

Exercises week 12

Monday 23 November 2009

NB: Disse opgaver er frivillige — der vil ikke være eksamensspørgsmål i Scheme og køretidskodegenerering i F2009.

Goal of the exercises

The main goal of this week's exercises is to get a taste of programming in Scheme, to understand program generation using Scheme, and to understand runtime bytecode generation in .NET using C#.

Do this first

Install a Scheme system, such as PLT Scheme (for Windows, MacOS or Linux) or Jaffer's scm system (primarily for Linux). 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 computing also `(fac 1000)`.

Exercise 12.1 Each subproblem in this exercise can be solved by a Scheme function definition 4 to 7 lines long, but you are welcome to define auxiliary functions for clarity. Apart from the new syntax and lack of pattern matching and type checking, Scheme programming is rather close to F#.

(i) 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.

(ii) Define a function `(makelist1 n)` that returns the `n`-element list `(n ... 3 2 1)`.

(iii) Define a function `(makelist2 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 calling that auxiliary function from `(makelist2 n)`. Note that the result of `(makeaux i n)` should be the empty list if `i > n`.

(iv) 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)`.

(v) More recursion. A list can contain a mixture of numbers and sublists, as in

```
(define list1 '(1 (1 2) (1 2 3)))
```

and

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

Write a function `(numbers mylist)` that counts the number of numbers in `mylist`. The result of `(numbers list1)` should be 6, and that of `(numbers list2)` should be 8.

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

Exercise 12.2 (Scheme and code generation)

(i) The exponential function e^x can be approximated by the expression

$$1 + \frac{x}{1} \left(1 + \frac{x}{2} \left(1 + \frac{x}{3} \left(1 + \frac{x}{4} \left(1 + \frac{x}{5} (\dots) \right) \right) \right) \right)$$

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 1, replacing ‘...’ at the end with 1. Note the similarity to Exercise 12.1 part (iii).

Thus (`makeexp 1 0`) should be the expression 1, and (`makeexp 1 1`) should be the expression $1 + x * 1/1 * 1$, and (`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 (`* x y z`).

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

```
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 equal to 0, 1 and 2.

Then define x to be 2 and compute (`eval (makeexp 1 n)`) for n equal 1, 10 and 20. The correct result is close to 7.38905609893065.

(ii) Now define a 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 n`) should use (`makeexp 1 n`) to do its job.

Exercise 12.3 (.NET runtime code generation)

To get started, download the example file `rtcg.zip` and unpack it; compile `RTCG2D.cs` and run it; and look at the contents of file `RTCG2D.cs`.

For the exercises below, use the `DynamicMethod` approach when generating new methods at runtime. Using class `DynamicMethod` from namespace `System.Reflection.Emit` one can generate a method that can be turned into a delegate, which can then be called as any other delegate. Methods generated this way can also be collected by the garbage collector when no longer in use.

(i) Modify the `RTCG2D.cs` example 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. Check that the new method works by calling it with arguments 1.0 and 10.0.

(ii) Modify the `RTCG2D.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;
}
```

Note that you need to change the generated method’s signature, and you that the generated delegate’s type will be `Func<double, double, double>`, where `Func<>` is the generic delegate type from .NET namespace `System`.

(iii) Write a C# method

```
public static Func<int,int> MakeMultiplier(int c) { ... }
```

that takes as argument an integer c , and then generates and returns a delegate, of type `Func<int, int>`, that corresponds to a method declared like this:

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

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`.

(iv) The exponential function e^x can be computed by the expression

$$1 + \frac{x}{1} \left(1 + \frac{x}{2} \left(1 + \frac{x}{3} \left(1 + \frac{x}{4} \left(1 + \frac{x}{5} (\dots) \right) \right) \right) \right)$$

Write a C# method

```
public static Func<double,double> MakeExp(int n) { ... }
```

that takes as argument an integer `n` and uses runtime code generation to generate and return a delegate, of type `Func<double, double>`, that corresponds 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.

Hint (b): To push integer `i` as a double, use .NET bytecode instruction `ldc.r8 i`, which can be generated like this:

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

(v) Write a C# method

```
public static Func<double,double> MakePower(int n) { ... }
```

that takes as argument an `int n`, and uses runtime code generation to generate and return a delegate, of type `Func<double, double>`, that corresponds 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.