

= 0 : 1 : 3 : []

- 3% per necessary step:
- 1% for reducing the right redex
 - 2% for performing the reduction correctly
 - -1% for not writing down
-

Exercise 5.2 — Haskell Typing (19% of Midterm 1, 2004)

Provide **detailed derivations** of the Haskell types of the following functions:

```
swibble x y = [ ( x , y ) , ( x ++ " ' ", y + 1 ) ]
```

```
swoon g h = [ g ( (1 + ) . h ) ]
```

Solution Hints

Type classes have not been taught yet, only mentioned: Numeric types can be defaulted to *Integer* or *Int*.

$swibble :: (Num\ n) \Rightarrow String \rightarrow n \rightarrow [(String, n)]$

Assuming $1 :: Integer$, we must have $y :: Integer$ because of $y + 1$.

Since $"" :: String$, we also have $x :: String$ because of $x ++ "" :: String$.

Then $(x, y) :: (String, Integer)$, and the type of *swibble* follows easily.

$swoon :: (Num\ n) \Rightarrow ((a \rightarrow n) \rightarrow b) \rightarrow (a \rightarrow n) \rightarrow [b]$

Assuming $1 :: Integer$, we have $(1 +) :: Integer \rightarrow Integer$, and because of the composition, we must have

$h :: a \rightarrow Integer$ for some type a .

Therefore, we have $((1 +) \circ h) :: a \rightarrow Integer$, and may assume $g :: (a \rightarrow Integer) \rightarrow b$ for some type b .

Then we have $[g((1 +) \circ h)] :: b$, and therefore

$swoon\ g :: (a \rightarrow Integer) \rightarrow b$

and

$swoon :: ((a \rightarrow Integer) \rightarrow b) \rightarrow (a \rightarrow Integer) \rightarrow b$.

Exercise 5.3 — Defining Haskell Functions (20% of Midterm 1, 2004)

Define the following Haskell functions (the solutions are independent of each other):

(a) $sum :: [Integer] \rightarrow Integer$

such that $sum\ xs$ evaluates to the sum of all elements of the list xs .

(b) $all :: (a \rightarrow Bool) \rightarrow [a] \rightarrow Bool$

such that $all\ p\ xs$ evaluates to **True** if p considered as a predicate holds for all elements of xs , and

to **False** if there is at least one element in *xs* for which *p* does not hold.

E.g., *all* (*> 1*) [2..10] = **True**

(c) *selMod* :: *Integer* → [*Integer*] → [*Integer*]

such that *selMod* *k* *xs* selects from the list *xs* all those elements that are equivalent to *k* modulo *k + 1*, e.g.,

selMod 2 [2, 3, 8, 1, 2, 5] = [2, 8, 2, 5]

(d) *sources* :: *Eq a* ⇒ [(*a*, *a*)] → [*a*]

such that *sources* *ps* returns the *sources* of the graph *ps*.

Here, the list *ps* of pairs is considered as representing a simple graph by representing each edge from node *x* to node *y* by the pair (*x*, *y*).

The *context* “*Eq a* ⇒” just means that you may use the equality test for elements of type *a*, i.e., (*==*) :: *a* → *a* → *Bool*.

Example: *sources* [(2,3), (3,4), (1,4), (1,5), (2,5)] = [2,1]

(The order is irrelevant.)

Solution Hints

sum = *foldl* (+) 0

sum = *foldr* (+) 0

sum [] = 0

sum (x:xs) = x + *sum* xs

all = *foldr* (&&) **True**

all p [] = **True**

all p (x:xs) = p x && *all* p xs

selMod :: *Integer* → [*Integer*] → [*Integer*]

selMod *k* *xs* = [x | x ← *xs* , x ‘mod’ (k+1) ≡ k]

sources, *sources'* :: *Eq a* ⇒ [(*a*, *a*)] → [*a*]

sources *ps* = **let** (*srcs*, *trgs*) = *unzip* *ps*

in *filter* (‘notElem’ *trgs*) *srcs*

sources' *ps* = **let** *trgs* = [*snd* p | p ← *ps*]

in [x | (x,y) ← *ps*, x ‘notElem’ *trgs*]

Exercise 5.4 — Finite-State Machines (25% of Midterm 1, 2004)

Let the following type synonyms be given, as in the presentation in the first lecture:

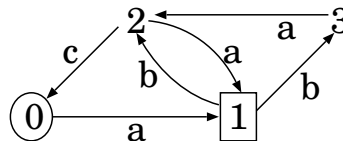
type *State* = *Int*

type *Symbol* = *Char*

type *TransRel* = [(*State*, *Symbol*, *State*)]

type *FSM* = (*State*, *TransRel*, [*State*])

- (a) Define *fsm1* :: *FSM* such that it represents the following finite-state machine (with start state circled and end states in boxes):



- (b) Define the Haskell function *isDet* :: *FSM* → *Bool*

such that *isDet fsm* evaluates to the Boolean value indicating whether the finite-state machine *fsm* is deterministic or not.

For example, *isDet fsm1* = **False** since leaving state 1, there are two *b*-edges directed towards different nodes.

Hint: Define auxiliary functions! For example:

- Calculate all start nodes of transitions in a *TransRel*.
- Given a state, calculate all edges leaving that state in a *TransRel*.
- Given a *Symbol* and a *TransRel*, find all target nodes of edges with that symbol.
- Given a *State* and a *TransRel*, find out whether any edges leaving that state violate determinacy.

Other functions may be useful, too. **Document your functions!**

Solution Hints

type *State* = *Int*

type *Symbol* = *Char*

type *TransRel* = [(*State*, *Symbol*, *State*)]

type *FSM* = (*State*, *TransRel*, [*State*])

fsm1 :: *FSM* -- 6%

fsm1 = (0, tr1, [1])

where

tr1 =

[(0, 'a', 1)

, (1, 'b', 2)

, (1, 'b', 3)

, (2, 'a', 1)

```
,(2,'c',0)
,(3,'a',2)
]
```

```
edgeStarts tr = [ s | (s, c, t) ← tr ] -- 3%
```

```
outEdges tr s = [ (c, t) | (s', c, t) ← tr, s' ≡ s ] -- 3%
```

```
isUnique es (c, t) = all (t ≡) [ t' | (c', t') ← es, c' ≡ c ] -- 5%
```

```
isDetState tr s = all (isUnique es) es -- 4%
```

```
  where es = outEdges tr s
```

```
isDet (s0, tr, fin) = all (isDetState tr) (edgeStarts tr) -- 4%
```
