Fachbereich Mathematik und Informatik
Prof. Dr. K. Ostermann

Sebastian Erdweg, seba@informatik
Tillmann Rendel, rendel@informatik

Philipps Universität Marburg

October 22, 2009

Programming Languages and Types
Homework Assignment H2

Please hand in your homework by email to `mailto:pllecture@informatik.uni-marburg.de` until October 29. Please submit your solutions in appropriate file formats.

H2.1 If Zero

Solve Exercise 4.3.1 in the textbook, that is, add the `if0` construct to all relevant definitions of F1WAE including the parser. Furthermore, add a testcase (using the `test` function) which includes a definition and usage of the factorial function. Feel free to add more language primitives (such as multiplication) to the language, if you think this is necessary.

H2.2 If Zero 2

It is unsatisfactory to have many language primitives (such as `if0`). It would be more elegant if there would just be a number of "pre-installed" functions that are called using normal function application.
Let's say you define a function `myif0` as a F1WAE-function of three parameters (ignore the problem that our functions can only have one parameter). The parameters are `if-part`, `then-part` and `else-part`, and `(myif0 if-part then-part else-part)` is just implemented as `(if0 if-part then-part else-part)`.
Think about whether you could use `myif0` rather than the builtin `if0` in your factorial function. Explain why or why not. If not, think about a possible fix that would let you use `myif0`.

H2.3 Lazy Evaluation

Lazy evaluation can modularize backtracking algorithms in an elegant way. Here is a solution to the 8-queens problem in Haskell using lazy evaluation:

please turn the page!

```
boardSize = 8

queens 0 = [[]]
queens n = [ x : y | y <- queens (n-1),
                     x <- [1..boardSize], safe x y 1]

safe x [] n = True
safe x (c:y) n = x /= c &&
                 x /= c + n &&
                 x /= c - n &&
                 safe x y (n+1)

main = print (queens 8)
```

Understand how this program works. What happened to the usual backtracking? (you don't need to write this in your hand-in). If you don't understand the syntax in the 4th line, see `http://www.haskell.org/haskellwiki/List_comprehension`. Compare the given program with a typical "eager" solution such as `http://www.cs.princeton.edu/introcs/23recursion/Queens.java.html`. What would you have to do in both version, respectively, to

a) compute only the first 5 solutions and then stop,

b) compute the number of solutions,

c) find all solutions where the queen in column 3 is at row 5?

You do not need to describe the actual changes or implementations of a), b) and c). Instead outline briefly what you would have to do in both versions, and describe to what extend the two versions differ.