Excercise 1 (6.1 in Mitchell's book)

A)

What is the type of the following function and why?

```
fun a(x,y) = x+2*y;
```

The + and * operators either take in only reals or only integers

Since 2 is an integer, and it's used with both x and y, the answer must be int * int -> int

Excercise 1 (6.1 in Mitchell's book)

B)

What is the type of the following function and why?

```
fun b(x,y) = x+y/2.0;
```

The + and * operators either take in only reals or only integers

Since 2.0 is a real, and it's used with both x and y, the answer must be real * real -> real

Excercise 1 (6.1 in Mitchell's book)

C)

What is the type of the following function and why?

```
fun c(f) = fn y => f(y);
```

y is an argument of some type, let's call it 'a

Since f takes y as an argument, it's type must be 'a -> 'b (both types unknown)

c takes f as an argument and, which is applied to y. C's type is therefore ('a -> 'b) -> 'a

The complete type is therefore ('a -> 'b) -> 'a -> 'b

Excercise 1 (6.1 in Mitchell's book)

D)

What is the type of the following function and why?

```
fun d(f,x) = f(f(x));
```

x is an argument of some type 'a

f is applied to x, and it's also applied to the return value of itself, so 'a -> 'a

d takes a pair consisting of f and x, argument type is therefore ('a -> 'a) * 'a

The complete type is therefore ('a -> 'a) * 'a -> 'a

Excercise 1 (6.1 in Mitchell's book)

E)

What is the type of the following function and why?

```
fun e(x,y,b) = if b(y) then x else y;
```

x and y has to be of the same type, since they can both be returned, so 'a

b is a function that takes the type of y as an argument, so 'a as argument

It is used in the if clause, and such must return a boolean

The complete type is therefore 'a * 'a * ('a -> bool) -> 'a

Excercise 2 (6.4 in Mitchell's book)

A)

```
fun Y f x = f (Y f) x;
val Y = fn : (('a -> 'b) -> 'a -> 'b) -> 'a -> 'b

fun F f x = if x=0 then 1 else x*f(x-1);
val F = fn : (int -> int) -> int -> int

val factorial = Y F;
val factorial = fn : int -> int
```

Y is a function that takes a function as its first parameter, which applied to an argument of type 'a gives something of type 'b

Example of use: factorial(5); [it = 120]

Excercise 2 (6.4 in Mitchell's book)

```
fun F f x = if x=0 then 1 else x*f(x-1);
val F = fn : (int -> int) -> int -> int
```

b) Apply F to the identity function (fn x => x) and the following arguments: 0, 1, 2, 3. What is supposed to be the result? Why? Is F the factorial function? Explain.

No, F is not a recursive function and as such cannot be the factorial function (it calls f, not F)

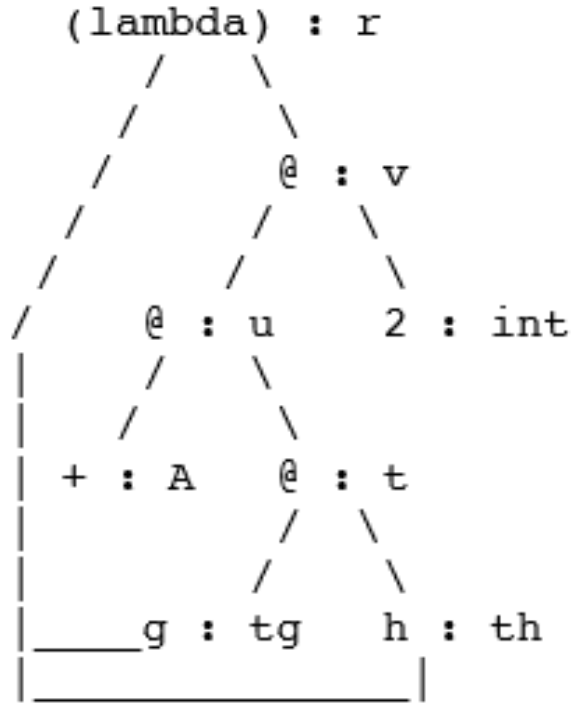
F (fn x => x) n gives us n*(n-1)

```
- F (fn x => x) 0;
val it = 1 : int
- F (fn x => x) 1;
val it = 0 : int
- F (fn x => x) 2;
val it = 2 : int
- F (fn x => x) 3;
val it = 6 : int
But:
- F (fn x => x) 4;
val it = 12 : int
```

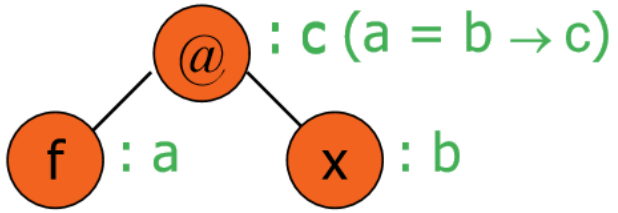
Excercise 3 (6.5 in Mitchell's book)

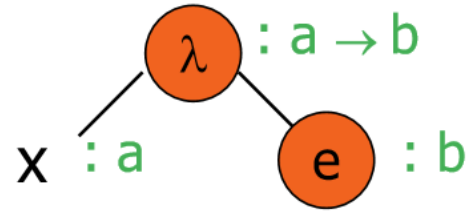
```
fun f(g,h) = g(h) + 2;
```

1. The following shows the assignment of type to nodes:



where A =def= int->int->-int.





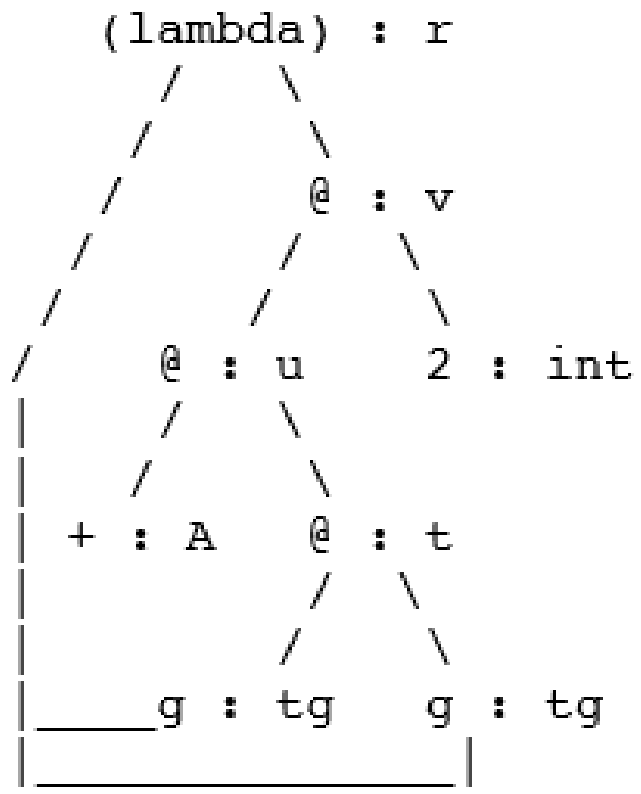
1. tg = th -> t
2. int -> (int -> int) = t -> u
3. u = int -> v
4. r = (tg * th) -> v

5. From 2 we know t = int and u = int -> int
6. From 3 and 5, we know v = int
7. From 1, 4, 5 and 6, we know r = (th->int) * th -> int

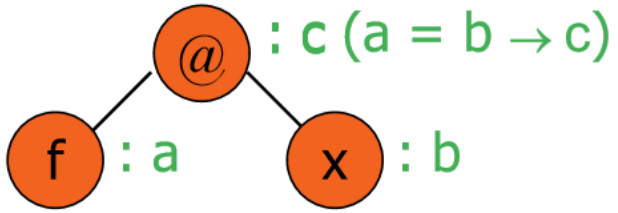
8. Substitute th with an unknown type variable 'a
9. Ending up with r = ('a -> int) * 'a -> int

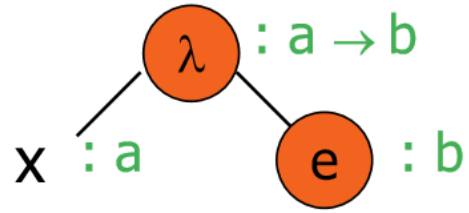
Excercise 4 (6.6 in Mitchell's book)

```
fun f(g) = g(g) + 2;
```



where A =def= int->int->-int.





1 tg = tg -> t

2. int -> (int -> int) = t -> u

3. u = int -> v

4. r = tg -> v

5. From 2 we know t = int and u = int -> int

6. From 3 and 5, we know v = int

7. From 4 and 6, we know that r = tg -> int

8. Looking at 1, we need to know tg to get to know tg

9. Type inference algorithm cannot give appropriate type, error

Excercise 5 (6.7 in Mitchell's book)

```
fun append(nil, l) = l
  | append(x::l,m)=append(l,m);

append: ('a list * 'b) => 'b
```

The first argument has to be a list of some type because of nil and x::l

The second argument the compiler knows nothing about, no operations, or anything else to indicate its type. So it's given the type 'b

The return type of append has to be of type 'b, and by this we can see that it will not guarantee a list to be returned

Calling append with a list and some value of type 'b, it will just return the value

Excercise 6

A)

Answer:
Context-independent

Because:
All combinations of char and string are defined

There are two kinds of overloading of functions/operators:
- context-independent: overloading only done on parameters to function or type of operands for an operator;
- context-dependent: which operator to use depends also on the type of the result.

Hence, with context-independent overloading, the function to be called is solely based upon the type of the actual parameter. With context-dependent overloading, the function to be called may not be identified by the type of the parameters. The context must be taken into account to identify the function to be called.

a) Consider the operator ++ (concatenation) to be overloaded, with types

```
Char x Char -> String
Char x String -> String
String x Char -> String
String x String -> String
```

Is this overloading context-independent or context-dependent? Have a look at the following examples:

```
c ++ s
(s ++ c) ++ c
s ++ (c ++ c)
```

where c is a Char variable and s is a String variable, and identify for each ++ what kind of concatenation it is.

```
c++s: C*S->S
(s++c)++c: (S*C->S)*C->S
s++(c++c): S*(C*C->S)->S
```

Excercise 6

B)

b) Define in ML four infix operators ++ with the types specified in item a) (in the same order as they are defined). Define then the following four values:

```
val c1 = #"a";
val c2 = #"h";
val s1 = "bcd";
val s2 = "efg";
```

Before executing, guess what would be the result of the following:

Answer:

```
infix ++;

fun c ++ s = str(c)^s;
fun c1 ++ c2 = str(c1)^str(c2);
fun s ++ c = s^str(c);
fun s1 ++ s2 = s1^s2;

c1 ++ c2;
s1 ++ c2;
c1 ++ s1;
s1 ++ s2;
```

c1 ++ c2; => ah

s1 ++ c2; => bcdh

c1 ++ s1; => abcd

s1 ++ s2; => bcdefg

Excercise 7

A)

Would overloading for procedures be context-dependent or context-independent?

Remember that procedures are a set of operations without results

Context-independent, because as stated above, they do not return results

Excercise 7

b) Assume that the operator / is overloaded, with types

```
int * int -> int
```

and

```
int * int -> double.
```

Assume further that we have an overloaded proper procedure "write",
the argument of which may be either int or double.

Give examples of procedure calls where the procedure to be called cannot be defined uniquely.

write(5/9)

Excercise 8

A)

Consider the following parametric function in C++ like syntax:

```
template <typename T>
T second(T x, T y) {return y}
```

It is parameterised by the type parameter T, and it returns the value of the second parameter. It is called in this way:

```
int i,j;
second<int>(i,j)
```

The <**int**> is not really needed, as the compiler may infer that the type shall be int, but it is included here for clarity reasons.

a) The body of the function does not really use any properties of the type T. Why is it so that C++ still has to generate different code for different actual types?

It needs to allocate space for the variables on the activation record
Different types can have different space requirements

Excercise 8

b) Assume that we had the following version of "second":

```
template <class T>
T second(T *x, T *y) {return y}
```

Why would it be possible to use the same code for "second" for different actual classes for T?

Pointers take up the same amount of space regardless of the class they are pointing to