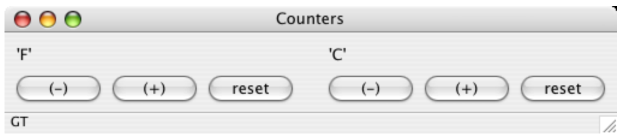
ST Master Course on Advanced Functional Programming

Friday, April 21, 2006 (9:00-12:00)

The exam consists of 5 open questions: the maximum number of points for each question is given (100 points in total). Give short and precise answers. If a Haskell function is asked for, try to find an elegant solution. It is recommended to read all parts of a question before you provide an answer. You may consult course material during the test. Good luck!

1 Counter Panel (15 points)

A *counter panel* is a panel that contains a text label showing a value, and a decrement, an increment, and a reset button. Such a panel is polymorphic in the value it controls, as long as we can *show* and *compare* values of that type. Furthermore, we use the *Enum* type class to implement the decrement and increment operations: *pred* (for decrement) and *succ* (for increment) both have type $Enum\ a \Rightarrow a \rightarrow a$. The following frame contains two counter panels for characters:



The status bar displays GT, which is the result of comparing the two current values (because `'F'` > `'C'`). The value of a counter panel is restricted to a minimum and a maximum value. For this, we introduce another type class:

```
class (Show a, Ord a, Enum a) ⇒ MinMax a where
  minValue :: a
  maxValue :: a

instance MinMax Char where
  minValue = 'A'
  maxValue = 'Z'
```

The data type *Control* (from the second lab assignment) is used to implement the observer/observable pattern.

```
data Control a = C{model :: IORef a, observers :: IORef [IO ()]}

createControl :: a → IO (Control a)
createControl a =
  do m   ← newIORef a
     obs ← newIORef []
     return (C m obs)

getValue :: Control a → IO a
getValue = readIORef ∘ model

setValue :: Control a → a → IO ()
setValue ctrl a =
  do writeIORef (model ctrl) a
     readIORef (observers ctrl) ≫= sequence_
```

```
addObserver :: Control a → IO () → IO ()
addObserver ctrl callback =
  modifyIORef (observers ctrl) (callback:) ≫ callback
```

Now we can define the function *main*, which creates a status bar, two counter panels, and puts these into a frame.

```
main :: IO ()
main = start $
  do f      ← frame [text := "Counters"]
     status ← statusField []

     (cp1, ctrl1 :: Control Char) ← counterPanel f
     (cp2, ctrl2 :: Control Char) ← counterPanel f

     let callback c = addObserver c (updateBar status ctrl1 ctrl2)
     mapM_ callback [ctrl1, ctrl2]

     set f [statusbar := [status], layout := row 10 [widget cp1, widget cp2]]
```

Two functions used by *main* are not yet defined: their type signatures are:

```
updateBar    :: MinMax a ⇒ StatusField → Control a → Control a → IO ()
counterPanel :: MinMax t ⇒ Window a → IO (Panel (), Control t)
```

a) Give a definition for *updateBar* such that the status bar reflects the comparison of the two counter characters. Use *compare* :: *Ord a* ⇒ *a* → *a* → *Ordering* for comparing the *Char*s.

```
updateBar sf ctrl1 ctrl2 =
  do x ← getValue ctrl1
     y ← getValue ctrl2
     set sf [text := show (x `compare` y)]
```

b) Define the function *counterPanel* such that:

- A new control and a new panel are created. Use *minValue* as the initial value.
- The panel should contain one text label and three buttons. The layout of the panel should resemble the layout of the screenshot.
- Implement the *command* events for the three buttons. Make sure that the value remains between *minValue* and *maxValue* at all time. The reset button should set the value to *minValue*.
- The text label showing the counter panel's value should change whenever the value of its control changes.

```
counterPanel w =
  do ctrl  ← createControl minValue
     p     ← panel w []
     txt   ← staticText p []
     decr  ← button   p [text := "(-)",   on command := onDecr  ctrl]
     incr  ← button   p [text := "(+)",   on command := onIncr  ctrl]
     reset ← button   p [text := "reset", on command := onReset ctrl]
     set p [layout := margin 10 $ column 10
                      [widget txt, row 10 [widget decr, widget incr, widget reset]]]
     addObserver ctrl (updateText txt ctrl)
     return (p, ctrl)

  -- event handlers
onDecr  ctrl = changeValue ctrl (max minValue ∘ pred)
onIncr  ctrl = changeValue ctrl (min maxValue ∘ succ)
onReset ctrl = setValue ctrl minValue

updateText :: Show a ⇒ StaticText () → Control a → IO ()
updateText t ctrl =
  do x ← getValue ctrl
     set t [text := show x]

changeValue :: Control a → (a → a) → IO ()
changeValue control f =
  do a ← getValue control
     setValue control (f a)
```

c) Suppose that we want to *share* the value of the two counters: pressing the (+) button of the left counter also increments the value displayed by the right counter (and vice versa). Describe how the program should be changed (code is not required).

Instead of creating a *Control* inside the *counterPanel* function, we will pass a *Control* to this function: *counterPanel* now expects two arguments. Only one *Control* is created in the *main* function, which is then passed to both counterPanels.

```
main :: IO ()
main = start $
  do ...
     ctrl ← createControl (minValue :: Char)
     cp1 ← counterPanel f ctrl
     cp2 ← counterPanel f ctrl
     ...
```

2 List Monad (25 points)

The following list comprehension generates an infinite list containing infinite lists:

```
squareList :: [[Int]]
squareList = [[sq, sq * 2 ..] | i ← [1 ..], let sq = i * i]
```

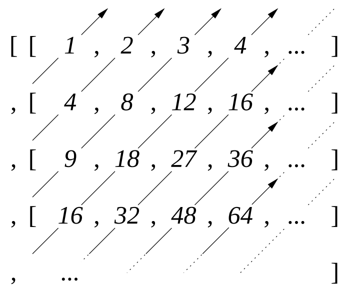
This list of lists could be visualized as follows:

(1 : 2 : 3 : 4: ...) : (4 : 8 : 12 : 16: ...) : (9 : 18 : 27 : 36: ...) : (16 : 32 : 48 : 64: ...) : ...

We have the *Prelude* function *concat* at our disposal to flatten this list. However, *concat squareList* will *only* return elements from the first list. The function *join*, defined below, has *concat*'s type, but takes elements in a diagonal fashion:

```
join :: [[a]] → [a]
join = rec []

  where
    rec [] [] = []
    rec as bs =
      let notEmpty = not ∘ null
          (hd, tl)  = splitAt 1 bs
          rest      = hd ++ map tail as
      in map head as ++ rec (filter notEmpty rest) tl
```



Indeed, the expression *take* 10 (*join squareList*) evaluates to $[1,4,2,9,8,3,16,18,12,4]$.

a) Explain in your own words why the potentially dangerous functions *head* and *tail* in *join*'s definition are safe (no pattern match failures).

The functions *head* and *tail* are both mapped over *as* (*rec*'s first argument). All lists in *as* are non-empty because we start with the empty list (contains no list at all), and for each recursive call to *rec* we first throw away all empty lists (*filter notEmpty rest*).

b) We continue with the introduction of a wrapper data type for a normal Haskell list:

```
newtype List a = List{ listify :: [a]} deriving Eq
```

This brings into scope the unwrapper function *listify* of type *List a* → [*a*]. Make *List* an instance of the *Functor* type class:

```
class Functor f where
  fmap :: (a → b) → f a → f b
```

Because normal lists are in the *Functor* type class, we can write:

```
instance Functor List where
  fmap f = List ∘ fmap f ∘ listify
```

c) Give a point-free definition for *joinList* :: *List* (*List a*) → *List a*, which uses the function *join* to combine lists. There will be a small penalty for a definition that is not point-free.

```
joinList = List ∘ join ∘ listify ∘ fmap listify
```

d) Having *join* and *map* (or actually, *fmap*) defined makes it easy to define the monadic (≫=) operator. Make *List* an instance of the *Monad* type class. Of course you may reuse code defined earlier.

```
instance Monad List where
  return a = List [a]
  as ≫= f  = joinList (fmap f as)
```

e) Remember the three algebraic laws for monads:

$$\begin{aligned} \mathit{return}\ x \gg\!\!= f &\equiv f\ x && \text{(LEFT UNIT)} \\ m \gg\!\!= \mathit{return} &\equiv m && \text{(RIGHT UNIT)} \\ m_1 \gg\!\!= (\lambda x \to m_2 \gg\!\!= (\lambda y \to m_3)) &\equiv (m_1 \gg\!\!= (\lambda x \to m_2)) \gg\!\!= (\lambda y \to m_3) && \text{(BIND)} \end{aligned}$$

assuming that x does not appear free in m_3

Do the monad laws hold for the instance defined at **d)**? If you think they hold, give a short motivation (no formal proof is required). Otherwise, give a counter-example.

The first two laws hold, but not (BIND). For proving the left and right unit laws, you can use the following properties of the *join* function:

$$\begin{aligned} &\mathit{join}\ [\mathit{xs}] &&== \mathit{xs} && \text{-- only one row} \\ &\mathit{join}\ (\mathit{map}\ (\lambda x \to [x])\ \mathit{xs}) &&== \mathit{xs} && \text{-- only one column} \end{aligned}$$

To construct a counter-example for (BIND), we choose $m_1 = \mathit{List}\ [1,2]$, $m_2 = \mathit{List}\ [x,x]$, and $m_3 = \mathit{List}\ [y,y]$:

$$\begin{aligned} &m_1 \gg\!\!= (\lambda x \to m_2 \gg\!\!= (\lambda y \to m_3)) \\ &\quad = \mathit{List}\ [1,2] \gg\!\!= (\lambda x \to \mathit{List}\ [x,x] \gg\!\!= (\lambda y \to \mathit{List}\ [y,y])) \\ &\quad = \mathit{List}\ [1,2,1,2,1,2,1,2] \end{aligned}$$

whereas

$$\begin{aligned} &(m_1 \gg\!\!= (\lambda x \to m_2)) \gg\!\!= (\lambda y \to m_3) \\ &\quad = (\mathit{List}\ [1,2] \gg\!\!= (\lambda x \to \mathit{List}\ [x,x])) \gg\!\!= (\lambda y \to \mathit{List}\ [y,y]) \\ &\quad = \mathit{List}\ [1,2,1,1,2,2,1,2] \end{aligned}$$

f) Define three *QuickCheck* properties to check the monad laws for your *List* instance. You may assume that a random data generator for *List*s is provided.

The type signatures are required by QuickCheck.

```
  -- quickCheck leftUnit returns "OK, passed 100 tests."
leftUnit :: (Int → List Int) → Int → Bool
leftUnit f x = (return x >>= f) == f x

  -- quickCheck rightUnit returns "OK, passed 100 tests."
rightUnit :: List Int → Bool
rightUnit m = (m >>= return) == m

  -- quickCheck bind returns "Falsifiable."
bind :: List Int → (Int → List Int) → (Int → List Int) → Bool
bind m f g   = ((m >>= f) >>= g) == (m >>= (λx → f x >>= g))
```

g) Suppose we want to have a monad transformer for *List*. Give a suitable type definition for this transformer. You do not have to give the instance declarations.

```
newtype ListT m a = ListT (m [a])
```

3 Operational Semantics (20 points)

We will study the behavior of the following function:

$f :: (Int, (Int, Int)) \to [Int]$
$f\ (a, (b, c)) = 0\ :\ a\ :\ f\ (c, (a, b))$

a) Consider the result of $f\ (1 + 2 + 3, (0, 0))$. Whether or not the subexpression $1 + 2 + 3$ is evaluated and reduced to 6 depends on the context of the expression. Describe two situations: one in which the reduction takes place, and one in which the subexpression is not evaluated.

Reduction takes place if we need the second element in the list:

$f\ (1 + 2 + 3, (0, 0))\ !!\ 1$
$= \bot$

The subexpression is not evaluated if the list's elements are not evaluated:

$null\ \$\ f\ (1 + 2 + 3, (0, 0))$
$= False$

To check whether or not the reduction takes place, replace $1 + 2 + 3$ by $\bot$.

b) The functions g, h, and k are variations on f:

$g\ {\sim}(a, (b, c)) = 0\ :\ a\ :\ g\ (c, (a, b))$
$h\ (a, {\sim}(b, c)) = 0\ :\ a\ :\ h\ (c, (a, b))$
$k\ (a, {\sim}(b, c)) = 0\ :\ a\ :\ seq\ b\ (k\ (c, (a, b)))$

Remember that the expression x ‘*seq*‘ y evaluates x to weak head normal form (WHNF) and then returns y. Indicate as precise as possible at which point the evaluation of the expression *show* $\$\ f\ (1, \bot)$ diverges and returns $\bot$. Answer the same question for g, h, and k when used instead of f.

f diverges directly when matching $(1, \bot)$ against $(a, (b, c))$:

show $\$\ f\ (1, \bot)$
$=$ `"*** Exception: Prelude.undefined"`

g delays the pattern match, and diverges when a (the list's second element) is needed:

show $\$\ g\ (1, \bot)$
$=$ `"[0,*** Exception: Prelude.undefined"`

h delays pattern matching against $\bot$ until after the recursive call:

show $\$\ h\ (1, \bot)$
$=$ `"[0,1,0,*** Exception: Prelude.undefined"`

k diverges as a result of *seq*-ing the value b (for which we have to pattern match):

show $\$\ k\ (1, \bot)$
$=$ `"[0,1*** Exception: Prelude.undefined"`

c) Evaluating *length xs* has one of the following outcomes (depending on *xs*):

- The length of the list is returned.
  Example: *length* $[1\,..\,10]$.

- An exception is thrown, or the computation is non-terminating.
  Examples: $length\ (head\ [\,])$ and $length\ [1\,..]$.

What is the outcome of evaluating $length\ \$\ f\ (1, \bot)$? Answering with "the length" or "an exception" suffices. Answer the same question for g, h, and k when used instead of f.

$length\ \$\ f\ (1, \bot)$
$= \bot$

$length\ \$\ g\ (1, \bot)$
$=$ `infinite computation`

$length\ \$\ h\ (1, \bot)$
$=$ `infinite computation`

$length\ \$\ k\ (1, \bot)$
$= \bot$

d) Write a function

$seqTriple :: (a, (b, c)) \to (a, (b, c))$

that forces evaluation to WHNF of all three components.

$seqTriple\ p@(a, (b, c)) = a\ `seq`\ b\ `seq`\ c\ `seq`\ p$

4 Generalized Algebraic Data Types (20 points)

Consider the following data type declarations:

data *Succ n*
data *Zero*

data *LengthList n a* **where**
  $Cons :: a \to LengthList\ n\ a \to LengthList\ (Succ\ n)\ a$
  $Nil\ \ :: LengthList\ Zero\ a$

Be careful: *Succ* and *Zero* are types, not values. The data types *Succ* and *Zero* are empty: no value of such a type exists (except for $\bot$). *LengthList*'s first type argument is a phantom type, which we use for encoding the length of the list using *Succ* and *Zero*.

a) What type is inferred for the expression $Cons\ 1\ (Cons\ 2\ (Cons\ 3\ Nil))$?

Because the numeric literals are overloaded, the inferred type is:

$Num\ a \Rightarrow LengthList\ (Succ\ (Succ\ (Succ\ Zero)))\ a$

The more specific type with *Int* or *Integer* is also approved.

b) Assume that we want *Append*, which combines two lists, to be *LengthList*'s third constructor function. Because the length of *xs* (n_1) and the length of *ys* (n_2) together determine the length of *Append xs ys* ($n = n_1 + n_2$), we need a type class for "adding two types":

class $Add\ n_1\ n_2\ n \mid n_1\ n_2 \to n$

We extend *LengthList*'s GADT with the following constructor function:

$Append :: Add\ n_1\ n_2\ n \Rightarrow LengthList\ n_1\ a \to LengthList\ n_2\ a \to LengthList\ n\ a$

Explain the purpose of the functional dependency in the type class *Add*.

The functional dependency states that *Add*'s third type argument is uniquely determined by the first two type arguments. This restricts the instances that can be given for *Add* (they must adhere to the dependency) and it allows the compiler to improve/simplify types containing an *Add* predicate. For example, the expression *Append* (*Cons* 1 *Nil*) *Nil* can be given the type

$$Add\ (Succ\ Zero)\ Zero\ a \Rightarrow LengthList\ a\ Int$$

which can then be improved (using the functional dependency) to:

$$LengthList\ (Succ\ Zero)\ Int$$

c) Give instance declarations such that we can add *Succ*s and *Zero*s on the type-level. Hint: try to find an inductive definition on the value-level first.

On the value-level, add can be defined as (assuming that *Succ* and *Zero* are value constructors):

$$\begin{aligned}(Succ\ n_1)\ \text{`}add\text{`}\ n_2 &= Succ\ (n_1\ \text{`}add\text{`}\ n_2)\\ Zero\quad \text{`}add\text{`}\ n_2 &= n_2\end{aligned}$$

On the type-level, this corresponds to the following instances:

instance *Add Zero t t*
instance *Add t1 t2 t3* ⇒ *Add* (*Succ t1*) *t2* (*Succ t3*)

Because the type class *Add* has no class methods, its instance declarations are also empty.

d) Consider the following two conversion functions:

$$\begin{aligned}withLength &:: [a] \rightarrow LengthList\ n\ a\\ withoutLength &:: LengthList\ n\ a \rightarrow [a]\end{aligned}$$

Give a definition for these two functions: also take the constructor function *Append* into account. If you feel that one (or two) of these functions cannot be defined, then motivate your judgement instead.

Although we could define

$$withLength = foldr\ Cons\ Nil$$

this definition will not type-check. In fact, we are not able to construct a *LengthList* of an arbitrary length (remember that the type variable *n* in *withLength*'s type signature is universally quantified). The other way around is straightforward though:

$$\begin{aligned}withoutLength\ Nil &= [\,]\\ withoutLength\ (Cons\ x\ xs) &= x : withoutLength\ xs\\ withoutLength\ (Append\ xs\ ys) &= withoutLength\ xs +\!\!+\ withoutLength\ ys\end{aligned}$$

e) The *Prelude*'s function *head* throws an exception when applied to the empty list. We could provide a similar function for *LengthList*. Because the length of the list is statically known (it is part of the type), we can define a function *safeHead* that returns the first element for a non-empty list, and results in a *type error* when applied to *Nil* or *Append Nil Nil*. Give a type signature and a definition for this function.

$$\begin{aligned}&safeHead :: LengthList\ (Succ\ n)\ a \rightarrow a\\ &safeHead = head \circ withoutLength\end{aligned}$$

5 Stream Functions (20 POINTS)

Recall the arrow of *stream functions*:

$$\textbf{newtype}\ SF\ a\ b = SF\{runSF :: [a] \to [b]\}$$

This arrow supports all operations of the *Arrow* and the *ArrowLoop* type classes. For your convenience, here is a list of available combinators for composing stream functions:

$$\begin{array}{ll}
arr & :: Arrow\ arr \Rightarrow (a \to b) \to arr\ a\ b \\
(>>>) & :: Arrow\ arr \Rightarrow arr\ a\ b \to arr\ b\ c \to arr\ a\ c \\
first & :: Arrow\ arr \Rightarrow arr\ a\ b \to arr\ (a, c)\ (b, c) \\
second & :: Arrow\ arr \Rightarrow arr\ b\ c \to arr\ (a, b)\ (a, c) \\
(***) & :: Arrow\ arr \Rightarrow arr\ a\ c \to arr\ b\ d \to arr\ (a, b)\ (c, d) \\
(\&\&\&) & :: Arrow\ arr \Rightarrow arr\ a\ b \to arr\ a\ c \to arr\ a\ (b, c)
\end{array}$$

$$loop \quad :: ArrowLoop\ arr \Rightarrow arr\ (a, c)\ (b, c) \to arr\ a\ b$$

In addition to these arrow combinators, we provide two more utility functions: *delay* for delaying a stream by one element, and *plus* for adding two values of type *Int*.

$$\begin{array}{l}
delay :: a \to SF\ a\ a \\
delay\ x = SF\ (x{:})
\end{array}$$

$$\begin{array}{l}
plus :: Arrow\ arr \Rightarrow arr\ (Int, Int)\ Int \\
plus = arr\ (uncurry\ (+))
\end{array}$$

These are all the ingredients needed to construct the cyclic machine depicted in Figure 1. In Haskell, this machine is given the type *SF Int Int*.

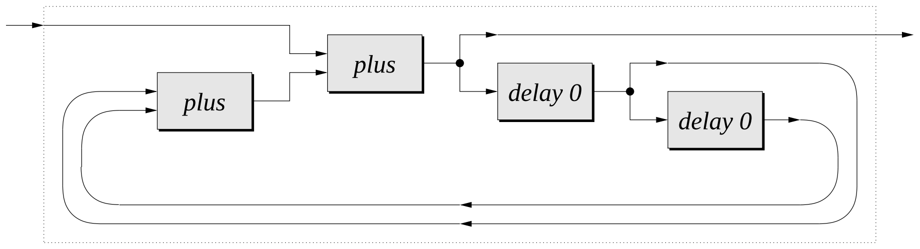


Figure 1: A stream function with two *plus* and two *delay* arrows.

a) Suppose we supply the stream (*repeat* 1) to this machine. Then what are the first five elements of the output stream? Indicate briefly how you arrived at your answer.

Consider the input wire (*repeat* 1), and the two wires used as input for the leftmost *plus* arrow. The values of the output wire can be computed by taking the sum of the three wires (for each position). Wire 1 and wire 2 are equivalent to the output wire, except for a delay of 1 or 2, respectively.

$$\begin{array}{lllllllll}
\mathit{input} & : 1 , & 1 , & 1 , & 1 , & 1 & , 1 & , & ... \\
\mathit{wire}\ 1 & : 0 , & 1 , & 2 , & 4 , & 7 & , 12 & , & ... \\
\mathit{wire}\ 2 & : 0 , & 0 , & 1 , & 2 , & 4 & , 7 & , & 12 \\
\mathit{output} & : 1 , & 2 , & 4 , & 7 , & 12 , & ... & &
\end{array}$$

We can check our answer by running the machine to be defined for the next question:

$$\begin{array}{l}
\mathit{take}\ 5\ (\mathit{runSF}\ \mathit{machine}\ (\mathit{repeat}\ 1)) \\
\quad = [1, 2, 4, 7, 12]
\end{array}$$

b) Define the machine of Figure 1 using the arrow combinators.

$$\begin{array}{l}
\mathit{machine} :: \mathit{SF}\ \mathit{Int}\ \mathit{Int} \\
\mathit{machine} = \mathit{loop}\ (\mathit{second}\ \mathit{plus} \ggg \mathit{plus} \ggg \\
\qquad\qquad\qquad \mathit{arr}\ \mathit{id}\ \&\&\&\ (\mathit{delay}\ 0 \ggg \mathit{arr}\ \mathit{id}\ \&\&\&\ \mathit{delay}\ 0))
\end{array}$$

c) By choosing the right input stream for our machine, we can compute the sequence of Fibonacci numbers:

$$0 : 1 : 1 : 2 : 3 : 5 : 8 : 13 : 21 : 34 : ...$$

Define the infinite list

$$\mathit{fibs} :: [\mathit{Int}]$$

that returns all Fibonacci numbers by running the stream function you have defined for question **b)**.

$$\mathit{fibs} = \mathit{runSF}\ \mathit{machine}\ (0 : 1 : \mathit{repeat}\ 0)$$

d) The sequence of Fibonacci numbers can be computed by adding the two previous Fibonacci numbers. This corresponds nicely with the fact that our machine contains two *delay* and two *plus* arrows. We will generalize our machine. Write a recursive function *delays* that takes an integer n and then composes n *delay* components similar to the machine in Figure 1. This function should have the following type:

$$\mathit{delays} :: \mathit{Int} \rightarrow \mathit{SF}\ \mathit{Int}\ [\mathit{Int}]$$

All lists in the output stream should have length n ("the number of wires used for feedback"). Hint: this observation may help you to define the base case for *delays*.

$$\begin{array}{l}
\mathit{delays}\ 0 = \mathit{arr}\ (\mathit{const}\ [\,]) \\
\mathit{delays}\ n = \mathit{delay}\ 0 \ggg \mathit{arr}\ \mathit{id}\ \&\&\&\ \mathit{delays}\ (n-1) \ggg \mathit{arr}\ (\mathit{uncurry}\ (:))
\end{array}$$

e) Define the function

$$\mathit{genMachine} :: \mathit{Int} \rightarrow \mathit{SF}\ \mathit{Int}\ \mathit{Int}$$

that generalizes the stream function shown in Figure 1.

$$\mathit{genMachine}\ n = \mathit{loop}\ (\mathit{second}\ (\mathit{arr}\ \mathit{sum}) \ggg \mathit{plus} \ggg \mathit{arr}\ \mathit{id}\ \&\&\&\ \mathit{delays}\ n)$$

Note that *genMachine* 2 is equivalent to *machine* defined for question **b)**.