, you ponder whether or not it would be possible to check divisibility by three in a similar way.

Next, you will define a function \(\text{divisibleByThree} : \text{int} \rightarrow \text{bool}\) such that for all \(n \geq 0\)
\(\text{divisibleByThree } n \) evaluates to

**Task 7.**

\[
\text{divisibleByThree} : \text{int} \rightarrow \text{bool} \\
\text{REQUIRES: } n \geq 0 \\
\text{ENSURES: } \text{divisibleByThree } n \iff \text{true} \text{ if } n \text{ is a multiple of 3 and to false otherwise.}
\]

**Constraint:** Do not use the SML \texttt{mod} operator for this task.

**Hint:** You will need a new pattern of recursion to define this function. Explain the pattern.
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 8.

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

```sml
GCD : int * int -> int
REQUIRES: m \geq 0, n \geq 0
ENSURES: GCD (m, n) \Rightarrow the GCD of m and n
```

Task 9.

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

Task 10.

Write a few test cases based on your code for \texttt{GCD}. Try to consider different cases and cover them!
5 Divide and... Mod

Suppose we wanted to implement division in SML. While you’ll do addition and multiplication on the homework, we’ll do division in lab. One thing you’ll notice is that division is a little bit trickier - 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: subtract denom from num until num is less than denom, at which point num is the remainder, and the number of total subtractions is the quotient. (Note that this is somewhat dual to multiplication!)

**Task 11.**

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

```sml
divmod : int * int -> int * int
REQUIRES: n ≥ 0, d > 0
ENSURES: divmod (n, d) ⇒ (q,r) such that qd + r = n and 0 ≤ r < d
```

Recall that when applied to an argument that does not meet the precondition, your implementation may have any behavior you like.

**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.