DM 509 Programming Languages

Spring 2012 Re-exam Project

Department of Mathematics and Computer Science
University of Southern Denmark

May 14, 2012

Introduction

The purpose of the project for DM509 is to try in practice the use of logic and functional programming for small but non-trivial examples. The project consists of two parts. The first deals with logic programming and the second part with functional programming. Please make sure to read this entire note before starting your work on the project. Pay close attention to the sections on deadlines, deliverables, and exam rules.

Exam Rules

This project is the 4th quarter re-exam for the course.

Thus, the project must be done individually, and no cooperation is allowed beyond what is explicitly stated in this document.

Deliverables

A short project report (at least 6 pages without front page and appendix) has to be delivered. This report has to contain the following 7 sections (either twice for each part or covering both parts at once):

- **front page** (course number, name, section, date of birth)
- **specification** (what the programs are supposed to do)
- **design** (how the programs were planned)
- **implementation** (how the programs were written)
- **testing** (what tests you performed)
- **conclusion** (how satisfying the results are)
- **appendix** (complete source code)

The report has to be delivered in TWO copies:

- 1 electronic copy using Blackboard's Assignment Hand-In functionality
- 1 paper copy to the teacher's mailbox at the secretariat

Deadline

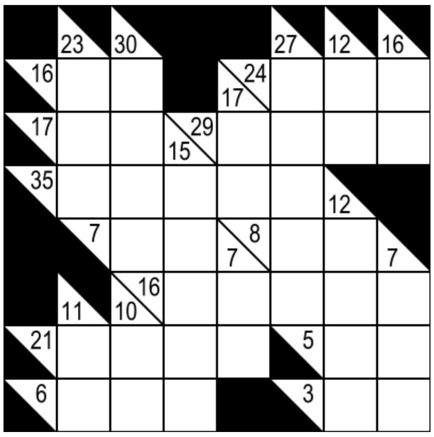
June 18, 2012, 12:00

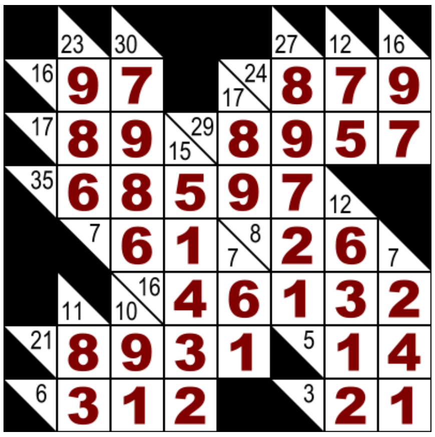
Part 1

Your task in this part of the project is to write a solver for Kakuro puzzles. Kakuro puzzles are a kind of crossword puzzles with numbers where the following two conditions have to be met:

- In each consecutive row or column of empty fields, all numbers have to be from the set $\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$ and they must all be different.
- For each consecutive row or column of empty fields, a number at the to the left or above of the fields specifies the sum of all these numbers.

The following two figures show a Kakuro puzzle and its solution (taken from Wikipedia, both graphics are under the GNU Free Documentation License).





The Input

For input to your program, the Kakuro puzzles are represented as Prolog terms. More specifically, they are represented as matrices (lists of rows which are lists of fields) where the fields can be one of the following four types:

- A frame block with no sum information (black in the graphics above) is represented by the term `x/x`.
- A frame block with sum information for a row (black lower left, number N in upper right) is represented by the term `x/N`, e.g., `x/16` or `x/24`.
- A frame block with sum information for a column (number N in lower left, black upper right) is represented by the term `N/x`, e.g., `23/x`.
- An empty field is represented by just the term `x`.

Thus, for our example above we obtain the following Prolog term:

```
[[x/x , 23/x, 30/x , x/x  , x/x  , 27/x, 12/x, 16/x],
 [x/16, x   , x    , x/x  , 17/24, x   , x   , x   ],
 [x/17, x   , x    , 15/29, x    , x   , x   , x   ],
 [x/35, x   , x    , x    , x    , x   , 12/x, x/x ],
 [x/x , x/7 , x    , x    , 7/8  , x   , x   , 7/x ],
 [x/x , 11/x, 10/16, x    , x    , x   , x   , x   ],
 [x/21, x   , x    , x    , x    , x/5 , x   , x   ],
 [x/6 , x   , x    , x    , x/x  , x/3 , x   , x   ]].
```

The home page of the course contains a number of possible inputs to test your program on.

The Output

The output of your solver should also be a Prolog term. The representation is similar to the one for the Input except for all `x` being replaced by the appropriate number.

Thus, for our example above we obtain the following Prolog term:

```
[[x/x , 23/x, 30/x , x/x  , x/x  , 27/x, 12/x, 16/x],
 [x/16, 9   , 7    , x/x  , 17/24, 8   , 7   , 9   ],
 [x/17, 8   , 9    , 15/29, 8    , 9   , 5   , 7   ],
 [x/35, 6   , 8    , 5    , 9    , 7   , 12/x, x/x ],
 [x/x , x/7 , 6    , 1    , 7/8  , 2   , 6   , 7/x ],
 [x/x , 11/x, 10/16, 4    , 6    , 1   , 3   , 2   ],
 [x/21, 8   , 9    , 3    , 1    , x/5 , 1   , 4   ],
 [x/6 , 3   , 1    , 2    , x/x  , x/3 , 2   , 1   ]].
```

The Task

Implement a predicate `solve/2` that takes an unsolved Kakuro puzzle as the first argument and instantiates the second argument by its solved form.

Keep in mind, that there are many different ways how to implement such a `solve/2` predicate. Explain your approach, then implement it and produce the final report.

The Foundations

There is a number of built-in predicates that you might find useful when building a Kakuro solver:

- `var/1`, which is true if the argument is an (uninstantiated) variable
- `fd_var/1`, which is true if the argument is an (uninstantiated) constraint variable
- `number/1`, which is true if the argument is an integer or a floating point number
- `read/1`, `write/1`, and `nl/0` for input and output
- `fd_domain/3`, `fd_all_different/1`, `fd_labeling/1`, and `#=` for constraint solving

To make life easier for you, I have also defined some predicates for outputting Kakuro puzzles (`show/1`, works both on puzzles in input and in output form) and for converting Kakuro puzzles in a different notation to our representation (`convert/2`, works when the first argument is a term as used in the additional examples linked from the course home page).

Finally, there is a template available from the course home page for calling your `solve/2` predicate (see next section) using the predicate `kakuro/0`.

Example Output

The printed output when posing the query `?- kakuro.` and inputting the input from above could be:

```
Please enter puzzle as matrix:
[[x/x , 23/x, 30/x , x/x  , x/x  , 27/x, 12/x, 16/x],
 [x/16, x   , x    , x/x  , 17/24, x   , x   , x   ],
 [x/17, x   , x    , 15/29, x    , x   , x   , x   ],
 [x/35, x   , x    , x    , x    , x   , 12/x, x/x ],
 [x/x , x/7 , x    , x    , 7/8  , x   , x   , 7/x ],
 [x/x , 11/x, 10/16, x    , x    , x   , x   , x   ],
 [x/21, x   , x    , x    , x    , x/5 , x   , x   ],
 [x/6 , x   , x    , x    , x/x  , x/3 , x   , x   ]].
```

```
+-------+-------+-------+-------+-------+-------+-------+-------+
| x /x  | 23/x  | 30/x  | x /x  | x /x  | 27/x  | 12/x  | 16/x  |
+-------+-------+-------+-------+-------+-------+-------+-------+
| x /16 |       |       | x /x  | 17/24 |       |       |       |
+-------+-------+-------+-------+-------+-------+-------+-------+
| x /17 |       |       | 15/29 |       |       |       |       |
+-------+-------+-------+-------+-------+-------+-------+-------+
| x /35 |       |       |       |       |       | 12/x  | x /x  |
+-------+-------+-------+-------+-------+-------+-------+-------+
| x /x  | x /7  |       |       | 7 /8  |       |       | 7 /x  |
+-------+-------+-------+-------+-------+-------+-------+-------+
| x /x  | 11/x  | 10/16 |       |       |       |       |       |
+-------+-------+-------+-------+-------+-------+-------+-------+
| x /21 |       |       |       |       | x /5  |       |       |
+-------+-------+-------+-------+-------+-------+-------+-------+
| x /6  |       |       |       | x /x  | x /3  |       |       |
+-------+-------+-------+-------+-------+-------+-------+-------+
Setting up constraints ... DONE
Solving constraints    ... DONE
+-------+-------+-------+-------+-------+-------+-------+-------+
| x /x  | 23/x  | 30/x  | x /x  | x /x  | 27/x  | 12/x  | 16/x  |
+-------+-------+-------+-------+-------+-------+-------+-------+
| x /16 | 9     | 7     | x /x  | 17/24 | 8     | 7     | 9     |
+-------+-------+-------+-------+-------+-------+-------+-------+
| x /17 | 8     | 9     | 15/29 | 8     | 9     | 5     | 7     |
+-------+-------+-------+-------+-------+-------+-------+-------+
| x /35 | 6     | 8     | 5     | 9     | 7     | 12/x  | x /x  |
+-------+-------+-------+-------+-------+-------+-------+-------+
| x /x  | x /7  | 6     | 1     | 7 /8  | 2     | 6     | 7 /x  |
+-------+-------+-------+-------+-------+-------+-------+-------+
| x /x  | 11/x  | 10/16 | 4     | 6     | 1     | 3     | 2     |
+-------+-------+-------+-------+-------+-------+-------+-------+
| x /21 | 8     | 9     | 3     | 1     | x /5  | 1     | 4     |
+-------+-------+-------+-------+-------+-------+-------+-------+
| x /6  | 3     | 1     | 2     | x /x  | x /3  | 2     | 1     |
+-------+-------+-------+-------+-------+-------+-------+-------+

yes
```

Part 2

Your task in this part of the project is to work with a data structure for representing polynomial fractions, i.e., expressions built from variables, constants, addition, subtraction, multiplication, and division.

The data structure is defined in the following way:

```
data Operator = Add | Sub | Mul | Div
data Poly = C Float | V String | Op Poly Operator Poly
```

Using this data structure, we can for example represent the polynomial $4 - x + 3x^2$ by the following expression:

```
Op (Op (C 4) Sub (V "x")) Add (Op (C 3) Mul (Op (V "x") Mul (V "x")))
```

There is a template available from the course home page that defines this data structure and binds this expression to the variable `testpoly`.

This template also contains an incomplete definition of a function for evaluating variable-free polynomial fractions:

```
eval :: Poly -> Float
eval (Op p1 o p2) = interpret o val1 val2 where
    val1 = eval p1
    val2 = eval p2
```

The Tasks

Implement the following operations on polynomial fractions by implementing the following functions (and any auxiliary functions you might consider needed):

1. Implement the function `interpret` used in `eval` above to complete the definition of our evaluation function. Use the form given in the template.

2. Define a function `derive :: String -> Poly -> Poly` which computes the (symbolic) derivation of a polynomial fraction with respect to a given variable identified by its name. For example, the expression `derive "x" testpoly` should return a polynomial which corresponds to `-1 + 6x`.

3. Define a function `simplify :: Poly -> Poly` to simplify polynomial fractions. You can use rules like $0 + x = x$ and $0 * x = 0$ as well as the function `eval`. For example, `simplify (derive "x" testpoly)` should return a result at least as simple as $-1 + 3 * (x + x)$. And `simplify (Op (V "x") Div (C 1.0))` should evaluate to `V "x"`.

4. Define a data type `Substitution` that maps some variables to constant values. Then define a function of the type `instantiate :: Substitution -> Poly -> Poly` that takes a polynomial fraction and instantiates all variables mentioned in the substitution by the corresponding constant. After instantiating the variables, the resulting expression should be simplified. For example, if you call `instantiate` on `testpoly` with a substitution that maps x to 1.0, the result should be `C 6.0`.

Hint: By removing `deriving Show` behind the definitions of `Operator` and `Poly` and uncommenting the `show` declarations at the bottom of the template, you can view the polynomials in a more human-readable format.