Exercise sheet 5
for Monday 28 February and Wednesday 2 March 2011

Last update 2011-02-28

Hand in your solutions by sending a single zip file to sestoft@itu.dk by Friday 18 March 2011. The file name must be **`SASP-05-yourname.zip`**; for instance `SASP-05-OleHansen.zip`.

Either just briefly describe the necessary changes, with code fragments showing the code changes you have made; **or** concatenate the full edited files including easily searchable comments that clearly indicate what changes you have made and for which exercise.

The Scheme programming language

Getting started

Install a Scheme system, such as DrScheme or Jaffer's scm system. To check that everything works out of the box, start the Scheme system and enter this expression:

```
(define (fac n) (if (= n 0) 1 (* n (fac (- n 1)))))
```

and then compute

```
(fac 10)
```

The result should be 3628800. For kicks, try also `(fac 1000)`.

Each subproblem below can be solved by a function definition 4 to 7 lines long, but you are welcome to define auxiliary functions for clarity. Apart from the syntax and lack of pattern matching and type checking, this sort of programming is very close to Standard ML, F#, or Scala.

Exercise 5.1 Define a function `(prod xs)` that returns the product of the numbers in `xs`. It is reasonable to define the product of an empty list `()` to be 1.

Exercise 5.2 Define a function `(makelist n)` that returns the `n`-element list `(n ... 3 2 1)`.

Exercise 5.3 Define a function `(makelist n)` that returns the `n`-element list `(1 2 3 ... n)`.

This can be done by defining an auxiliary function `(makeaux i n)` that returns the list `(i ...n)` and then let `(makelist n)` call that auxiliary function. Note that the result of `(makeaux i n)` should be the empty list if `i>n`.

Exercise 5.4 The function `(filter p xs)` returns a list consisting of those elements of list `xs` for which predicate `p` is true.

Thus if `xs` is `'(2 3 5 6 8 9 10 12 15)` then `(filter (lambda (x) (> x 10)) xs)` should give `(12 15)` and `(filter (lambda (x) (= 0 (remainder x 4))) xs)` should give `(8 12)`.

Define function `(filter p xs)` and try it.

Exercise 5.5 More recursion. A list can contain a mixture of numbers and sublists, as in these two examples:

```
(define list1 '(1 (1 2) (1 2 3)))
(define list2 '(1 (1 (6 7) 2) (1 2 3)))
```

Write a function `(numbers mylist)` that counts the total number of numbers in `mylist`.

Hint: The predicate `(number? x)` is true if `x` is a number, not a list or symbol.

The result of `(numbers list1)` should be 6, and that of `(numbers list2)` should be 8.

Scheme and program generation

Exercise 5.6 The exponential function e^x can be approximated by the expression

```
1 + x/1 * (1 + x/2 * (1 + x/3 * (1 + x/4 * (1 + x/5 * (...)))))
```

Define a Scheme function `(makeexp i n)` that returns a Scheme *expression* containing the terms from 1+x/i to 1+x/n of the above expression, replacing ... at the end with 1. Note the similarity to Exercise 5.3.

- `(makeexp 1 0)` should be the expression `1`
- `(makeexp 1 1)` should be the expression `1 + x * 1/1 * 1`
- `(makeexp 1 2)` should be `1 + x * 1/1 * (1 + x * 1/2 * 1)`
- and so on.

Hint: In Scheme, multiplication may be applied to any number of arguments, so `x * y * z` can be written simply as `(* x y z)`.

Hence, in Scheme the three expressions shown above can be written like this:

```
1
(+ 1 (* x 1 (+ 1 0)))
(+ 1 (* x 1 (+ 1 (* x 0.5 (+ 1 0)))))
```

Check that `(makeexp 1 n)` produces these results for `n` = 0, 1, 2.

Then define `x` to be 2 and compute `(eval (makeexp 1 n))` for `n` being 1, 10 and 20. The correct value of e^2 is close to 7.38905609893065.

Exercise 5.7 Now define a Scheme function `(makemyexp n)` that, as a side effect, defines a function `myexp`. When `(myexp x)` is called, it must compute e^x using `n` terms of the above expansion.

Function `makemyexp` should use `(makeexp 1 n)` to do its job.

.Net runtime code generation

Getting started

Download example file RTCG2D2.cs. The example uses classes DynamicMethod and ILGenerator from the System.Reflection.Emit namespace. Using DynamicMethod one can generate a method which can subsequently be called as a delegate. Methods generated this way can also be collected by the garbage collector when no longer in use.

Exercise 5.8 Make a copy of the RTCG2D2.cs example and modify it so that it generates a method corresponding to this one:

```
public static double MyMethod1(double x) {
  Console.WriteLine("MyMethod1() was called");
  return x * 4.5;
}
```

Hint: The bytecode instruction for multiplication is `OpCodes.Mul`, from the System.Reflection.Emit namespace.

Note that you need a new delegate type D2D like this

```
public delegate double D2D(double x);
```

Check that the new method works by calling it with arguments 1.0 and 10.0.

Exercise 5.9 Modify the RTCG2D2.cs example so that it generates a method corresponding to this one:

```
public static double MyMethod2(double x, double y) {
  Console.WriteLine("MyMethod2() was called");
  return x * 4.5 + y;
}
```

You will need to change the generated method's signature. Moreover, either you must define a new delegate type DD2D like this

```
public delegate double DD2D(double x, double y);
```

or you may use the type instance `Func<double,double,double>` of generic delegate type `Func'3<A1,A2,R>`.

Exercise 5.10 Write a C# method

```
public static I2I MakeMultiplier(int c) { ... }
```

that takes as argument an integer `c`, and then generates and returns a delegate of type I2I corresponding to a method declared like this:

```
public static int MyMultiplier_c(int x) {
  return x * c;
}
```

where delegate type I2I is declared like this:

```
public delegate int I2I(int x);
```

Or you may use `Func<int,int>` instead of I2I.

Hint: The `MakeMultiplier` method must include everything needed to generate a `MyMultiplier_c` method. The generated method's return type and its parameter type should be `int`.

Exercise 5.11 The exponential function e^x can be computed by the expression

```
1 + x/1 * (1 + x/2 * (1 + x/3 * (1 + x/4 * (1 + x/5 * (...)))))
```

Write a C# method

```
public static D2D MakeExp(int n) { ... }
```

that takes as argument an integer `n` and uses runtime code generation to generate and return a delegate of type D2D corresponding to a method declared like this:

```
public static double MyExp_n(double x) {
  ... compute res as first n terms of the above product ...
  return res;
}
```

Hint (a): If you were to compute the term instead of generating code for it, you might do it like this:

```
double res = 1;
for (int i=n; i>0; i--)
  res = 1 + x / i * res;
return res;
```

The `generated code` should contain no for-loop and no manipulation of `i`, only the sequence of computations on `res` performed by the iterations of the loop body. The *generator*, on the other hand, is likely to contain a loop.

Hint (b): To push integer `i` as a double, use

```
ilg.Emit(OpCodes.Ldc_R8, (double)i);
```

Exercise 5.12 Write a C# method

```
public static D2D MakePower(int n) { ... }
```

that takes as argument an integer `n`, and uses runtime code generation to generate and return a delegate of type D2D corresponding to a method declared like this:

```
public static double MyPower_n(double x) {
  ... compute res as x to the n'th power ...
  return res;
}
```

Hint (a): If you were to compute the `n`'th power of `x` instead of generating code for it, you might do it like this:

```
double res = 1;
for (int i=0; i<n; i++)
  res = res * x;
return res;
```

The generated code should contain no for-loop and no manipulation of `n`, only the sequence of computations on res performed by the loop's body.

Hint (b): A much faster way of doing the same — time $O(\log(n))$ instead of time $O(n)$ — is this:

```
double res = 1;
while (n != 0) {
  if (n % 2 == 0) {
    x = x * x;
    n = n / 2;
  } else {
    res = res * x;
    n = n - 1;
  }
}
return res;
```

Again, the generated code should contain no loop and no manipulations of `n`; only the sequence of operations on `x` and `res` performed by the loop body.