Exercise

Define a function

```
power : int * int -> int
```

so that, for $m \geq 0$ $power(n, m) = n^m$ holds. Assume that 0^0 is defined as 1.

Solution

```
fun power(n,m) = if m=0 then 1 else n * power(n,m-1);

val power = fn: int * int -> int

> power (0,0);
val it = 1: int
> power (2,4);
val it = 16: int
> power (10,3);
val it = 1000: int
>
```

Controlling the environment

- ML uses static scoping
- Non-local symbols are resolved by reference to the environment where they were defined
- How do we define blocks?
  - We've already seen one way
  - In a declaration

    ```
    val f = fn v => 2 * v;
    ```

    `v` is a local variable
  - Example

    ```
    val v = 5;
    val f = fn v => 2 * v;
    ```

    what is the result of `f 2`?

Example

```
> val v = 5;
val v = 5: int
> val f = fn v => 2 * v;
val f = fn: int -> int
> f 2;
val it = 4: int
>
```

Environment

- Non-local symbols are searched for in the active environment when the function is evaluated
- Example

  ```
  val f = fn x => x+y;
  ```

  and

  ```
  val y=2;
  val f = fn x => x+y;
  ```

Environment

- Nested can also be done using `let` and `local`
- `let`
  - Syntax

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

  - The result of this expression is the result of the expression at the end
  - Bindings in `declaration` are valid only until the `end`

Tail recursion

- Tail recursion: Recursion that only has a recursive call at the end
- Tail recursive

```
fun fact n = if n = 0 then 1 else n * fact (n - 1);
```

- Not tail recursive

```
fun fact n = if n = 0 then 1 else fact (n - 1) * n;
```

Example

```
> fun fact n = if n = 0 then 1 else n * fact (n - 1);
val fact = fn: int -> int
> fact 10;
val it = 3628800: int
> fun fact n = if n = 0 then 1 else fact (n - 1) * n;
val fact = fn: int -> int
> fact 10;
val it = 3628800: int
```

Converting recursion to tail recursion

- Use a second parameter to store partial results

```
val rec fact_tr = fn n => fn res => if n=0 then res else fact_tr (n-1)(n*res);
```

- This function takes two arguments. We can then define

```
val fact = fn n => fact_tr n 1;
```

Example

```
> val rec fact_tr = fn n => fn res => if n=0 then res else fact_tr (n-1)(n*res);
val fact_tr = fn: int -> int -> int
> val fact = fn n => fact_tr n 1;
val fact = fn: int -> int
> fact 5;
val it = 120: int
> fact_tr 4 5;
val it = 120: int
> fact_tr 3 5*4;
val it = 120: int
> fact_tr 2 5*4*3;
val it = 120: int
```

Using `let`

- Problem: `fact_tr` is visible in the global environment. How can we prevent that?
- Note that `fact_tr` is only used in the final call
- Use `let` to make the declaration local

```
val fact = fn n =>
    let
     val rec fact_tr = fn n => fn res =>
         if n = 0 then
             res
         else
              fact_tr (n - 1) (n * res)
     in
        fact_tr n 1
end;
```

Example

```
> val fact = fn n =>
     let
     val rec fact_tr = fn n => fn res =>
         if n = 0 then
             res
         else
              fact_tr (n - 1) (n * res)
       in
        fact_tr n 1
end;
val fact = fn: int -> int
> fact 5;
val it = 120: int
> fact_tr 5 1;
poly: : error: Value or constructor (fact_tr) has not been declared
Found near fact_tr 5 1
Static Errors
```

Using `local`

- Problem: Define a function on positive integers (`unsigned int`)
- We define `integer_f` that calls `f` if the argument is positive and returns -1 otherwise. We illustrate this for the function $f(x) = 5 + x$

```
local
   val integer_f = fn n => 5 + n
in
   val f = fn n => if n<0 then ~1 else integer_f n
end;
```

`local`

- We can also use `local` to hide the function `fact_tr`

```
local
  val rec fact_tr = fn n => fn res =>
         if n = 0 then
             res
         else
              fact_tr (n - 1) (n * res)
in
  val fact = fn n => fact_tr n 1
end;
```

Functions on functions

- In functional languages, functions can be denotable objects, and can be the results of functions
- We define the approximate derivative of a function

```
val derivative1 = fn (f, x) => (f(x) - f(x-0.001))/0.001;
```

  Note how ML has derived the types

- Now, we define a function that returns the derivative of `f`

```
val derivative2 = fn f  => (fn x => (f(x) - f(x-0.001)) / 0.001);
```

Example

```
val derivative1 = fn (f, x) => (f(x) - f(x-0.001))/0.001;
val derivative1 = fn: (real -> real) * real -> real

val derivative2 = fn f  => (fn x => (f(x) - f(x-0.001)) / 0.001);
val derivative2 = fn: (real -> real) -> real -> real
```

Currying

- How did we get from `derivative1` to `derivative2`?
- We convert a function of two variables x and y to one that takes one parameter x and returns a function of y
- This converts a function $f : R \times R \rightarrow R$ to a function $f_c : R \rightarrow R \times R$
- Example. The function

  ```
  val sum =  fn (x,y) =>  x + y;
  ```

  can be written, via currying, as

  ```
  val sum_c = fn x => (fn y => x + y);
  ```

Currying

- $f(x, y) = x^2 + y^2$ and the curried version $f_c()$
- f has domain $R \times R$ and range R, while f_c has domain R and range $R \times R$
- ML definitions

```
val f = fn (x,y) =>  x * x + y * y;
val f_c = fn x =>  (fn y => x * x + y * y);
```

Example

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

val f_c = fn x =>  (fn y => x * x + y * y);
val f_c = fn: int -> int -> int
```

Currying

- ML has an abbreviated syntax for currying
- The command

  ```
  fun f a b =exp;
  ```

  is equivalent to

  ```
  val rec f = fn a => fn b = exp;
  ```

- We can then write

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

  and

  ```
  fun derivative2 f x = (f(x)-f(x-0.001))/0.001;
  ```

Example

```
fun f(x,y)=x*2 + y*2;
val f = fn: int * int -> int

fun derivative2 f x = (f(x)-f(x-0.001))/0.001;
val derivative2 = fn: (real -> real) -> real -> real
```

Exercise

The positive integer square root of a non-negative integer is a function `introot` such that `introot m` is the largest integer `n` such that `n*n` is less than or equal to m. Define this function in ML.

Solution

```
fun introot m = let fun aux(k,m) = if k*k > m then k-1 else aux(k+1,m)
                    in aux(0,m)
end;

val introot = fn: int -> int



> introot 15;
val it = 3: int
> introot 16;
val it = 4: int
>
```

Exercise

In ML, as in other languages, the if-then-else construct is non-strict, i.e. only one of the branches is evaluated, depending on the the result of the test. What would be the consequences for recursive definitions if the if-then- else were strict (meaning that first both the branches are evaluated, and then one of the results is picked, depending on the result of the test), and pattern-matching were not available?

Solution

It would not be possible to define recursive functions anymore. The recursive definition of a function f has typically the following scheme

```
fun f x = if x=0 then < expression > else
                  < expression_with_recursive_call_of f >
```

If the if-then-else were strict, the evaluation of the recursive call would always be required, even when $x = 0$, and would always loop

If definition by pattern-matching were available, we could give the alternative definition

```
fun f 0 = < expression >
     | f x = < expression_with_recursive_call_of f >
```

Factorial

```
val rec fact_tr = fn n => fn res => if n=0 then res else fact_tr (n-1)(n*res);
```

- `fact_tr` uses the second argument to "accumulate" the result
- The multiplication by n is *before* the recursive call, and not after
- This means that the value of n does not have to be saved

Example

- Evaluation of `fact 4` is as follows
  - `fact_tr 4 1`
  - `1 * fact_tr 3 4`
  - `1 * fact_tr 2 12`
  - `1 * fact_tr 1 24`
  - `1 * fact_tr 0 24`
  - `24`

Programming paradigms

- A style/paradigm of programming
  - There are languages (ML) that make it easier, or force us, to use a functional style
  - But programs can be written in a functional style even with imperative languages such as C
  - We usually think of a program as a sequence of instructions to be executed in a specifc order

Euclid's algorithm

- Find the Greatest Common Divisor (gcd) of 2 integers
- Given two integers a and b, if $b = 0$ then a is the gcd. Otherwise take the gcd of b and the remainder after dividing a by b
- This can be converted directly to an imperative language such as C

GCD

```
unsigned int gcd(unsigned int a, unsigned int b) {
while (b != 0)
{ unsigned int tmp;
  tmp = b;
  b=a%b;
  a = tmp;
 }
return a ;
}
```

GCD

- Why does this work?
  - The gcd of a and 0 is clearly a
  - The gcd of a and $b \neq 0$ is equal to the gcd of a and $a\%b$ (provable by induction)
- Therefore

$$\gcd(a,b) = \begin{cases} a & b = 0 \\ \gcd(b, a\%b), & \text{otherwise} \end{cases}$$

gcd

- This gives us the following implementation

```
unsigned int gcd(unsigned int a, unsigned int b)
{
    if (b == 0) {
       return a ;
    }
 return gcd(b, a % b);
}
```

Functional implementation

```
unsigned int gcd(unsigned int a, unsigned int b)
{
    return (b == 0) ? a : gcd(b, a % b);
}
```

Functional style

- The imperative implementation modifies the values of various variables, while the functional version does not modify any variables
- The imperative version uses a `while` cycle, while the functional version uses recursion, with `b==0` as a test to terminate the recursion
- Note that the imperative approach uses a test using a variable that is changed during the computation. In the absence of such variables, another approach is needed

ML

```
val rec  gcd = fn(a,b) => if b=0 then a else   gcd(b,a-b*(a div b));
val gcd = fn: int * int -> int

> gcd (3,6);
val it = 3: int
> gcd (6,3);
val it = 3: int
> gcd (6,15);
val it = 3: int
```

Mathematical background

- Based on "pure functions"
- Function $f \subseteq D \times C$; D domain and C codomain
- $(x, y) \in f$ written $f(x) = y$
  - f always associates the same $y \in C$ to each $x \in D$
  - The only effect of f on x is to calculate y

Consequences

- No assigment, so no modifiable variables
- Therefore, recursion replaces cycles
- Similarly, `if` is replaced by the "arithmetic if" that evaluates one of two expressions, depending on the condition

Consequences

- If we evaluate a function twice, with the same arguments, we should get the same result
- This is not true with imperative languages

```
int f(int v) {
  static int acc ;
  acc = acc + 1;
  return v + acc; }
```

- Calls $f(2)$, $f(2)$, $f(2)$ give 2, 3, and 4

Recursion and iteration

- The functional equivalent of iteration is recursion
- This is used to define an "entity" based on itself
- For example, $f(n)$ is defined based on other values of f, typically $f(n-1)$

Recursion

- Recursive definition typically by cases
- Base case (inductive basis) and one or more inductive clauses
- A function $f : N \to X$ is defined using a function $g : N \times X \to X$
  - Define $f(1) = a$
  - $f(n+1) = g(n, f(n))$
- A function can be defined recursively if it has domain N (no restrictions on the codomain)

Typical example of recursion: Factorial

```
unsigned int fact (unsigned int n)
   {
   if (n == 0) { return 1 ;
   }
  return n * fact (n - 1);
}
```

Functional implementation

```
unsigned int fact (unsigned int n) {
     return (n == 0) ? 1 : n * fact(n - 1); }
```

- Evaluation: `fact(4) = (4==0)?1:4*fact(4-1) = 4*fact(3)`
- This is equal to `4*((3==0)?1:3 * fact(3-1)) = 4*(3* fact(2))`

Evaluation

- The system needs to
  - Search the environment for the definition of the function/parameters
  - Textual substitution
  - Arithmetic calculation

Substitution

- Evaluation by substitution
  - Only works in the absence of side effects
  - Absence of variables might make writing programs harder, but this is often due to habit rather than inherent in the programming style itself
  - Next example: Towers of Hanoi, which is much simpler to do in the functional paradigm

Towers of Hanoi

- A non-recursive solution is difficult
- With recursion, it is easy
- Move N disks from stack a to stack b:
  - Move $N-1$ disks from stack a to stack c
  - Move the remaining disk (the largest) from stack a to stack b
  - Move the $N-1$ disks from stack c to stack b
- The first and third steps are recursive solutions to the problem with $N-1$ disks