1 Intro to ML

1.1 Basic types

Need ; after expression

```
- 42
= ;
val it = 42 : int
- 7+1;
val it = 8 : int
```

Can reference `it`

```
- it+2;
val it = 10 : int
- if it > 100 then "big" else "small";
val it = "small" : string
```

Different types for each branch. What will happen?

```
if it > 100 then "big" else 0;
```

The type checker will complain that the two branches have different types, one is string and the other is int

If with then but no else. What will happen?

```
if 100 > 1 then "big";
```

There is not if-then in ML; must use if-then-else instead.

Mixing types. What will happen? What is this called?

```
10 + 2.5
```

In many programming languages, the int will be promoted to a float before the addition is performed, and the result is 12.5. In ML, this type coercion (or type promotion) does not happen. Instead, the type checker will reject this expression.

`div` for integers, '`/` for reals

```
- 10 div 2;
val it = 5 : int
- 10.0 / 2.0;
val it = 5.0 : real
```

1.1.1 Booleans

```
1=1;
val it = true : bool
1=2;
val it = false : bool
```

Checking for equality on reals. What will happen?

```
1.0=1.0;
```

Cannot check equality of reals in ML.

Boolean connectives are a bit weird

```
- false andalso 10 > 1;
val it = false : bool
- false orelse 10 > 1;
val it = true : bool
```

1.1.2 Strings

String concatenation

```
"University " ^ "of" ^ " Oslo"
```

1.1.3 Unit or Singleton type

- What does the type `unit` mean?
- What is it used for?
- What is its relation to the `zero` or `bottom` type?

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

https://en.wikipedia.org/wiki/Unit_type

https://en.wikipedia.org/wiki/Bottom_type

The boolean type is inhabited by two values: `true` and `false`. The singleton type is inhabited by a single value; for that reason, the type name and the element name are usually the same. In terms of notation, the singleton type is often represented in a different font from the value.

Several computer programming languages provide a unit type to specify the result type of a function with the sole purpose of causing a side effect.

Although the analogy does not hold 100%, an example of what can be considered a singleton type is `void` in C.

Singleton types are not to be confused with the bottom type. Bottom or zero type is inhabited by no value. Bottom is used to represent the return type of a function that does not return a value: for instance, one which loops forever, signals an exception, or exits.

1.2 Compound types

1.2.1 Tuples

Can you explain the `*` below?

```
- (1,2,3);
val it = (1,2,3) : int * int * int
```

The tuple is a product type. If A and B are types, then A * B is the type composed of any element from A followed by any element from B.

Indexing, starts with 1

```
- #1("apple","banana","carrot");
val it = (1,2,3) : int * int * int
```

1.2.2 Records

Tuples with names

Order does not matter

```
- {name="Theo", age=21};
val it = {age=21,name="Theo"} : {age:int, name:string}
- {name="Theo", age=21} = {age=21, name="Theo"};
val it = true : bool
```

1.2.3 Lists

Can you explain the `'a list` below?

```
- nil;
val it = [] : 'a list
- 1 :: nil;
val it = [1] : int list
```

This is an example of parametric polymorphism. As opposed to having different definition of lists, for example ListInt, ListBool etc, we have a generic List that is polymorphic (can take different shapes) and parametric (the shape is determined by a parameter).

Two ways of constructing lists

```
- 1 :: 2 :: nil;
- [1,2];
- 1 :: 2 :: nil;
```

List concatenation

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

1.3 Value declaration

Associates a `value` with a `pattern`

```
val <pattern> = <exp>;
```

For example:

```
- val x = 5;
val x = 5 : int
- x;
val x = 5 : int
- x + 2;
val it = 7 : int
```

A pattern can be an identifier, tuple, list, record, or a declared data-type

```
<pattern> ::= <id> | <tuple> | <cons> | <record> | <constr>
<constr> ::= <id>(<pattern>, ..., <pattern>)
```

For example:

```
- val l = [1,2,3,4];
val l = [1,2,3,4] : int list
```

So far in the examples above, the `<pattern>` has been `<id>`. Here are different examples:

```
- val tup = ("apple", "banana");
val tup = ("apple","banana") : string * string
- val (x,y) = tup;
val x = "apple" : string
val y = "banana" : string
```

A variable is not allowed to occur more than once in a pattern:

```
- val (x,x) = tup;
```

1.3.1 Local declarations and the `let` construct

```
- let val x = 2+3 in x*10 end;
val it = 50 : int
```

1.4 Functions and pattern matching

1.4.1 Anonymous function or lambda

Below we declare a function that takes an argument `x` and adds `2` to it.

```
- fn x => x + 2;
val it = fn : int -> int
- it 3;
val it = 5 : int
```

Here, in one step, we create the function and call on an argument (the number 3):

```
- (fn x => x + 2) 3;
val it = 5 : int
```

1.4.2 Named functions

Here we declare an anonymous function and then bind it to the name `f`:

```
- val f = fn x => x + 2;
val f = fn : int -> int
- f 2;
val it = 4 : int
```

We could do the same as the example above using `fun`, which is a short-hand for named functions:

```
- fun f x = x + 2;
val f = fn : int -> int
- f 2;
val it = 4 : int
```

1.4.3 Function as an Algebraic Data Type

- What is an Algebraic Data Type (ADT)
- Examples of Algebraic Data Types
- Why the name "algebraic"?
- Not to be confused with Abstract Data Type (ADT)

The function type is also called exponential type. A function from type A to type B can map an element of A to any element of B; there are B^A possible mappings, thus the name "exponential."

Other algebraic types are product types (i.e. tuples) and sum types (i.e. variants or disjoint unions).

The name "algebraic data types" comes from the fact that these types obey algebraic laws. For example, we can show that the type `A -> B -> C` is isomorphic `A * B -> C`.

`A -> (B -> B)` as $(C^B)^A = C^{B*A}$

`A * B -> C` as C^{B*A}

These two functions are isomorphic:

```
- fun f (x,y) = x+y;
val f = fn : int * int -> int
- fun f x y = x + y;
val f = fn : int -> int -> int
```

Other identities that work on types `A`, `B`, and `C` are:

$$A + B \simeq B + A$$
$$A * B \simeq B * A$$

$$(A + B) + C \simeq A + (B + C)$$
$$(A * B) * C \simeq A * (B * C)$$

$$A * (B + C) \simeq A * B + A * C$$
$$A- > (B- > C) \simeq A * B- > C$$
$$(A- > C) * (B- > C) \simeq (A + B)- > C$$

These are given here simply for your own curiosity.

1.4.4 Currying and partial evaluation

Below we define a function `f` that takes an `int` `x` and another `y` and adds them. Of course, calling `f` on `2` and `3` returns `5.` But what happens when we call `f` on `2` only?

```
- fun f x y = x + y;
val f = fn : int -> int -> int
- f 2;
val it = fn : int -> int
- it 3;
val it = 5 : int
```

We can interpret `f` as a function that takes an `int` and returns a function from `int` to `int`. So, calling `f` with argument `2` returns a function that takes an `int` and adds `2` to it. This is called **partial evaluation.**

The function `fun f x y = x + y;` is said to be the curry'ed version of the function `fun f2 (x,y) = x + y;`. Note that currying allows for partial evaluation; partial evaluation is not possible in `f2`.

1.4.5 Multiple clause functions

```
- fun f(x,0) = x
    | f(0,y) = y
    | f(x,y) = x+y;
```

1.4.5.1 Case construct

A function `d` for division of `int`s and `real`s. Notice currying again.

```
- datatype num = I of int | R of real;
- I(10);
- R(2.0);
- fun d x y =
    case x of I(n1) => ( case y of I(n2) => I(n1 div n2) )
            | R(n1) => ( case y of R(n2) => R(n1 / n2) );

- d(I(10))(I(2));
val it = I 5 : num
- d(R(10.0))(R(2.0));
val it = R 5.0 : num
```

What happens if we mix them?

```
- d(R(10.0))(I(2));
```

We can write a division function that promotes ints to reals when they get mixed together:

```
- fun d x y =
    case x of I(n1) => ( case y of I(n2) => I(n1 div n2)
                                 | R(n2) => R( (real(n1) / n2 ) ) )
             | R(n1) => ( case y of R(n2) => R(n1 / n2)
                                 | I(n2) => R(n1 / real(n2) ) );
```

1.4.6 Recursion

We'll look at recursion after data type declarations

1.5 Data-type declaration

```
- datatype color = Red | Yellow | Blue;
```

`Red` `Yellow` and `Blue` are called **constructors.** They construct an element of type color

```
- datatype color = Red | Yellow | Blue;
datatype color = Blue | Red | Yellow
- Blue;
val it = Blue : color
```

Constructors can take arguments

```
- type name = string;
- datatype school = UiO | NTNU;
- datatype faculty = MatNat | Engineering | Humanities;
- datatype student = BS of name | MS of name*school | PhD of name*faculty;
```

Constructors don't *do anything* to their arguments other than "tag" them so the arguments can be distinguished via pattern matching.

```
- PhD("Daniel",MatNat);
val it = PhD ("Daniel",MatNat) : student
- val PhD(a,b) = it;
val a = "Daniel" : name
val b = MatNat : faculty
```

If `PhD("Daniel",MatNat)` constructs a PhD student, then we can interpret

`val PhD(a,b) = PhD("Daniel",MatNat)`

as "destructing" the student into a value `a` and a value `b`.

Finally, note that `datatype` is used to define a new type while `type` gives a “new name” to a type.

```
- type name = string;
type name = string
```

1.6 Functions (continued): Recursion

(Example from *The little MLer*)

```
- datatype shish_kebab =
  Skewer
  | Onion of shish_kebab
  | Lamb of shish_kebab
  | Tomato of shish_kebab;

- Skewer;
- Onion(Skewer);
- Lamb(Onion(Skewer));
```

```
- fun is_vegetarian(Skewer)    = true
    | is_vegetarian(Onion(x))  = is_vegetarian(x)
    | is_vegetarian(Lamb(x))   = false
    | is_vegetarian(Tomato(x)) = is_vegetarian(x);
```

```
- is_vegetarian( Onion(Tomato(Skewer)) )
- is_vegetarian( Onion(Tomato(Lamb(Skewer))) )
```