McMaster University
Department of Computing and Software
Dr. W. Kahl

SFWR ENG 3E03
Exercise Sheet 6
corrected version

Design and Selection of Programming Languages

20 October 2005

Exercise 6.1 — *Map* and *Set*

In the "**hierarchical libraries**" currently available in Hugs and GHC there are modules providing the following datatypes:

- **data** *Map a b*
  implements (finite) partial functions from *a* to *b*, i.e., elements of the set ($a \nrightarrow b$).
- **data** *Set a*
  implements finite sets of elements of type *a*.

(a) Familiarise yourself with the interface to theses datatypes — you find the documentation inside the "Hierarchical Libraries" part of the GHC documentation either in your own GHC installation, or below http://haskell.org/ghc/.

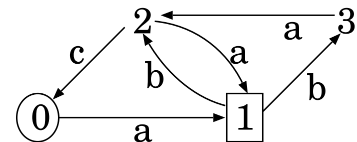
The following Haskell module should load without problems in both Hugs (Nov 2003) and GHCi:

```
module NewFSM where

import qualified Data.Map as Map hiding (Map)
import Data.Map (Map)
import qualified Data.Set as Set hiding (Set)
import Data.Set (Set)

type Symbol = Char
type Word = [Symbol]
type TransRel state = Map (state, Symbol) (Set state)
type TransFun state = Map (state, Symbol) state

tr1 :: TransRel Int
tr1 = Map.fromList
  [((1,'a'), Set.singleton 2)
  ,((1,'b'), Set.singleton 3)
  ,((2,'b'), Set.singleton 3)
  ,((3,'a'), Set.singleton 1)
  ,((3,'b'), Set.singleton 2)
  ]
```



(b) Define *tr2* :: *TransRel Int* to be the transition relation of the automaton drawn above.

(c) Define the following functions, corresponding to those from FSM.hs from the first lecture:

```
processSymbol :: Ord state ⇒ TransRel state → Symbol → state → Set state
processWord :: Ord state ⇒ TransRel state → Word → state → Set state
```

(d) Provide appropriate type synonyms *FSM* for (non-determinsitic) finite-state machines and DFSM for deterministic finite-state machines.

(e) Define now also the following functions:

accept :: *Ord state* ⇒ *FSM state* → *Word* → *Bool* −− is word accepted by FSM?

isDet :: *Ord state* ⇒ *TransRel state* → *Bool* −− is transition relation deterministic?

reachable :: *Ord state* ⇒ *FSM state* → *Set state* −− set of reachable states
unreachable :: *Ord state* ⇒ *FSM state* → *Set state* −− set of unreachable states

usedSymbols :: *Ord state* ⇒ *FSM state* → *Set Symbol* −− set of symbols on reachable edges

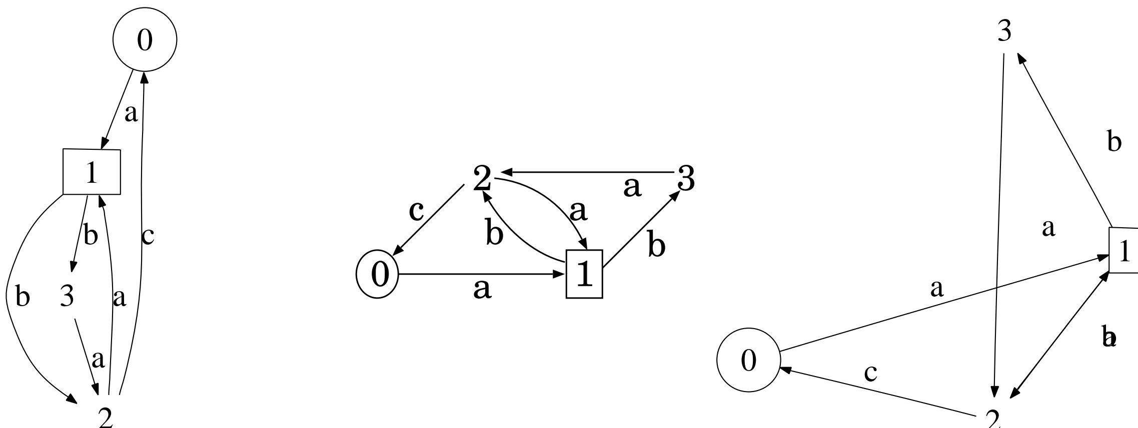
Exercise 6.2 — Drawing Finite-State Machines Using `dot`

The graphviz tool suite from AT&T includes the graph layout tools

- dot for layout of directed, typically acyclic graphs — the main principle of the algorithm is to arrange nodes into “levels”, and then reduce edge crossings, and
- neato for layout of undirected graphs using a “spring embedding” algorithm that understands nodes as electically charged and therefore repelling each other, and edges as springs of equal spring constants and lengths, and therefore produces always drawings with straight-line edges.

Here are the (scaled-down) drawings these tools produce for the finite-state machine from Exercise 6.1(b) (in the middle), using the following invocations:

```
dot -Tps -o MT1a-dot.ps MT1a.dot
neato -Tps -o MT1a-neato.ps MT1a.neato
```



As you can see, neither of them is ideal for finite-state machines, but frequently such quick visualisations are much more useful than manual analysis or drawing…

Input file for dot:

```
digraph MT1a {
  node [shape="plaintext",fontsize="30"];
```

```
  edge [labeldistance="1",fontsize="30"];
  "0" [shape="circle"];
  "1" [shape="box"];
  "0" -> "1" [label="a"];
  "1" -> "2" [label="b"];
  "1" -> "3" [label="b"];
  "2" -> "0" [label="c"];
  "2" -> "1" [label="a"];
  "3" -> "2" [label="a"];
}
```

Input file for neato:

```
graph MT1a {
  node [shape="plaintext",fontsize="30",height="0",width="0"];
  edge [dir="forward",labeldistance="1",fontsize="30",len="4"];
  "0" [shape="circle"];
  "1" [shape="box"];
  "0" -- "1" [label="a"];
  "1" -- "2" [label="b"];
  "1" -- "3" [label="b"];
  "2" -- "0" [label="c"];
  "2" -- "1" [label="a"];
  "3" -- "2" [label="a"];
}
```

(a) Produce functions to generate dot and neato files for finite-state machines represented as values of the datatypes *FSM* and *DFSM* from Exercise 6.1.

Take care to share as much functionality as possible using appropriate function abstraction.

(b) Use the standard library function *System.system* to invoke dot resp. neato from within a Haskell *IO* computation.

Document your design and your function specifications!

Exercise 6.3 — Implementation of FSM Constructions

(a) The union of the languages accepted by two FSMs is accepted by a union FSM.

Spell out the mathematical definition.

Implement the construction in Haskell, and visualise some examples!

(b) The concatenation of the languages accepted by two FSMs is accepted by a different union FSM.

Spell out the mathematical definition, implement the construction in Haskell, and visualise some examples!

(c) The intersection of the languages accepted by two FSMs is accepted by a product FSM.

Spell out the mathematical definition, implement it in Haskell, and visualise some examples!

(d) Implement the powerset construction of a deterministic FSM from a non-determinsitic FSM.

Visualise some examples!

Exercise 6.4 — Lexing

Using the Haskell module SimpleLexer from the course page as a starting point, produce a Haskell module JayLexer for lexing the tokens of Jay, as defined in appendix B.1 of the textbook:

InputElement → *WhiteSpace* | *Comment* | *Token*

WhiteSpace → `space` | \t | \r | \n | \f | \r\n

Comment → `//` any string ended by \r or \n or \r\n

Token → *Identifier* | *Keyword* | *Literal* | *Separator* | *Operator*

Identifier → *Letter* | *Identifier Letter* | *Identifier Digit*

Letter → `a` | `b` | … | `z` | `A` | `B` | … | `Z`

Digit → 0 | 1 | … | 9

Keyword → `boolean` | `else` | `if` | `int` | `main` | `void` | `while`

Literal → *Boolean* | *Integer*

Boolean → `true` | `false`

Integer → *Digit* | *Integer Digit*

Separator → `(` | `)` | { | } | `;` | `,`

Operator → `=` | `+` | `-` | `*` | `/` | `<` | `<=` | `>` | `>=` | `==` | `!=` | `&&` | `||` | `!`

Every input element that is not a *Token* — that is, every instance of *Whitespace* or *Comment* — is bypassed during the lexical analysis of a Jay program.

For this purpose, you may want to state a regular grammar more explicitly.