A Client for the Z21 Model Railway Control

An exercise in Haskell

Sven Eric Panitz
Hochschule RheinMain

This document contains the implementation of a Z21 modell railway control client. The implementation is completely done in Haskell.

This project started as a little exercise in Haskell. The goal is to control a digital model railway. As server a z21 control unit is used. Z21 and z21 are products of Modelleisenbahn GmbH. They can be pruchased by Roco and Fleischmann model railways.

The z21 control unit communicates with clients by means of UDP. The here presented program is able to control locomotives. Furthermore it is possible to program automatic sequences as trains commuting between to points.

Contents

Chapter 1

Introduction

1.1 Motivation

The two goals of this project are: writing a Haskell application and having fun with model railways.

For the first goal: I have known Haskell as programming language from the very beginning in the early nineties, i.e. the time before the arrival of monads. I have done quite a lot of programming in Haskell during the nineties. Then for about the last 15 years I had to focus on mainstream languages. (My last serious activity with Haskell was a small tutorial for hopengl [Pan03]). I worked on larger Java projects and teach Java in introductory courses. With this strong background on object oriented languages I was interested on how it feels, to write an application in Haskell. An application that I typically would have written in a language as Java. Or at least in something like Scala.

This project is rather small. However it has a lot of aspects that make it interesting. It contains a graphical user interface. It entails network communication. It needs global state information. It needs low level bit operations for binary data. And it addresses questions of concurrency. The only aspects it lacks is some persistency layer.

I was interested in the question: How are these things solved in a lazy evaluated programming language. How can typical object oriented patterns be implemented in Haskell?

1.2 Overview

This documention contains the complete source code of the project. We give a short overview of the modules:

- The first module presented `Z21.Protocoll` contains the communication layer of the project. It provides code to send and recieve UDP packages to and from the Z21 control unit.

- The module `Z21.State` contains a data structure for the global state of the Z21 client. It is information on which loco address is running in which direction at which speed. this module currently is rather inmature.
- A GUI component is provided by the module `Z21.Gui`. As GUI library gtk2hs is used.
- Some means to react to incoming messages are provided in the module `Z21.MessageEventListener`. It tries to implement some pattern for event driven actions.
- A dedicated module for programming automatically commuting trains is given by `Z21.Commuting`.
- The main function for starting the GUI of the project is given in module `Client`.
- Eventually we create a scripting language. The language enables the user to program arbitrary automatic sequences for a railway layout. A parser for the script language is implemented and scripts can be started from the command line. No GUI component is provided for this part of the project.

1.3 Related Works

A mature open source library for model train control exists: JMRI. JMRI stands for: Java modell railroad interface. [Com97] Quite a number of applications use this library. It is a huge project. It contains among a lot of other things an implementation of the Z21 protocoll. As the name says: it is implemented in the programming language Java.

Chapter 2

The Z21 Protocoll

This module implements communication with the Z21 server via User Datagram Protocol (UDP). The protocoll for communication with the Z21 server is defined in [Gmb13]. This module represents the transport layer of our application.

For communication we need the Haskell `ByteString` module. We need unsigned words as provided by the module `Data.Word` and do some bit manipulations as provided by the module `Data.Bits`. Therefore the following imports are done.

```
module Z21.Protocoll where

import qualified Data.ByteString as BS
import Network.Socket.ByteString

import Data.Bits
import Data.Word
```

2.1 Messages

First of all we define a data type for the Z21 messages. Every message is represented by a constructor. Some messages have further information like the address of a locomotive or the speed for a locomotive. Some messages have complex information. In these cases the record syntax with named fields is used.

```
data Message
 = LAN_GET_SERIAL_NUMBER
   |LOGOFF
   |LAN_X_GET_VERSION
   |LAN_X_GET_STATUS
   |LAN_X_SET_TRACK_POWER_OFF
   |LAN_X_SET_TRACK_POWER_ON
   |LAN_X_BC_TRACK_POWER_ON
   |LAN_X_BC_TRACK_POWER_OFF
```

```
    |LAN_X_SET_STOP
    |LAN_X_GET_LOCO_INFO Int
    |LAN_X_SET_LOCO_DRIVE
       {locID::Int, steps::Word8, speed::Word8, direction::Direction}
    |LAN_X_SET_LOCO_FUNCTION
       {locID::Int, switch::FunctionSwitch, index::Word8}
    |LAN_X_GET_FIRMWARE_VERSION
    |LAN_GET_BROADCASTFLAGS
    |LAN_SET_BROADCASTFLAGS{ general::Bool,rbus::Bool,systemState::Bool}
    |LAN_GET_LOCOMODE Int
    |LAN_SET_LOCOMODE {locId:: Int, mode:: LOC_MODE}
    |LAN_GET_HWINFO
    |LAN_X_LOCO_INFO
     {locID::Int
     ,busy::Bool
     ,stpes::SpeedSteps
     ,direction::Direction
     ,speed::Word8
     ,doubleTraction::Bool
     ,smartSearch::Bool
     ,light::Bool
     ,f1::Bool
     ,f2::Bool
     ,f3::Bool
     ,f4::Bool
     }
    |LAN_RMBUS_GETDATA Word8
    |LAN_RMBUS_DATACHANGED [Word8]
    |SERIAL_NUMBER
    |LAN_X_SET_TURNOUT Word16 Bool Bool
    |LAN_X_GET_TURNOUT_INFO Word16
    |LAN_X_TURNOUT_INFO  Word16 Word8
    |LAN_X_UNKNOWN_COMMAND
    |UNKNOWN [Word8]
       deriving Show
```

Some more data types are used within this definition. First of all a simple enum type for the direction of a locomotive:

```
data Direction = Forward | Backward deriving (Eq,Show,Read,Enum)

switchDirection Forward = Backward
switchDirection Backward = Forward

isForward  = (==)Forward
```

We need a type to denote the protocoll of a locomotives decoder. Currently two different decoder formats are known. The standard dcc format and motorolas mm format.

```
data LOC_MODE = DCC|MM  deriving (Show,Eq)
isDCC = (==)DCC
```

For switches (e.g. light) there are three commands. Turning it off or on and simple switching it.

```
data FunctionSwitch = On|Off|Switch deriving (Show,Eq)

functionSwitchCode:: FunctionSwitch -> Word8
functionSwitchCode On = 0x00
functionSwitchCode Off = 0x40
functionSwitchCode Switch = 0x80
```

Finally there are three different types for decoder speed selection: 14, 28 and 128 steps.

```
data SpeedSteps = S14|S28|S128 deriving (Show,Eq)
```

Currently the implementation does not realy bother about these three different types. We alway assume 128 steps.

2.2 Sending

This paragraph contains the implementation of function `mkmessage` for serializing Z21 messages as a byte string in order to send them as UDP message to the z21 server. For each message a list of 8 bit words is created. The function `pack` from module `Data.ByteString` is applied to create the byte string.

We use the hexadecimal notation for the 8 bit words directly from the specification of the protocoll [Gmb13].

```
mkMessage LAN_GET_SERIAL_NUMBER = BS.pack [0x04,0x00,0x10,0x00]
mkMessage LAN_GET_HWINFO =  BS.pack [0x04,0x00, 0x1A, 0x00]
mkMessage LOGOFF = BS.pack [0x08,0x00,0x10,0x00]
mkMessage LAN_X_GET_VERSION
  = BS.pack [0x07,0x00,0x40,0x00,0x21,0x21,0x00]
mkMessage LAN_X_GET_STATUS
  =  BS.pack [0x07,0x00,0x40,0x00,0x21,0x24,0x05]
mkMessage LAN_X_SET_TRACK_POWER_OFF
```

```
 = BS.pack [0x07,0x00,0x40,0x00,0x21,0x80,0xA1]
mkMessage LAN_X_SET_TRACK_POWER_ON
 = BS.pack [0x07,0x00,0x40,0x00,0x21,0x81,0xA0]
mkMessage LAN_X_SET_STOP = BS.pack [0x06,0x00,0x40,0x00,0x80,0x80]
mkMessage (LAN_X_GET_LOCO_INFO locID)
  = BS.pack [0x09,0x00,0x40,0x00,xheader,db0,db1,db2,xorbyte]
  where
    xheader = 0xE3
    db0 = 0xF0
    [db1,db2] = mkdLocIDBytes locID
    xorbyte = xheader `xor` db0 `xor` db1 `xor` db2
mkMessage LAN_X_SET_LOCO_DRIVE
              {locID=locID, steps=st, speed=sp, direction=dir}
 = msg_LAN_X_SET_LOCO_DRIVE locID st sp dir
mkMessage LAN_X_SET_LOCO_FUNCTION
              {locID=locID, switch=switch, index=index}
 = msg_LAN_X_SET_LOCO_FUNCTION locID switch index
mkMessage LAN_X_GET_FIRMWARE_VERSION
  = BS.pack [0x07,0x00,0x40,0x00,0xF1,0x0A, 0xFB]
mkMessage LAN_GET_BROADCASTFLAGS = BS.pack [0x04,0x00, 0x51, 0x00]
--Vorsicht. Hier ist die Dokumentation sehr irreführend
-- jaja little endian. Dann schreibt das auch so auf!
mkMessage LAN_SET_BROADCASTFLAGS
              {general=general,rbus=rbus,systemState=systemState}
 = BS.pack ([0x08,0x00, 0x50, 0x00, byte,sysByte,0x00,0x00])
   where
     genByte = if general then 0x01::Word8 else 0
     rbusByte = if rbus then 0x02::Word8 else 0
     sysByte = if systemState then 0x01::Word8 else 0
     byte = genByte .|. rbusByte
mkMessage (LAN_GET_LOCOMODE locID)
  = BS.pack ([0x06,0x00, 0x60, 0x00]++mkdLocIDBytes locID)
mkMessage LAN_SET_LOCOMODE {locId=id, mode=md}
 = msg_LAN_SET_LOCOMODE id md
mkMessage (LAN_RMBUS_DATACHANGED bs)
  = BS.pack ([0x0F,0x00,0x80,0x00]++bs)
mkMessage (LAN_RMBUS_GETDATA b) = BS.pack [0x05,0x00, 0x81,0x00,b]
mkMessage (LAN_X_SET_TURNOUT address1 active first)
  = BS.pack [0x09,0x00,0x40,0x00,xheader,db0,db1,db2,xorbyte]
  where
    address = address1+3
    xheader = 0x53
    db0 = fromIntegral$shiftR address 8
    db1 = fromIntegral address
    db2 = (0x80::Word8)
          .|. (if active then (0x08::Word8) else 0x00)
          .|. (if first then (0x01::Word8) else 0x00)
    xorbyte = xheader `xor` db0 `xor` db1 `xor` db2
mkMessage (LAN_X_GET_TURNOUT_INFO address)
  = BS.pack [0x08,0x00,0x40,0x00,0x43,db0,db1,0x43`xor`db0`xor`db1]
  where
    db0 = fromIntegral$shiftR address 8
```

```
    db1 = fromIntegral address
mkMessage (LAN_X_TURNOUT_INFO address value)
  = BS.pack [0x09,0x00,0x40,0x00,0x43,db0,db1,db2,xorbyte]
 where
    db0 = fromIntegral$shiftR address 8
    db1 = fromIntegral address
    db2 = value
    xorbyte = 0x43`xor`db0`xor`db1`xor`db2
mkMessage (UNKNOWN _) = BS.pack [0x40,0x61,0x82]
```

Thus messages can be send as UDP to the server.

```
sendMsg socket addr msg = do
  putStrLn ("> "++show msg)
  sendTo socket (mkMessage msg) addr
```

2.3 Receiving

When receiving a byte string we will read it as a Z21 message. Thus we write the inverse function `readMessage`. The following equation should hold:

$$id = \text{readMessage} \circ \text{mkMessage}$$

The given implementation uses pattern matching on the list of bytes.[1]

```
readMessage = rM.BS.unpack
  where
   rM :: [Word8] -> Message
   rM [0x04,0x00,0x10,0x00] = LAN_GET_SERIAL_NUMBER
   rM [0x04,0x00, 0x1A, 0x00] = LAN_GET_HWINFO
   rM [0x08,0x00,0x10,0x00] = LOGOFF
   rM [0x07,0x00,0x40,0x00,0x21,0x21,0x00] = LAN_X_GET_VERSION
   rM [0x07,0x00,0x40,0x00,0x21,0x24,0x05] = LAN_X_GET_STATUS
   rM [0x07,0x00,0x40,0x00,0x21,0x80,0xA1] = LAN_X_SET_TRACK_POWER_OFF
   rM [0x07,0x00,0x40,0x00,0x21,0x81,0xA0] = LAN_X_SET_TRACK_POWER_ON
   rM [0x07,0x00,0x40,0x00,0x61,0x01,0x60] = LAN_X_SET_TRACK_POWER_ON
   rM [0x07,0x00,0x40,0x00,0x61,0x00,0x61] = LAN_X_SET_TRACK_POWER_OFF
   rM [0x06,0x00,0x40,0x00,0x80,0x80] = LAN_X_SET_STOP
   rM [0x09,0x00,0x40,0x00,0xE3,0xF0,0xE3,db1,db2,xorbyte]
       = LAN_X_GET_LOCO_INFO (mkLocoInfo db1 db2)
   rM [0x0A,0x00,0x40,0x00,0xE4,0xF8,db1,db2,db3,xorbyte]
```

[1] I am sure there is a much smarter way to implement this. The list of bytes coding a message is written in both functions `mkMessage` and `readMessage`. This seems to be error prone.

```
    = LAN_X_SET_LOCO_FUNCTION
        { locID = (mkLocoInfo db1 db2)
        , switch = if (db3 .&. 0xC0 == 0) then Off
                   else if (db3 .&. 0xC0 == 0xC0) then On
                        else Switch
        , index = db3 .&. 0x3F}
  rM [0x0A,0x00,0x40,0x00,0xE4,db0,db1,db2,db3,xorbyte]
    = LAN_X_SET_LOCO_DRIVE
        { locID = (mkLocoInfo db1 db2)
        , steps = db0 `mod` 16
        , speed = db3 `mod`128
        , direction = if db3 >= 128 then Forward else Backward}
  rM [0x07,0x00,0x40,0x00,0xF1,0x0A, 0xFB]
    = LAN_X_GET_FIRMWARE_VERSION
  rM [0x04,0x00, 0x51, 0x00] = LAN_GET_BROADCASTFLAGS
  rM [0x06,0x00, 0x60, 0x00,hi,lo]
    = LAN_GET_LOCOMODE (  (fromIntegral hi)*256 + (fromIntegral lo))
  rM [0x07,0x00,0x61,0x00,locIDH,locIDL,mode]
    = LAN_SET_LOCOMODE
        { locId =   (fromIntegral locIDH)*256
                  + (fromIntegral locIDL)
        , mode = if mode==0 then DCC else MM}
  rM (l:0x00:0x40:0x00:0xEF:db0:db1:db2:db3:db4:_)
    = LAN_X_LOCO_INFO
        { locID = (mkLocoInfo db0 db1)
        , busy = db2 .&. 0x08 == 0x08
        , stpes
           = let step = db2 .&. 0x07
             in if step == 0 then S14
                else if step == 1 then S28
                     else S128
        , direction = if db3 .&. 0x80 == 0x80
                      then Forward else Backward
        , speed =  db3 .&. 0x7F
        , doubleTraction = db4 .&. 0x40 == 0x40
        , smartSearch = db4 .&. 0x20 == 0x20
        , light  = db4 .&. 0x10 == 0x10
        , f1 = db4 .&. 0x01 == 0x01
        , f2 = db4 .&. 0x02 == 0x02
        , f3  = db4 .&. 0x04 == 0x04
        , f4 = db4 .&. 0x08 == 0x08
        }
  rM (0x0F:0x00:0x80:0x00:bytes) = LAN_RMBUS_DATACHANGED bytes
  rM (0x08:0x00:0x10:0x00:serialNumber) = SERIAL_NUMBER
  rM (0x09:0x00:0x40:0x00:0x43:db0:db1:db2:_:[])
    = LAN_X_TURNOUT_INFO ((mkLocoInfo db0 db1)+1) db2
  rM (0x0F:0x00:0x40:0x00:0x61:0x82:0xE3:[]) = LAN_X_UNKNOWN_COMMAND
  rM bytes = UNKNOWN  bytes

  mkLocoInfo db1 db2
    =   (fromIntegral (db1 .&. 0x3F)) * 2^8
      + (fromIntegral db2)
```

Some of the more complicated messages are done in seperate functions.

The message for setting a locomotive system mode.

```
msg_LAN_SET_LOCOMODE:: Int -> LOC_MODE -> BS.ByteString
msg_LAN_SET_LOCOMODE locID locMode
  = BS.pack [0x07,0x00, 0x61,0x00
            ,fromIntegral (locID `div` 256)
            ,fromIntegral (locID `mod` 256)
            ,if isDCC locMode then 0 else 1]
```

The message for getting a locomotive to drive in a certain direction by a certain speed.

```
msg_LAN_X_SET_LOCO_DRIVE locID steps speed direction
  = BS.pack [0x0A,0x00,0x40,0x00,xheader,db0,db1,db2,db3,xorbyte]
    where
      xheader = 0xE4
      db0 = 16+steps
      [db1,db2] = mkdLocIDBytes locID
      db3 = (if isForward direction then 128 else 0)+speed
      xorbyte = xheader `xor` db0 `xor` db1 `xor` db2 `xor` db3
```

The message for switching a locomotive's function.

```
msg_LAN_X_SET_LOCO_FUNCTION locID switch index
  = BS.pack [0x0A,0x00,0x40,0x00,xheader,db0,db1,db2,db3,xorbyte]
    where
      xheader = 0xE4
      db0 = 0xF8
      [db1,db2] = mkdLocIDBytes locID
      db3 = functionSwitchCode switch + index
      xorbyte = xheader `xor` db0 `xor` db1 `xor` db2 `xor` db3
```

The decoder address of a locomotive is decoded in two bytes in the following way.

```
mkdLocIDBytes:: Int -> [Word8]
mkdLocIDBytes locID =
  [fromIntegral (locID `div` 256 `mod` (128+64))
  ,fromIntegral (locID `mod` 256)]
```

Chapter 3

Global State

This chapter contains the module `State`. It is the model of the application. The client will keep a global state. This state can be controlled through a graphical user interface. Furthermore the state can get modified through incoming messages from the z21 server. This will be the case, when other clients control trains.

```
module Z21.State where
import Z21.Protocoll
  (Direction(..),SpeedSteps(..),switchDirection)
import Data.Word
```

3.1 Data Types

The global state is basically a list of locomotive states. One locomotive is the currently controlled locomotive. This will not be included in the list of locomotives.

```
data State = Z21 {currentLoco::Loco,locos::[Loco]}
```

A locomotive state is represented by its address, speed, direction, steps for speed and its light status.[1]

```
data Loco = Loco
            { lid::Int
            , address::Int
            , steps::SpeedSteps
            , speed::Word8
            , direction::Direction
```

[1] We have a field for the address and a further field for some ID. Howerever, currently both are redundantely used. Maybe some day it might be nice to have a database of locomotives with own IDs and some description.

```
               , light::Bool} deriving (Eq,Show)
```

3.2 Construction

Two functions are provided to create states.

```
newState = let (l:ls) = map newLoco [1..20]
           in Z21{currentLoco=l,locos=ls}

newLoco lid
  = Loco
      { lid=lid
      , address = lid
      , steps = S128
      , speed=0
      , direction=Forward
      , light=True}

newLocoInState st lid = st{locos=newLoco lid:locos st}
```

3.3 Getter Functions

Some convenient getter functions to retrieve values are provided.

```
getDirectionOfLoco locoid st
  |locoid == (lid$currentLoco st) = direction$currentLoco st
  |otherwise = direction$head$filter (\loco->lid loco==locoid) $locos st

getAddress = address.currentLoco
getDirection  = direction.currentLoco
getSpeed   = speed.currentLoco
getLight = light.currentLoco

getLoco id [] = Nothing
getLoco id (l:ls)
  |id == lid l = Just l
  |otherwise = getLoco id ls
```

The following function is used to change the currently activ locomotive. It creates a new state. If the locomotive with the corresponding ID does not exist in the state, then a new locomotive is created.

```haskell
selectLoco locoid st
  |locoid == (lid$currentLoco st) = st
  |Nothing==loco = st{currentLoco=newLoco locoid
                     ,locos=(currentLoco st):locos st}
  |otherwise = let (Just (locs,loc)) = loco
               in st{currentLoco=loc,locos=(currentLoco st):locs}
  where
    loco = getLoco [] (locos st)

    getLoco locos [] = Nothing
    getLoco locos (l:ls)
      |lid l == locoid = Just (locos++ls,l)
      |otherwise = getLoco (l:locos) ls
```

3.4 Setter Functions

In this section some setter functions are defined. Since in Haskell we cannot modify any data, these functions transform the state and return a new state-

```haskell
setDirection dir st = st{currentLoco=(currentLoco st){direction=dir}}
setSpeed sp st = st{currentLoco=(currentLoco st){speed=sp}}
setLight l st = st{currentLoco=(currentLoco st){light=l}}

replaceNonActiveLoco loco@Loco{lid=id} st@Z21{locos=ls}
  = st{locos=map (\l->if lid l == id then loco else l) ls}

changeDirectionOfLoco locoid st@Z21{currentLoco=current,locos=ls}
  |locoid == lid current
   = st{currentLoco=changeDirectionInLoco current}
  |otherwise = st{locos=map
                        (\loco-> if (lid loco==locoid)
                                 then loco
                                 else changeDirectionInLoco loco) ls}

setDirectionOfLoco locoid dir st@Z21{currentLoco=current,locos=ls}
  |locoid == lid current
   = st{currentLoco=current{direction=dir}}
  |otherwise = st{locos=map
                        (\loco-> if (lid loco/=locoid)
                                 then loco
                                 else loco{direction=dir}) ls}

```

```

changeDirectionInLoco l = l{direction=switchDirection$direction l}
```

Chapter 4

Graphical user interface

The module `Gui` contains the definition of the graphical user interface of the application. As GUI-library the gtk2hs-library is used. It is a direct Haskell api for the *Gtk+* library. A basic tutorial can be found in [vT08].

Some useful hints on gtk2hs and threads can be found on the internet in an article by Daniel Wagner [Wag15].

```haskell
module Z21.Gui where
import Z21.Protocoll hiding(light ,direction , speed)
import Z21.State
import Util

import Data.Word

import Control.Concurrent

import Graphics.UI.Gtk
```

4.1 Controls

A data type is defined, which contains all the gui controls of the client application. These controls may change their values due to modifications of the global state.

```haskell
data GuiControls a
 = GuiControls
   { mainBox            :: VBox
   , statusLabel        :: TextView
   , addrLabel          :: Label
   , speedAdjust        :: Adjustment
   , addrSelect         :: [Button]
   , lightButton        :: ToggleButton
   , directionControl   :: [RadioButton]
   , powerControl       :: [RadioButton]
```

```
}
```

4.1.1 Construction

We define a straighforward constructor function for the gui controls.

```
newGuiControls = do
  mainPanel     <- vBoxNew False 10
  statusLabel   <- textViewNew
  addrLabel     <- labelNew$Just$show 1
  speedAdjust   <- adjustmentNew 0.0 0.0 128.0 1 1.0 1.0
  lightButton   <- toggleButtonNewWithLabel "Light On/Off"
  forwardButton<- radioButtonNewWithLabel$show Forward
  backButton    <-
    radioButtonNewWithLabelFromWidget forwardButton $show Backward
  addrButtons   <- sequence $map (buttonNewWithLabel.show) [1 .. 21]
  onButton      <- radioButtonNewWithLabel "Power On"
  offButton     <-
    radioButtonNewWithLabelFromWidget onButton "Power Off"
  textViewSetWrapMode statusLabel WrapChar
  widgetSetSizeRequest statusLabel (-1) 180
  return
    GuiControls
      { mainBox          = mainPanel
      , statusLabel      = statusLabel
      , addrLabel        = addrLabel
      , speedAdjust      = speedAdjust
      , addrSelect       = addrButtons
      , lightButton      = lightButton
      , directionControl = [forwardButton,backButton]
      , powerControl     = [onButton,offButton]
      }
```

4.2 Updates

When in the global state a new current locomotive is set, the gui controls need to be updated. this can be achieved by use of the following function.

```
updateGUI Loco{lid=lid,speed=sp,direction=dir,light=li} gui = do
  if (Forward==dir)
    then toggleButtonSetActive (head$directionControl gui) True
    else toggleButtonSetActive (head$tail$directionControl gui) True
  adjustmentSetValue (speedAdjust gui)$fromInteger$toInteger sp
```

```
  toggleButtonSetActive (lightButton gui) li
  labelSetText (addrLabel gui) (show lid)
```

4.3 Layout

This section the controls of the application GUI and put them together with some layout. Propably it would have been better to use the GUI builder tool glade in the first place rather than doing everything manually. However, here we go.

The first function is an auxilary function to create the layout for a list of radio buttons:

```
mkLayoutRadioButtons (b1:bs) = do
  mainbox <- vBoxNew False 0
  box1    <- hBoxNew False 0
  box2    <- hBoxNew False 10
  containerSetBorderWidth box2 10
  boxPackStart box1 box2 PackNatural 0
  boxPackStart box2 b1 PackNatural 0
  sequence$map (\b -> boxPackStart box2 b PackNatural 0) bs
  boxPackStart mainbox box1 PackNatural 0
  return mainbox
```

The next function creates a layout for the address selection buttons of the clients gui. They are placed in rows of three.

```
mkLayoutAddrSelect gui = do
  let buttons = addrSelect gui
  let adj = speedAdjust gui
  lines  <-sequence$take (length buttons `div` 3)$repeat hButtonBoxNew
  sequence_
      $ map (\(bb,bs)-> set bb [ containerChild := b| b <- bs ])
      $ zip lines
      $ splitNChunks (length buttons `div` length lines) buttons
  vbox <- vButtonBoxNew
  set vbox [ containerChild := l| l <- lines ]
  return  vbox
```

The following function creates a layout for the speed control, light switch, direction and the address selection.

```
mkLayoutLocoGui gui = do
  let adj1 = speedAdjust gui
  forwardBackwardButton <- mkLayoutRadioButtons (directionControl gui)

  let lightB = lightButton gui

  box1 <- hBoxNew False 0
  vsc  <- vScaleNew adj1
  box2 <- vBoxNew False 0
  boxPackStart box1 box2 PackGrow 0

  hsc1 <- hScaleNew adj1
  boxPackStart box2 hsc1 PackGrow 0

  mainBox <- vBoxNew False 10

  addrLBox <- hBoxNew False 10
  lab <- labelNew$Just "Current Loco Address: "
  boxPackStart addrLBox lab PackGrow 0
  boxPackStart addrLBox (addrLabel gui) PackGrow 0
  boxPackStart mainBox addrLBox PackGrow 0

  boxPackStart mainBox forwardBackwardButton PackGrow 0
  boxPackStart mainBox lightB PackGrow 0

  lSel <- labelNew$Just "Speed"
  boxPackStart mainBox lSel PackGrow 0
  boxPackStart mainBox box1 PackGrow 0

  lSel <- labelNew$Just "Loco Address Selection"
  boxPackStart mainBox lSel PackGrow 0
  addrSelect <- mkLayoutAddrSelect gui
  boxPackStart mainBox addrSelect PackGrow 0

  return mainBox
```

Putting everything together and adding the power control and the status display to the layout:

```
createOverallLayout gui = do
  let mainb = mainBox gui

  let onOffControl = powerControl gui
  onOffButtons <- mkLayoutRadioButtons onOffControl

  statusBox <- scrolledWindowNew Nothing Nothing
  widgetSetSizeRequest statusBox (-1) 80
  scrolledWindowAddWithViewport statusBox (statusLabel gui)

```

```
  boxPackStart mainb onOffButtons PackGrow 0

  locoPanel <- mkLayoutLocoGui gui
  boxPackStart mainb locoPanel PackGrow 0
  lSel <- labelNew$Just "Received Messages"
  boxPackStart mainb lSel PackGrow 0
  boxPackStart mainb statusBox PackGrow 0
```

4.4 Event Handler

In this section we will add event handlers to the controls of the GUI.

The first function is a utility function, that adds event handlers to a list of pairs of buttons and events.

```
addEventsRadioButtons las@((b1,_):_) = do
  toggleButtonSetActive b1 True
  sequence_$map (\(b,a)-> onToggled b (a >> return ())) las
```

Events for the buttons in the address selection control:

```
addEventAddrSelect gui state sendF = do
  let adj = speedAdjust gui
  sequence$map
   (\b ->
     onClicked b$ do
       nrl <- buttonGetLabel b
       let nr = read nrl
       sendF $LAN_X_GET_LOCO_INFO nr
       modifyMVar_ state (return.selectLoco nr)
       s <- readMVar state
       let loco = currentLoco s
       updateGUI loco gui
   )
   (addrSelect gui)
```

Events for further controls. First the event for the direction switch:

```
addGuiEvents gui state send =  do
  let adj1 = speedAdjust gui
  addEventsRadioButtons
      $map (\x->(x, do
```

```
            label <- buttonGetLabel x
            let dir = (read label)::Direction
            modifyMVar_ state (return.setDirection dir)
            s <- readMVar state
            let loco = currentLoco s
            adjustmentSetValue adj1 0
            send $LAN_X_SET_LOCO_DRIVE (address loco) 2 0 dir
           ))
           (directionControl gui)
```

Event for the light switch:

```
let lightB = lightButton gui
onClicked lightB $ do
  s <- readMVar state
  let loco = currentLoco s
  activ <- toggleButtonGetActive lightB
  let x = if activ then On else Off
  modifyMVar_ state (return.setLight activ)
  send LAN_X_SET_LOCO_FUNCTION{locID=address loco,switch=x,index=0}
  return ()
```

Event for speed adjustment:

```
onValueChanged adj1 $ do
  s <- readMVar state
  let loco = currentLoco s
  val <- adjustmentGetValue adj1
  let v = (truncate val)::Word8
  modifyMVar_ state (return.setSpeed v)
  send$LAN_X_SET_LOCO_DRIVE
       (address loco) 2 v (Z21.State.direction loco)
  return ()
```

The event for the power switch:

```
addEventsRadioButtons$
    zip (powerControl gui)
      [send LAN_X_SET_TRACK_POWER_ON
      ,send LAN_X_SET_TRACK_POWER_OFF]
```

```
addEventAddrSelect gui state send
```

4.5 Overall Window Creation

Eventually we provide a function to build everything, add events and display it in a window.

```
createGuiWindow gui state send = do
  -- Create a new window
  window <- windowNew

  -- Sets the border width of the window.
  set window [ containerBorderWidth := 10 ]

  createOverallLayout gui

  -- sets the contents of the window
  set window [ containerChild := mainBox gui ]

  addGuiEvents gui state send

  return window
```

Chapter 5

Message handling

We are receiving messages from the Z21 control. Messages will notify contacts on curcuit tracks, new values for turn outs or locos. In this module we define means to program reaction to recieved messages.

```
module Z21.MessageEventListener where
import Z21.Protocoll hiding(light,direction, speed)
import qualified Z21.Protocoll as Z21(light,direction, speed)
import Z21.State
import Z21.Gui

import Control.Monad (forever)
import Control.Concurrent

import Graphics.UI.Gtk

--import Network.Socket hiding (send, sendTo, recv, recvFrom)
import Network.Socket.ByteString
```

The main type is a list of message handlers. A message handlers is basically a function, which takes a messages and results an `(IO Bool)` event. This is an IO action that results in an boolean value. The boolean value signifies, if the message was of interest and had been successfully processed, e.g. there might be a message handler which waits for contact on a certain circuit track. Only a message signifying contact on this track will result in an succesfull IO operation.

Every message handler has a unique number and a boolean flag. The flag signifies, if the message handler will only be used one time successfully.

The message listener type has the list of message handlers and a number, which will be the number of the next message handler. Thus we can provide unique numbers for message handlers.

```
type MessageHandler = (Integer,Bool,Message -> IO Bool)
type MessageListener = (Integer,[MessageHandler])
```

The overall message listener will be a global state variable initialized with the empty handler list.

```
newEventListener = newMVar ((1,[])::MessageListener)
```

The main function for the global message listener will recieve the messages from some socket. It will process every message handler in he list. Afterwards every message handler which ended successful and has the flag will be deleted from the list.

```
receiveMessages eventListener socket = forever $ do
    message <-recv socket 1024
    let msg = readMessage message
    print msg
    (nr,listener)<- readMVar eventListener
    rs <-sequence $ map (\(nr,_,f) -> f msg) listener
    let toDelete = filter (\((nr,oneShot,_),shot)->oneShot && shot)
                      $zip (listener) rs
    sequence_
     (map (\((nr,_,_),_)->removeListener eventListener nr) toDelete)
```

The function above will be started in an own thread.

```
startEventListener listener socket =
  forkIO$receiveMessages listener socket
```

We provide two functions to add new message handlers to the global message listener. The result of the action is the number of the added handler.

Two versions are provided. One for handlers which will be removed after the first successful reaction, one for handlers that stay in the list.

```
addListener = addListenerAux False
addOneTimeListener = addListenerAux True

addListenerAux oneTime eventListener f = do
  modifyMVar_ eventListener
    (\(nr,fs)->return (nr+1,(nr,oneTime,f):fs))
  (nr,_)<- readMVar eventListener
  return (nr-1)
```

Of course it is possible to remove handlers again:

```
removeListener eventListener nr = do
  (_,listener)<- readMVar eventListener
  modifyMVar_
    eventListener
    (\(nmr,fs) -> return (nmr,filter (\(n,_,_) -> not (n==nr)) fs))
```

5.1 Special handlers

In this paragraph we define two general message handlers.

The first one updates the global state. Furthermore it updates the view of the global state. Since the handler functions are not evaluated in the gui thread we capsule every gui functionality within the call of the gtk2hs standard function `postGUIAsync`.

```
processMessage gui state send
  LAN_X_LOCO_INFO{locID=id ,Z21.direction=dir ,Z21.speed=sp ,Z21.light=l}
   = do
       modifyMVar_ state$return.replaceNonActiveLoco
          (newLoco id)
          {speed=sp, direction=dir, light=l}
       return True
processMessage gui state send  LAN_X_BC_TRACK_POWER_ON = do
  postGUIAsync$toggleButtonSetActive (head$powerControl gui) True
  return True
processMessage gui state send  LAN_X_BC_TRACK_POWER_OFF = do
  postGUIAsync$toggleButtonSetActive (head$tail$powerControl gui) True
  return True
processMessage _ _ _ _ = return False
```

Another function that will be evaluated whenever a message is received is solely for logging purposes. It will display the message on the status label of the GUI. Furthermore the state of the current locomotive is printed to the console.

```
logging gui state msg = do
  postGUIAsync guiStuff
  s<-readMVar state
  return True
   where
     guiStuff = do
       textViewSetEditable (statusLabel gui) False
       buffer<-textBufferNew Nothing
       textBufferInsertInteractiveAtCursor buffer (show msg) True
```

```
        textViewSetBuffer (statusLabel gui) buffer
```

Chapter 6

Automatic Commuting

A typical usecase for a modell railway display is a train commuting between two stops. In German there is the nice word: *Pendelzugautomatik*. To put this into realization we need some contacts that signifies when a train reaches a certain point. We can use circuit tracks for this purpose.

This module implements some means to program a Pendelzugautomatik. It provides a function, which will start a Pendelzugautomatik and a GUI component, to start a commuting train.

```haskell
module Z21.Commuting where

import Z21.State
import Z21.Protocoll
import Z21.MessageEventListener

import Data.Maybe

import Text.Read

import Control.Concurrent.MVar
import Control.Concurrent.Timer
import Control.Concurrent.Suspend.Lifted

import Data.Bits

import Graphics.UI.Gtk
```

The Z21 signifies contact on a circuit track with a `LAN_RMBUS_DATACHANGED` message. It uses the so called R-Bus for feedback modules. The circuit tracks are grouped, such that a contact is represented by a pair of numbers: the group and the address.

```haskell
type Contact = (Int,Int)
```

To start a Pendelzugautomatik we need basic information between which stops, which loco and the idle time at each stop. Furthermore we need global information: the event listeners and the state of the control unit. And of course the communication function with the z21 control unit.

The result is an IO action which returns the number of the event listener, that controls the Pendelzugautomatik. Thus we get the following type[1].

```
startCommuting
  ::    Contact                 -- first contact
    ->  Contact                 -- second contact
    ->  Int                     -- loco id
    ->  Int                     -- idle time
    -> MVar MessageListener     -- event listeners
    -> MVar State               -- global state
    -> (Message -> IO a1)       -- communication with Z21
    -> IO Integer               -- return the number of the event listener
```

The function `startCommuting` is basically implemented by a new event listener function. This function will evaluate `LAN_RMBUS_DATACHANGED` messages. In order to recieve these messages from the Z21 control unit, we need to send an appropiate `LAN_SET_BROADCASTFLAGS` message.

We assume that the locomotive is somewhere between the two stops. The locomotive will be started in one arbitrary direction.

A further state variable is used to remember which was the last stop the loco activated. It is initialized with the illegal contact `(-1,-1)`.

```
startCommuting leftC rightC locid delay eventListener state send = do
  lastContact <- newMVar (-1,-1)
  send LAN_SET_BROADCASTFLAGS
              {general=True,rbus=True,systemState=False}

  nr <- addListener eventListener (eval lastContact)

  send (LAN_X_SET_LOCO_DRIVE locid 2 6 Forward)

  modifyMVar_ state (\s -> return$setDirectionOfLoco locid Forward s)
  return nr
```

The main work is done by the event listener function. It has two arguments. The state variable, which contains the circuit track that has been contacted the last time. The second argument is the message to be processed.

[1] The actual derived type is more general, but for a clearer understanding it has been simplified a bit.

First of all the function needs to analyse the incoming message. Which contact is active. Wen need to have a closer look at the single bytes of the message. [2]

The first local definitions create a list of all active contacts. As a matter of fact the message may contain information on several activ contacts. Currently the implementation neglects simultanious contacts in different groups. This is simply duw to the fact that I have not enough hardware to test several groups.

```
  where
    eval lastContact (LAN_RMBUS_DATACHANGED (gi:gs) ) = do
      let adder = if gi==0 then 0 else 10
      let numbered=filter (\(_,entry)-> entry/=0)$zip [1..] gs
      if (null numbered) then return False else do
      let changedGroup = adder+fst(head numbered)
      let groupVal = snd(head numbered)
      let addrs = filter (\(_,code)-> code == groupVal.&.code)
                    $zip
                     [(1::Int)..]
                     [0x01,0x02,0x04,0x08,0x10,0x20,0x40,0x80]
      if (null addrs)  then return False else do
      let newContacts = map (\(x,_)->(changedGroup,x)) addrs
```

First of all we check if the last contact we reacted to is still active or again active. If this is the case the locomotive obviously has not moved very much and still triggers the contact of the same stopping. Then we do not react at all.

```
      oldContact <- readMVar lastContact
      if (elem oldContact newContacts) then return False else do
```

If none of the contacts at the two stoppings is active then no reaction is necessary.

```
      if not (elem leftC newContacts || elem rightC newContacts)
        then return False else do
```

Eventually at this point we know that the train reached the other stopping. No several things are to be done. Stop the train. Modify the global state. After the specified time of delay start the train with the switched direction. The wait is done with the timer function: `oneShotTimer`.

[2] It would have been more consequent to have done this in the module `Protocoll` and provided more structured information with the constructor `LAN_RMBUS_DATACHANGED`.

```
      let newContact = if elem leftC newContacts
                       then leftC
                       else rightC

      modifyMVar_ lastContact (\_->return newContact)

      s <- readMVar state
      let dir = switchDirection$getDirectionOfLoco locid s
      modifyMVar_ state (\s -> return$setDirectionOfLoco locid dir s)

      send (LAN_X_SET_LOCO_DRIVE locid 2 0 dir)

      oneShotTimer
        ((send$LAN_X_SET_LOCO_DRIVE locid 2 6 dir) >>return ())
        (sDelay $fromIntegral delay)
      return True
```

We are only interested in `LAN_RMBUS_DATACHANGED` messages. For all other message no reaction is necessary.

```
    eval _ _ = return False
```

Since the Pendelzugautomatik is controlled by a single event listener function the Pendelzugautomatik can be stopped by removing this listener from the global listener queue.

```
stopCommunting listenId locid eventListener send = do
  send (LAN_X_SET_LOCO_DRIVE locid 2 0 Forward)
  removeListener eventListener listenId
```

6.1 GUI

We provide a GUI component for the control of commuting trains. We apply the same pattern as in the module `Z21.Gui`. All relevant controls are collected in one type.

```
data -- BoxClass a =>
   CommutingGui a = CommutingGui
   { mainPanel          :: VBox
   , addrEntry          :: Entry
```

```
  , fstCircuitGroup   :: Entry
  , fstCircuitEntry   :: Entry
  , sndCircuitEntry   :: Entry
  , sndCircuitGroup   :: Entry
  , delayEntry        :: Entry
  , startStopControl  :: ToggleButton
  }
```

A constructor function is provided.

```
newCommutingGui = do
  mainPanel       <- vBoxNew False 10
  addEntry        <- entryNew
  fstCircuitEntry <- entryNew
  entrySetText fstCircuitEntry "1"
  sndCircuitEntry <- entryNew
  entrySetText sndCircuitEntry "2"
  fstCircuitGroup <- entryNew
  entrySetText fstCircuitGroup "1"
  sndCircuitGroup <- entryNew
  entrySetText sndCircuitGroup "1"
  delayEntry      <- entryNew
  entrySetText delayEntry "5"
  startStopButton <- toggleButtonNewWithLabel "Start/Stop"
  return
    CommutingGui
      { mainPanel        = mainPanel
      , addrEntry        = addEntry
      , fstCircuitEntry  = fstCircuitEntry
      , sndCircuitEntry  = sndCircuitEntry
      , fstCircuitGroup  = fstCircuitGroup
      , sndCircuitGroup  = sndCircuitGroup
      , delayEntry       = delayEntry
      , startStopControl = startStopButton
      }
```

6.1.1 Layout

A layout is added to the components.

```
mkLayoutCommutingGui gui = do
  lokP      <- hBoxNew False 10
  lokLabel  <- labelNew$Just "Lokadresse"
  boxPackStart lokP lokLabel PackNatural 0
  boxPackStart lokP (addrEntry gui) PackNatural 0
  boxPackStart (mainPanel gui) lokP PackNatural 0
```

```
  fstLabel    <- labelNew$Just "erster Kontakt (Gruppe,Nr)"
  boxPackStart (mainPanel gui) fstLabel PackNatural 0
  fstP        <- hBoxNew False 10
  boxPackStart fstP (fstCircuitGroup gui) PackNatural 0
  boxPackStart fstP (fstCircuitEntry gui) PackNatural 0
  boxPackStart (mainPanel gui) fstP PackNatural 0
  sndLabel    <- labelNew$Just "zweiter Kontakt (Gruppe,Nr)"
  boxPackStart (mainPanel gui) sndLabel PackNatural 0
  sndP        <- hBoxNew False 10
  boxPackStart sndP (sndCircuitGroup gui) PackNatural 0
  boxPackStart sndP (sndCircuitEntry gui) PackNatural 0
  boxPackStart (mainPanel gui) sndP PackNatural 0
  delayP      <- hBoxNew False 10
  delayLabel <- labelNew$Just "Wartezeit"
  boxPackStart delayP delayLabel PackNatural 0
  boxPackStart delayP (delayEntry gui) PackNatural 0
  boxPackStart (mainPanel gui) delayP PackNatural 0
  boxPackStart (mainPanel gui) (startStopControl gui) PackNatural 0
```

6.1.2 Events

And eventually we add event listeners to the controls. Internally we keep a state variable for the number of the listener that is active. It is initialized with the illegal number `-1`.

```
addCommutingGuiEvents gui eventListener state send = do
  let startStopB = startStopControl gui
  listenerNr  <- newMVar (-1::Integer)
  addrS <- entryGetText (addrEntry gui)
```

User Input

Whenever the start/stop button is clicked, we first have a look if the button is active.

```
    startStopB `onClicked` do
    mode  <- toggleButtonGetActive startStopB
```

If this is not the case then we stop the running commuting train.

```
    if not mode then do
        nr <- readMVar listenerNr
        stopCommunting nr (read addrS) eventListener send
     else do
```

Validation of User Input

Otherwise we can read and validate the user input. For validation of the user input we use the standard function `readMaybe`. Since the type `Maybe` is an instance of `Monad` we can use the do-notation for validation of the user input. If the user input cannot be evaluated we return with no action at all.

```
    fstCS <- entryGetText (fstCircuitEntry gui)
    sndCS <- entryGetText (sndCircuitEntry gui)
    fstGS <- entryGetText (fstCircuitGroup gui)
    sndGS <- entryGetText (sndCircuitGroup gui)
    delayS<- entryGetText (delayEntry gui)

    let inputValues =
            do
              a  <- (readMaybe addrS) ::Maybe Int
              fC <- (readMaybe fstCS) ::Maybe Int
              sC <- (readMaybe sndCS) ::Maybe Int
              fG <- (readMaybe fstGS) ::Maybe Int
              sG <- (readMaybe sndGS) ::Maybe Int
              d  <- (readMaybe delayS)::Maybe Int
              return (a,fG,fC,sG,sC,d)
    if (isNothing inputValues) then return () else do
```

Otherwise the user input can be used to start a Pendelzugautomatik.

```
    let (Just (addr,fstG,fstC,sndG,sndC,delay)) = inputValues
    nr <- startCommuting (fstG, fstC) (sndG, sndC) addr delay
                eventListener state send
    modifyMVar_ listenerNr (\_ -> return nr)
```

Chapter 7

Main GUI Client

Time to start everything in an main function. This is our main GUI entry point.

```
module Main where
import Z21.Protocoll hiding(light,direction, speed)
import Z21.State
import Z21.Gui
import Z21.MessageEventListener
import Z21.Commuting
import Z21.Constants

--gui library
import Graphics.UI.Gtk

-- Network library
import Network.Socket hiding (send, sendTo, recv, recvFrom)

--concurrency
import Control.Concurrent
import Control.Concurrent.Timer
import Control.Concurrent.Suspend.Lifted

--prog args and such
import System.Environment
```

Before we start the program we define a function that will send a message to the control every 10 seconds. This is needed by the control unit. It assures that our client is still alive.

```
keepAlive sendF = do
  sendF  LAN_GET_SERIAL_NUMBER
  repeatedTimer (sendF LAN_GET_SERIAL_NUMBER >>return ())$sDelay 10
```

```haskell
main :: IO ()
main = withSocketsDo $ do
  -- process arguments
  args <- getArgs
  let (host:portA:_) =
        if length args < 2
        then [_DEFAULT_CLIENT, show _DEFAULT_PORT]
        else args

  -- network stuff
  let port = fromInteger (read portA)
  sock <- socket AF_INET Datagram defaultProtocol
  bindAddr <- inet_addr "0.0.0.0"
  hostAddr <- inet_addr host

  bindSocket sock (SockAddrInet port bindAddr)
  let addr = (SockAddrInet port hostAddr)
  let sendF = sendMsg sock addr

  -- keep alive timer
  keepAliveThread <- keepAlive sendF

  -- synchronized state variable
  state <- newMVar newState

  -- make the gui
  initGUI

  gui <- newGuiControls

  eventListener <- newEventListener
  addListener eventListener (logging gui state)
  addListener eventListener $processMessage gui state sendF
  eventThread <- startEventListener eventListener sock

  commutingGui <- newCommutingGui
  mkLayoutCommutingGui commutingGui
  addCommutingGuiEvents commutingGui eventListener state sendF

  window2 <- windowNew
  set window2 [ containerBorderWidth := 10 ]
  set window2 [ containerChild := mainPanel commutingGui]

  window <- createGuiWindow gui state sendF

  window `onDestroy` do
    sendF LOGOFF
    killThread eventThread
    stopTimer keepAliveThread
    sClose sock
    mainQuit

```

```
  --show the windows
  widgetShowAll window
  widgetShowAll window2

  --start the gui thread
  mainGUI
  sClose sock
```

Chapter 8

Command language

In this chapter we provide a tiny library, which enables the user to program automatic sequences an cycles.

```
module Z21.CommandLanguage where

import Z21.Protocoll
import Z21.State
import Z21.MessageEventListener
import Z21.Commuting(Contact)

import Control.Concurrent.MVar
import Control.Concurrent.Timer
import Control.Concurrent.Suspend.Lifted

import Data.Bits
import Control.Applicative hiding ((<|>))
```

8.1 Scripting

First we give a data definition for commands. The data type is a generic (polymorphic) type. It has a type variable. This is not used in any way. The only reason for this is, to make it an instance of the type class `Monad`. This will enable the use of the do-notation.

```
data Command a =
```

A simple command consists of a direct Z21 message.

```
  Com Message
```

The next command is a timed command. After some seconds wait the command is to be executed.

```
|Wait Int (Command a)
```

Next a command is provided, which is triggered by some contact events. Only when every circuit track in the list of contacts has triggered a signal the command is excuted.

```
|OnEvent [Contact] (Command a)
```

A sequence of two commands will execute these command one after the other. It does not main, that we will wait for the first command to have finished completely.

```
|Sequenz (Command a) (Command a)
```

The last type of command allows to start to commands in parallel.

```
|Parallel (Command a) (Command a)
```

Furthermore we provide the possibility to call simple macros.

```
|Macro String
```

And it is possible to integrate an arbitrary IO action into an script. This will be used to ensure that connections can be closed and threads can be stoppped after the execution of a command.

```
|IOCommand (IO a)
```

For simple debugging reasons an instance of the class `Show` is provided. We could not derive the default implementation of this class, because `IO` is not an instance of `Show`[1].

[1] I wonder why function types and types such as `IO` do not have an derived default instance of `Show`. This would make things easier in many situations.

```
instance Show (Command a) where
  show (Com message) = "("++show message++")"
  show (Wait secs message)
    = "(Wait "++show secs++" "++ show message++")"
  show (OnEvent cts command)
    = "(OnEvent "++show cts++" "++ show command++")"
  show (Sequenz c1 c2) = "("++show c1++"\n <+> "++ show c2++")"
  show (Parallel c1 c2) = "("++show c1++"\n <|> "++ show c2++")"
  show (Macro name) = "(Macro "++name++")"
  show (IOCommand _) = "IOCommand"
```

The two constructors `Sequenz` and `Parallel` will not be used directly. Instead we define two operators to combine two commands. `<+>` is used for a sequential combination, `<|>` is used for a parallel combination.

```
infixr 5 <+>
infixr 4 <|>
```

Most cases for the sequential operator a straight forward.

```
(<+>) (Sequenz c1 c2) c3 = Sequenz c1 (c2<+>c3)
(<+>) (Wait sec c1) c2 = Wait sec (c1 <+> c2)
(<+>) (OnEvent contact c1) c2 = OnEvent contact (c1 <+> c2)
```

The interesting case is: where to put further commands after the parallel execution. We arbitrarily choose after the second command of two parallel commands. We will show examples on how this can be used to synchronize after several parralel executed commands.

```
(<+>) (Parallel c1 c2) c3 = Parallel c1 (c2<+>c3)
(<+>) c1 c2 = Sequenz c1 c2
```

The parallel operator is a direct call to the constructor `Parallel`.

```
(<|>) = Parallel
```

8.2 Execution of commands

The main function is called `run` and will execute a command. It gets the MessageListener, the global client state[2], the function to send messages to the z21 control unit, a map for the marcos and eventually the command to be processed.

```
run :: MVar MessageListener
    -> MVar State
    -> (Message -> IO a)
    -> [(String, Command a1)]
    -> Command a1
    -> IO ()
```

Simple cases are: sending directly a message to the control unit and executing some IO action.

```
run listener state send defs (Com c) = send c >> return ()
run listener state send defs (IOCommand c) = c >> return ()
```

Running a sequence of two commands can simply done by a sequence of recursive calls to `run` within a do-block.

```
run listener state send defs (Sequenz c1 c2) = do
  run listener state send defs c1
  run listener state send defs c2
```

To wait some seconds before executing a command is implemented with the help of the function `oneShotTimer`. This will actually start a new thread.

```
run listener state send defs (Wait sec c1) = do
  oneShotTimer
   (run listener state send defs c1)
   (sDelay$fromIntegral sec)
  return ()
```

The most complex command is the reaction to events on the circuit rails. This can be the point of synchronization. The command waits for a list of contacts. If these contacts are triggered by serveral commands in parallel, this is the moment where we synchronize on these command.

[2] This is not yet used.

The implementation resembles the implementation which we have done for commuting trains before. It is a generalization of the allready seen implementation.

First of all we determine the number of contacts we are waiting for. The we define a local state variable for counting the contacts which were triggered. Then we add a new event listener function.

```
run listener state send defs (OnEvent contacts c1) =
  let s = length contacts in do
  triggeredContacts <- newMVar ([])
  nr <- addOneTimeListener listener (evl triggeredContacts)
  return ()
```

The event listener function will react to `LAN_RMBUS_DATACHANGED` messages. First of all it determines the contacts that are signified in the message to have triggered[3]

```
  where
     evl triggeredContacts (LAN_RMBUS_DATACHANGED (gi:gs)) = do
        let adder = if gi==0 then 0 else 10
        let numbered=filter (\(nr,entry)-> entry/=0)$zip [1..] gs
        if (null numbered) then return False else do
        let changedGroup = adder+fst(head numbered)
        let groupVal = snd(head numbered)
        let addrs = filter (\(nr,code)->code==groupVal.&.code)
              $zip [(1::Int)..][0x01,0x02,0x04,0x08,0x10,0x20,0x40,0x80]
        if (null addrs)  then return False else do
        let newContacts = map (\(x,_)->(changedGroup,x))  addrs
```

Now we can check, if there are active contacts in the message, that we are waiting for and have not been triggered before.

```
        allreadyContacted <- readMVar triggeredContacts
        let actives = filter
                          (\c ->          c `elem` contacts
                                 && not (c `elem`allreadyContacted))
            newContacts
        if null actives then return False else do
```

If there are new active contacts in the message we will add them to the list of allready triggered contacts. Afterwards we can check, if all contacts of the command have triggered.

[3] To be done: Currently only messages for one single group are interpreted correctly.

```
      modifyMVar_ triggeredContacts (\x-> return (x++actives))
      allreadyContacted <- readMVar triggeredContacts
      if (length allreadyContacted==length contacts)
        then run listener state send defs c1 >> return True
        else return False
    evl _ _ = return False
```

The command for parallel execution is simply executed by starting to new threads. For some unknown reason we do this by the help of `oneShotTimer` instead of `forkIO`.

```
run listener state send defs (Parallel c1 c2) = do
  oneShotTimer (run listener state send defs c1) (sDelay 0)
  oneShotTimer (run listener state send defs c2) (sDelay 0)
  return ()
```

The call to a marco amounts in a lookup in the list of macros.

```
run listener state send defs (Macro name) =
  maybe
    (return ())
    (run listener state send defs)
    (lookup name defs)
```

8.3 Some useful commands (in German)

In this section we provide some useful functions that construct commands. The names of the functions are given in the german language. English translations are given further down.

We start with straighforward function for driving a locomotive, switching turnouts and german names for constructors.

```
fahre richtung lokAdresse geschwindigkeit =
  Com$LAN_X_SET_LOCO_DRIVE lokAdresse 2 geschwindigkeit richtung

halte lokAdresse = Com$LAN_X_SET_LOCO_DRIVE lokAdresse 2 0 Forward

schalteWeiche nr abzweig = do
  Com$LAN_X_SET_TURNOUT nr True abzweig
  warte 1$Com$LAN_X_SET_TURNOUT nr False abzweig

```

```
warte = Wait
wenn = OnEvent
```

Repeating commands can easily be implemented by means of a fold.

```
endlos = foldr1 (<+>) . repeat

wiederhole n = foldr1 (<+>) . take n . repeat
```

A function is providing for doing nothing.

```
macheNichts = Com$LAN_GET_SERIAL_NUMBER
```

The next two functions implement a Pendelautomatik.

```
hinUndHer locoid hin her geschwindigkeit = do
  fahre Forward locoid geschwindigkeit
  wenn [hin] (halte locoid)
  warte 5 (fahre Backward locoid geschwindigkeit)
  wenn [her] (halte locoid)
  warte 5 macheNichts

pendel locoid hin her geschwindigkeit =
  endlos  (hinUndHer locoid hin her geschwindigkeit)
```

Further German names for constructors.

```
vorwärts = Forward
rückwärts= Backward
```

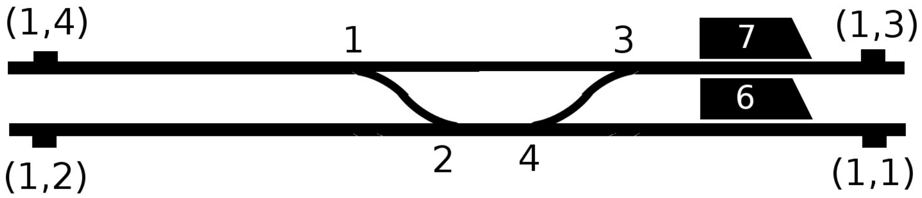
8.3.1 English Translation

Her the English versions of the functions above.

```
go             = fahre
commute        = pendel
forthAndBack   = hinUndHer
doNothing      = macheNichts
switchTurnout  = schalteWeiche
repeatNTimes   = wiederhole
wait           = warte
when           = wenn
forever        = endlos
stop           = halte
```

8.3.2 Example Script

We give a simple example script. It is written for the following track layout. There are 4 turnouts and 4 circuit tracks. Two locomotives. We want the locos cross over to the other track. The is done sequentually. Then both to go to the other end of the track. Now they have changed places from the original scenario. This is repeated so that eventually the starting point is reached for both locomotives.




```
crossing =
        schalteWeg 1 4
  <+> warte 5 (fahre rückwärts 6 5)
  <+> wenn  [(1,4)] (halte 6)
  <+> schalteWeg 3 2
  <+> warte 5 (fahre rückwärts 7 5)
  <+> wenn [(1,2)] (halte 7)
  <+> schalteWeg 4 3
  <+> warte 5
    (      fahre vorwärts 6 5 <+> wenn [(1,3)] (halte 6)
     <|>   fahre vorwärts 7 5 <+> wenn [(1,1)] (halte 7)
     <|>  wenn  [(1,3),(1,1)]
        (schalteWeg 1 4
        <+> warte 5 (fahre rückwärts 7 5)
        <+> wenn [(1,4)] (halte 7)
        <+> schalteWeg 3 2
        <+> warte 5 (fahre rückwärts 6 5)
        <+> wenn [(1,2)] (halte 6)
        <+> schalteWeg 4 3
        <+> warte 5
```

```
          (        fahre vorwärts 6 5 <+> wenn [(1,1)] (halte 6)
            <|>    fahre vorwärts 7 5 <+> wenn [(1,3)] (halte 7)
            <|>    wenn [(1,3),(1,1)] macheNichts
          )
        )
    )

schalteWeg 1 4 = do
  schalteWeiche 2 False
  schalteWeiche 4 True
  schalteWeiche 1 False
schalteWeg 4 1 = schalteWeg 1 4
schalteWeg 3 2 = do
  schalteWeiche 3 False
  schalteWeiche 4 False
  schalteWeiche 2 True
schalteWeg 2 3 = schalteWeg 3 2
schalteWeg _ _ = do
  schalteWeiche 1 True
  schalteWeiche 2 True
  schalteWeiche 3 True
  schalteWeiche 4 True
```

8.4 Making `Command` an instance of `Monad`

One of the nice thing of monads is the do-notation. The command language we presented seems to be a perfect candidate for an instance of `Monad`. Then we can use the do-notation as allready done in the example script in the section above.

However, the command language does not easily fit into a command. A type that is an instance of `Monad` needs to have a type variable. Our command language does not need a variable inner type. In the definition of `Command` we introduced a dummy type variable. To make it an instance of `Monad` we first need it to be an instance of `Functor`. Actually we do not need this instance but it will be used pathologically.

```
instance Functor Command where
  fmap f (Com com) = Com com
  fmap f (Wait sec com) = Wait sec (fmap f com)
  fmap f (OnEvent contact com) = OnEvent contact (fmap f com)
  fmap f (Sequenz c1 c2) = (Sequenz (fmap f c1) (fmap f c2))
  fmap f (Parallel c1 c2) = (Parallel (fmap f c1) (fmap f c2))
```

Since very recent times we need to be an instance of `Applicative`. We make a dummy implementation, which will never be used.

```
bot = bot

instance Applicative Command where
  pure a = macheNichts
  f <*> a = fmap bot a
```

Eventually we implement the instance of `Monad`. As a matter of fact we are onbly interested in the sequence operator `>>`.

```
instance Monad Command where
  return a = macheNichts
  c1 >> c2 = fmap (\_->bot) c1 <+> c2
  c1 >>= fc2 = fmap (\_->bot) c1 <+> fc2 bot
```

8.5 Command language parser

In this section we present a parser for the command language. Writing parsers has been a killer application for lazy evaluated functional languages. The idea is to write higher order combinator functions which combine two parsers sequentially and alternatively. The sequence means: first parse the input with one parser and then parse the remaining tokens with a second parser. The combination results in a new parser. The alternative means: try to parse the input wiht one of the two parsers. A fine example of a cominator parser is given in [FL89]. This work has even be done in pre Haskell times. With the arrival of monads and the do-notation [Lau93] [Lau93] it became clear that the sequence operator can be written as the bind operation in monads. Thus parser combinators where implemented as instance of `Monad`. Therefore parser combinator libraries are now called monadic parser libraries. A mature and efficient Haskell parser library is `parsec` [Lei01]. Within this section a parser is implemented with the parsec library.

```
module Main where
import Z21.CommandLanguage hiding ((<|>))
import qualified Z21.CommandLanguage((<|>))

import Z21.Protocoll
import Z21.Constants

import Text.ParserCombinators.Parsec
import Text.Parsec.Pos
import Text.Parsec.Prim

```

```
import Language.Haskell.Lexer

import Z21.State
import Z21.MessageEventListener

-- Network library
import Network.Socket hiding (send, sendTo, recv, recvFrom)

--concurrency
import Control.Monad (forever)
import Control.Concurrent
import Control.Concurrent.Timer
import Control.Concurrent.Suspend.Lifted

--prog args and such
import System.Environment
```

8.5.1 Lexer and Parsing of Token

Before we write the actual parser we need a lexer. Since our command language is defined in Haskell, we can use a Haskell lexer. Fortunately a complete Haskell lexer is available in the module `Language.Haskell.Lexer`. We will apply this lexer by the call of `lexerPass0`, afterwards remove white space token and eventually add the position information as needed by the parsec library.

```
lexAll fileName
  = (map (\(tok,(pos,s))->
     (newPos fileName (fst$simpPos pos) (snd$simpPos pos),(tok,s))))
    . rmSpace . lexerPass0
```

The Haskell lexer emits token consisting of a pair. The first component is an enumeration type signifiying the token type. The second component is the actual token string.

```
type HaskellTok = (Token, String)
```

We define two atomic parser. One for accepting integer token and for accepting arbitrary identifier token.

```
integerNumber ::  (GenParser (SourcePos,HaskellTok) () Int)
integerNumber = token (show.snd) fst (testIdent.snd)
```

```
  where
    testIdent v@(IntLit,n) = Just$read n
    testIdent _ = Nothing

identifier = token (show.snd) fst (testIdent.snd)
  where
    testIdent v@(Varid,n) = Just n
    testIdent _ = Nothing
```

We provide a general function for generating arbitrary atomic parsers.

```
parseTok
  :: Token -> String -> a -> (GenParser (SourcePos,HaskellTok) () a)
parseTok tok name res = token showToken posToken testToken
  where
    showToken (pos, tok) = show tok
    posToken  (pos,_)    = pos

    testToken (pos, v@(t,n))
      |t == tok && n == name  = Just res
      |otherwise              = Nothing
```

Thus we can define some specific parsers.

```
ident name res = parseTok Varid name res

constructor con res = parseTok Conid con res

special name  =  parseTok Special name ()

reservedop name  = parseTok Reservedop name ()

pVarsym name = parseTok Varsym name ()
```

8.5.2 Grammar

Now we can write down a grammar for the command language. Sequences can be expressed with the do notation. Alternatives are written down with the combinator `Text.Parsec.Prim.<|>`. However the combinator `<|>` for effeciency reasons does not perform backtracking. If the first parser can be successfully applied, the second parser will not be considered any more.

We write the grmmar top down and start with the start rule: `pSkript`. A script will consist of a number of macro definitions followed by one single command.

A skript for the Z21 client consists of several macro definitions. These are followed by a single command, which is to be executed. Since both, a macro definition and a command can start with an indentifier, we need to apply backtrakcing. This can be done with the `parsec` function `try`.

```
pSkript =
  do
   defs <- many (Text.ParserCombinators.Parsec.try pDefinition)
   cmd  <- pCommand
   return (defs,cmd)
```

A macro definition consists of some arbitrary identifier followed by the reserved operator symbol `=` and finally the command for the macro.

```
pDefinition = do
  n <- identifier
  reservedop "="
  cmd <- pCommand
  return (n,cmd)
```

The entry rule for commands is the parallel operation on commands.

```
pCommand = pParallel

pParallel = do
  ps <- sepBy1 pSequence (pVarsym "<|>")
  return$ foldr1 (Z21.CommandLanguage.<|>) ps
```

Thus the sequential operator binds stronger than the parallel operator.

```
pSequence = do
  ps <- sepBy1 pAtomicCommand (pVarsym "<+>")
  return$ foldr1 (Z21.CommandLanguage.<+>) ps
```

Atomic commands are the commands that were defined in the section before. These are basic commands for driving a train (`fahre`, `halte`), the command for switching a turn out (`schalteWeiche`), commands for waiting and synchronizing at events, the call of an macro definition and command in parantheses.

```
pAtomicCommand =
        pHalte
   <|> pFahreLokSpeed
   <|> pSchalteWeiche
   <|> ident "macheNichts" macheNichts
   <|> pWarte
   <|> pWenn
   <|> (identifier >>= (return.Macro))
   <|> pParCommand
```

All these parsers can easily be written done by use of the do notation.

```
pFahreLokSpeed = do
   f <- ident "fahre" fahre
   dir <- pDirection
   loco <- integerNumber
   speed <- integerNumber
   return$ f dir loco (fromIntegral speed)

pHalte = do
   ident "halte" ()
   loco <- integerNumber
   return$ halte loco

pWeichenstellung =
   (ident "geradeaus" True)
       Text.Parsec.Prim.<|>
   (ident "abzweigung" False)
       Text.Parsec.Prim.<|>
   pBool

pBool = (constructor "True" True)
            Text.Parsec.Prim.<|>
          (constructor "False" False)

pSchalteWeiche = do
   ident "schalteWeiche" ()
   nr <- integerNumber
   abzweig <- pWeichenstellung
   return $ schalteWeiche (fromIntegral nr) abzweig

pDirection = pBackward <|>  pForward

pBackward = constructor "Backward" Backward
pForward = constructor "Forward" Forward

pWarte = do
   ident "warte" ()
   sekunden <- integerNumber
   cmd <- pAtomicCommand
```

```
  return$ warte sekunden cmd

pPair = do
  special "("
  x <- integerNumber
  special ","
  y <- integerNumber
  special ")"
  return (x,y)

pWenn = do
  ident "wenn" ()
  special "["
  contacts <- sepBy1 pPair (special ",")
  special "]"
  cmd <- pAtomicCommand
  return $wenn contacts cmd

pParCommand = do
  special "("
  com <- pCommand
  special ")"
  return com
```

8.5.3 Executing Scripts

This module has a main function. A script file can be submitted as command line argument. The file we be parsed and then be executed.

We define a simple function for logging and a keep alive thread.

```
logger msg = print msg >> return False

keepAlive sendF = do
  Control.Monad.forever
    $oneShotTimer (sendF LAN_GET_SERIAL_NUMBER>>return ())
    $sDelay 10
```

The main function processes the arguments and amounts in a call of the function `moin`. This way it is possible to statr the function `moin` in the interpreter `ghci`[4].

```
main = do
  args <- getArgs
  if (length args < 1)
```

[4] ›Moin‹ in northern parts of Germany ist used for ›Hello‹. I use it as almost ›main‹

```
 then (putStrLn "usage: runZ21 skriptfile [host port]") else do

let (host:portA:_) =
      if length args < 3
      then [_DEFAULT_CLIENT, show _DEFAULT_PORT]
      else (tail args)

let skriptFile = head args
moin skriptFile host portA
```

In order to run a command, we need to install the network connection, create a global state and the event listener queue.

```
moin skriptFile host portA = do
  -- network stuff
  let port = fromInteger (read portA)
  sock <- socket AF_INET Datagram defaultProtocol
  bindAddr <- inet_addr "0.0.0.0"
  hostAddr <- inet_addr host

  bindSocket sock (SockAddrInet port bindAddr)
  let addr = (SockAddrInet port hostAddr)
  let sendF = sendMsg sock addr

  -- synchronized state variable
  state <- newMVar newState

  eventListener <- newEventListener
  addListener eventListener logger
  eventThread <- startEventListener eventListener sock
```

A state variable is used to signify if execution of the command has finished. This is used to wait for the end of the execution. Afterwards we need to close the socket and end the program.

```
  done <- newEmptyMVar
```

We will parse the script. Then add the setting of the variable `done` at the end of the script. Then hte execution is started.

```
  sendF LAN_SET_BROADCASTFLAGS
          {general=True,rbus=True,systemState=True}
  file <- readFile skriptFile
  let presult = parse pSkript skriptFile (lexAll skriptFile file)
```

```
  either
    (print)
    (\x->(print x >>
      (run eventListener state sendF (fst x)
       ( snd x
        <+> IOCommand(sClose sock >> putMVar done () )))))
    presult
```

We wait for the variable `done` to be set.

```
  takeMVar done -- blocks till MVar is full
  print "All done"
```

8.5.4 Example Script

Here follows the script which petforms the same procedure as seen in a previous section.

```
schalteWeg_1_4 =  schalteWeiche 2 abzweigung
  <+> schalteWeiche 4 geradeaus
  <+> schalteWeiche 1 abzweigung

schalteWeg_3_2 = schalteWeiche 3 abzweigung
  <+> schalteWeiche 4 abzweigung
  <+> schalteWeiche 2 geradeaus

schalteWeg_1_2_3_4 = schalteWeiche 1 geradeaus
  <+> schalteWeiche 2 geradeaus
  <+> schalteWeiche 3 geradeaus
  <+> schalteWeiche 4 geradeaus

      schalteWeg_1_4
  <+> warte 5 (fahre Backward 6 5)
  <+> wenn [(1,4)] (halte 6)
  <+> schalteWeg_3_2
  <+> warte 5 (fahre Backward 7 5)
  <+> wenn [(1,2)] (halte 7)
  <+> schalteWeg_1_2_3_4
  <+> warte 5
   (     fahre Forward 6 5 <+> wenn [(1,3)] (halte 6)
    <|>  fahre Forward 7 5 <+> wenn [(1,1)] (halte 7)
    <|>  wenn  [(1,3),(1,1)]
      (    schalteWeg_1_4
      <+> warte 5 (fahre Backward 7 5)
      <+> wenn [(1,4)] (halte 7)
```

```
        <+> schalteWeg_3_2
        <+> warte 5 (fahre Backward 6 5)
        <+> wenn [(1,2)] (halte 6)
        <+> schalteWeg_1_2_3_4
        <+> warte 5
          (       fahre Forward 6 5 <+> wenn [(1,1)] (halte 6)
            <|> fahre Forward 7 5 <+> wenn [(1,3)] (halte 7)
            <|> wenn [(1,3),(1,1)] (warte 5 macheNichts)
          )
       )
    )
```

Listing 8.1: crossing.z21

Chapter 9

Conclusion

So what are the lessons learnt? This is a rather personal conclusion.

First of all: success. Well there where no major problems putting the task into realization with the use of Haskell. Although having not used Haskell for quite a long time and not being familiar with most of the libraries used, everthing went quite smoothly. However, I have known Haskell for over 20 years by now. Introducing bachelor students in the fourth term to functional programming often results into a complete desaster. They have been trained in Java and C. They are quite firm in using these languages. But functional programming leads them to a completely alternative view of the world. They are confused by the syntax, by the concepts and the lack of classes and anything they have seen so far. In courses of functional programming I take them away everything they are used to as the backbone of programming: classes, loops, assignments. Only a few of my students get fascinated by this alternative world and start doing amazing things within it.

I must admit: I fell in love with Hoogle [Mit08]. It is a nice online help to get familiar with unknown libraries.

I still love the concept of literate programming. All modules where written in literate style. Viewing the source code of this project rather as a report than as a collection of source files gives a complete different feeling to programming. Feels rather like beeing an author than a programmer. This is especially due to the very short and concise nature of the Haskell syntax. Almost no boiler plate code is necessary as e.g. in Java. The literate style leads to a much closer preoccupation with te source code. Many bugs where found simply by writing down the explaining parts of the code. I personally read much more over the source code than I would have done in a corresponding Java project.

The problem beginners might have with Haskell is: no structure or architecture is preset by the language. Classes in object oriented languages give a firm structure for a programmer: structure your domain in classes, write constructor functions for object creation and so forth. Haskell simply says: write some functions and make some algebraic data definitions. Even type classes are not as generally used as interfaces in Java. This makes the beginner feel a bit lost. However, I was fascinated how easily standard solutions for concepts like event listener, or something like a model view control pattern could be implemented in Haskell. The use of the rather simple GTK library did not pose any serious problem.

My personal view of the type system. Well, I love it. Writing down programs in Haskell is like writing down things in a script language but with the comfort of being statically typed. It does not have the burden to writing down type signatures for every little function, but the type inference algorithmn does detect errors.

I love currying and lambda lifting. Everything seems so light weight. This is one of the things where the fun in functional programming comes from. Create a local function by leaving some arguments. Or define a local function which magically can use things defined in an outer context. Not much to worry about and nevertheless things work as expected. This is one of the reasons I like the concept of effectively final in Java 8.

What about monads? Well, I love and hate them. Even after two decades I find them still confusing. I know how to define an instance of `Monad`. I can work with monads. I somehow love the do notation although I am aware of the voices which consider the do notation harmful [doh][1]. However I find that I am not very firm in using monads. Functions like `liftM` do not come easily into my mind. I still feel lost when serveral different instances of `Monad` are used. Yes, most of all, there is the IO monad, but there are exceptions, error monad, state monad. Lists are monads and very useful the monadic instance of `Maybe`. Combining all these makes me feel a bit lost and stupid. A feeling Haskell programmers often get: I am not smart enough for these things. This language is build for the brilliant gals and guys. Easily followed by the euphorical moments: yes, I am one the few brilliant guys who understand this wonderful precise and clear language. Feelings you do not get when writing Java code. There are no such highlights in Java code. Writing down Java code is solid handcraft wihtout any fascination.

Haskell on the other hand has fascinating moments: this e.g. I think whenever I write a parser or define a parser combinator library on my own. I still find the most fascinating paper I have ever read, the early parser combinator paper by Frost and Launchberry from 1988 [FL89].

As has been stated several times and very convincingly by Simon Peyton-Jones: the future is parallel, and the future of parallel is declarative [PJ11] We needed concurrency within this project. As a matter of fact: the light weight threads and `MVar` implementation for concurrency in Haskell were easily applied. I am not sure if this is the state of the art way to solve these things in Haskell, but it was so beautifully easy to use. And it works. Great.

Implementing a complex modifiable state as done in module `State` of this project feels clumsy. Maybe there are better ways to this, which I do not know about. The way I started it seemed to be full of avoidable boiler plate code. Especially when compared to a simple assigment in Java.

[1] But is there any concept without an harmful article about it?

And the major drawback for using Haskell? Haskell is used to compile stand alone desktop or server applications. Especially in the presented project it would have been nice to have written code that could be used within an application for mobile phones or for browser applications. This currently is a strong point of Java. Android applications are written as Java source code and browser-server web-application can be written completely in Java by the help of Google Web Toolkit.

As for the other goal of this project. Yes, I did have ›fun with trains‹. Greetings to Sheldon Cooper.

Appendix A

Constants

The Z21 control has a fixed IP number and a fixed port. It even demands the clients to use the same port.

```
module Z21.Constants  where

_DEFAULT_CLIENT="192.168.0.111"
_DEFAULT_PORT=21105
```

Appendix B

Utility Functions

This module has been created to take all utility functions used within the project. Eventually just one such functions has been written. Most other functions were allready present in some standard library and needn't to be invented again.

```
module Util where

splitNChunks n = takeWhile(not.null) . map(take n) . iterate(drop n)
```

Appendix C

A Simple Test Server

This tiny module had been create for testing sending an receiving messages.

```haskell
module Main where
import Z21.Protocoll

import Network.Socket hiding (send, sendTo, recv, recvFrom)
import Network.Socket.ByteString

import Numeric (showHex)
import qualified Data.ByteString as C

import Control.Monad (forever, when)
import Data.IORef

port = 21105
host = "0.0.0.0"

type Host = SockAddr

main = withSocketsDo $ do
        s <- socket AF_INET Datagram defaultProtocol
        bindAddr <- inet_addr host
        bindSocket s (SockAddrInet port bindAddr)

        hostsRef <- newIORef []
        forever $ do
                (msg, hostAddr) <- recvFrom s 1024
                putStrLn $ (show hostAddr)
                putStrLn$show$map (\x -> showHex x "")$ C.unpack msg
                print$readMessage msg
                hosts <- readIORef hostsRef
                when (notElem hostAddr hosts) $ modifyIORef hostsRef (hostAddr:)
                hosts <- readIORef hostsRef
                sendToAll s ( msg) $ hosts
        sClose s

-- sendToAll :: Socket -> String -> [Host] -> IO ()
sendToAll socket msg hosts = do
```

```
            mapM_ (sendTo socket msg) $ hosts
```

Bibliography

[Com97] JMRI Community. JMRI a Java-based cross-platform application for model railroaders. `http://jmri.sourceforge.net/help/en/html/apps/index.shtml/`, 1997.

[doh] Do notation considered harmful. `https://wiki.haskell.org/Do_notation_considered_harmful`.

[FL89] R. Frost and J. Launchbury. Constructing natural language interpreters in a lazy functional language. *The Computer Journal*, 32(2):108–121, 1989.

[Gmb13] Modelleisenbahn GmbH. Z21 LAN Protokoll. `http://www.z21.eu/content/download/1589/16083/file/Z21_LAN_Protokoll%20V1.05.pdf`, March 2013.

[Lau93] John Launchbury. Lazy imperative programming. In *Yale University*, pages 46–56, 1993.

[Lei01] Daan Leijen. Parsec, a fast combinator parser. Technical Report 35, Department of Computer Science, University of Utrecht (RUU), October 2001.

[Mit08] Neil Mitchell. Hoogle overview. *The Monad.Reader*, (12):27–35, November 2008.

[Pan03] Sven Eric Panitz. HOpenGL – 3D Graphics with Haskell, A small Tutorial. Online Script, TFH Berlin, 2003. `www.panitz.name/hopengl/index.html`.

[PJ11] Simon Peyton-Jones. The future is parallel, and the future of parallel is declarative. https://yow.eventer.com/yow-2011-1004/the-future-is--parallel-and-the-future-of-parallel-is-declarative-by-simon-peyton-jones-1055, 2011.

[vT08] Hans van Thiel. Gtk2Hs Tutorial. `http://code.haskell.org/gtk2hs/docs/tutorial/Tutorial_Port/`, 2008.

[Wag15] Daniel Wagner. Threading and Gtk2Hs. `http://dmwit.com/gtk2hs/`, 2015.