Advanced programming - Exam paper

Examination number: 153

November 11, 2022

Assessment

For the Haskell parser I chose to use the Parsec library to parse the incoming text.

Completeness

Appy: Parser

The parser is working up to specification, except a few rules aren't followed. I have not done the comment part of the parser, I didn't do this as I saw other places in the assignment that I would rather spend my time to try and make them work. If I were to implement it I would check for "–" in the spaceTab function that runs after every time, a space could be made, and would just eat until a newline was made.
The same goes with the htext ability to do {{ and }}, but could be done using a new function, where it checks if the next String is "{{" or "}}", and then return a single if it is and otherwise each the printable character.

Appy: Transformer

In the transformer, there are many things that never got implemented into it, as I at first didn't know how I was supposed to do it, and therefore was not able. The *converter* does not make any checks on the grammar such as checking all the non-terminals are actually defined.
Besides this, the predicate is not defined in the converter, and will therefore not be converted to BNF, from its EBNF, and will even lead them to an error, and not a text error. I didn't implement left-factorization and did not try to do so, and it therefore just returns the given grammar.
For left recursion elimination, I have an implementation that I feel is close but not quite there and produces the completely wrong answer at this point. I have left it in the code so it's possible to see the idea that I wanted to do with it, but didn't have time to finish. Since I haven't implemented any monad for the I will instead just describe which monad I would have done if more time was given. I would have used a state monad as we are editing the state with new rules, and editing existing rules with changes to how they are built up. This is exactly what state monads are for as we are just trying to change the entire state.
When looking through the actions that are throughout the task I have simply not used them for anything other than Type that references it and uses it for the *AUser htext* to define that one. For the rest, it didn't make sense how to use it and, therefore I was not able to implement the actions and have therefore been left out of the implementation with some small irrelevant times where it is used.

Stock exchange: Question 1

I have implemented all of the given API for the user such that the user gets the correct result back as the user would expect.

There are still some internal functions where the different processes should call each other behind the scenes that are not working as expected. Here for example when a stock has been bought it should terminate the processes where the traders are calculating their answers, although right now, they are not doing this. There is also a Status field saved in the State of each of the considered transactions. Where the process of calculating should tell whenever it is done, doing the calculation. This then leads to us shutting down processes on a shutdown that have already been shut down, as this doesn't lead to an error. If that process ID is then used again, we would be shutting down another process which could lead to problems.

Stock exchange: Question 2

I have not solved any of the tasks in this question or any of the given tests in it, I have therefore reused the *test_erlst.erl* to encompass my unit test, where I go over all of the functionality with a unit test in mind. Since it haven't been implemented I will not speak further about this question in the report.

Correctness

Appy: Parser

I found some left recursion in the given grammar and fixed these, specifically, I found it in the simple0, and made a new such that we get out of the infinite looping. I also found some instances where left factorization was possible, in the Simple, Alts, and ERules.
For my implementation, I had chosen that the structure of what u build doesn't need to be on a specific amount of lines for each thing, and can write it with any number of spaces, new lines, or tabs.

Appy: Transformer

Because of the way the parser works, will nested bars sometimes be valid, as the datatype for the bar is:

Bar ERHS ERHS

If multiple bars are in a single *ERule* they must be nested with each other. The parser will use the second *ERHS* to contain the next bar, so when converting it from *EBNF* to *BNF* if a nested bar is located there after a non-nested bar, it will not detect it as a nested bar.

To make sure that we can indeed solve the passing with the extra rules that the system can create. I used a data structure looking like this:

([Rule], [([Simple], Action)])

This allowed me to create the right-hand side of the rule, in the converter and pass back any new rules that might come from changing from *EBNF* to *BNF*, in such a way

that we could easily add and use the right-hand side with a few *case ... of* to find the data we are looking for. This would also be what the state monad would interact with to keep a state of the current scheme, to make it easier to add to the states as this is something I do a lot and makes the code complicated and hard to look at.
I feel like I had a quite clever way of doing the *ESeq* such that it creates an array of *[Simple]* using the foldr, by taking the first element of a list as the first argument and the second argument is the rest of the list, and then adds them together. Inside the lambda function using the action of the first.

Stock exchange

Beneath in Figure 1, a simple diagram of the connections between the different processes can be seen. The server consists of 4 different processes that can be spawned. The *Stock exchange*, *Account*, *Trader* and *calculate decision.* The stock exchange is keeping track of the offers and account holdings and makes the heavy lifting whenever there comes a request requiring that, as it has the data necessary. The account is mostly a reference to the account where the traders are held, and used as a relay, for messages between the user and the *stock exchange*, and messages between the *traders* and the stock exchange. The *traders* are mostly to spawn new processes that can use their strategy, to decide if they wanna buy the stock. And the *calculate decision* is only to try and calculate a decision from an offer, and send the decision to the account. Going over all of the functionality would be a long walk-through and is not all interesting. Instead of going over all the functions a select few interesting functions such as making an offer, and executing trades, will be discussed.

Make an offer: When this exported function is called a signal is sent to the account specified. The account will send a signal to the stock exchange that holds all offers, and account data, and the stock exchange will decide if it is possible to make this offer. If it is possible, the stock is removed from the account and a message with the OfferId is sent back to the user, behind the scene a signal is sent to all accounts, and from there sent to all traders with the offer is sent. If it isn't possible a message with the error is sent back to the user.

Executing a trade: When each trader receives the offer, they start a new process and calculate using their strategy. The result is sent to the account that will forward it to the stock exchange. It raises the question of who will get it if 2 traders both accept the offer. In my implementation, this is a race condition, the first to send it to the exchange that has the account funds to do so is allowed. The offers status will then be changed and the funds changing hands, as well as the stock, will be added to the buyers' wallet. It all happens behind the scene, without anyone being noticed about it.

Another mention: Adding traders is not completely following the specification. When adding a trader, it sends a signal to the account that will open a new process to run the trader, and will be added to the accounts list of traders and sent the PID back to the user. The trader then works as described above. This way of creating the Trader leaves out some important ways of trading, especially since the trader will not trade on any of the earlier trades.

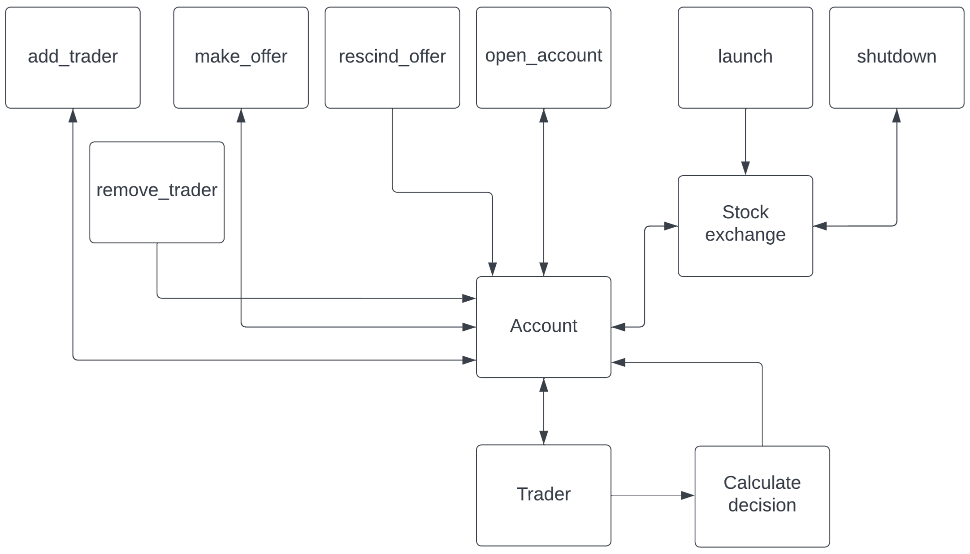

Figure 1: A simple process diagram and their interactions.

In my implementation, I chose to keep the holding data of each account on the *stock exchange*. I kept it in the stock exchange since it was where I would like to verify everything, so keeping a copy of everything, where it is used and changed, seems to be easier than waiting, for responses of the other processes keeping back both the stock exchange as it waits for the data as well as putting more load on those processes.
Since we are using race conditions to determine who will be able to buy the stock, I saw no use for the reject options to be sent to the stock exchange as it is not necessary. The function is now build in still, so it can easily be implemented if needed for anything such as logging the option will be there. I have some things that I would like to change if I were to make this exercise again. First using the guards of a *try ... catch ...* should have been used when updating an account's holdings if it were to fail such that an accurate error message would be sent back. This would also prevent some of the main servers from shutting down in volatile moments, to make sure that nothing would happen. This could also be done in many other places in the implementation.
Another consideration I chose not to do when I sat down with the task was to not use OTP as it seemed like a small task and wouldn't be needing the setup for it. After doing it with the many transactions there are between the servers it would have been fantastic to have an OTP setup to handle the servers.
Another thing I would change would be to not make a trader their own process but just a data structure in the *account* data structure, as it doesn't do that many things, mostly just spawning new processes or holding a bit of data, which could easily have been done in the account as a data structure.

Efficiency

Appy: Parser

The parser's efficiency is quite good as it, and I have no reason to suspect that there is any bad usage of memory, or performance in the parser.

Appy: Transformer

I do not think the transformer has bad memory handling nor do I think that the transformer isn't efficient, a monad would have made it easier to look at, and could probably also have made the code more efficient as it would not need as many cases.

Stock exchange

It is said in the stock exchange that the Offers should disappear whenever they are settled, I chose not to do this, and instead keep all of them for logging purposes, of their state, and so on. But since it is not removed, it could lead to a big use of memory, as the list grows when the stock exchange has been open for longer periods of time.

Robustness

Appy: Parser

The parser seems to be quite robust with all of the implemented and seems to pass, all of the things I have thrown at it correctly and, how I have envisioned it.

Appy: Transformer

The converter is not very robust not really checking up on anything and can't do a failing message, if anything should happen, it would just fail with an *error* and is quite messy. Some errors also exist in the converter which makes it not very reliable to be using and it will give the wrong answer in these cases damaging the robustness of the transformer.

Stock exchange

The stock exchange is not very reliable if an exception were to be thrown then the entire process can be laid down as there do not exist any fail safes as is discussed under the completeness, part of this report.

Maintainability

Appy: Parser

The parser is quite neatly implemented, into auxiliary functions that, that is easy to follow, and would be easy to extend the grammar using the functions that already exist.

Appy: Transformer

The maintainability of the transformer is quite a mess. Since I haven't used any monads to create the transformer. This results in the transformers being hard to understand, and being large without doing that much work, but splitting up the data, and creating different new data structures from that.

Test for Appy

There exists a test for both the given parts of the Appy parser and a bit for the Appy transformers, for the implemented functionality.
To run these tests, stand in the *.../appy* here you can easily run all the tests using the command *stack test* to run all the appy tests that are made.

Stock exchange

The maintainability of the Erlang code is not in a good state, a lot of the functionality inside the servers, especially the *stock exchange*, could make use of some functions to handle it. Many of the functions that already are made are very specific and could be split into small functions more general, and then some specific functions using those.

Test in erlang

To run the given test you should stand in the *.../erlst/src* and use "*erl*" command and copy "*c(erlst), c(test_erlst), test_erlst:test_all().*" into the first line. These will run some tests that will go over the simple and some a little more advanced tests of the different, functionalities of the server. Although a big part of erlang is that it should be easy to hold many instances of processes my test only makes use of a small amount of each instance which in a wider test should be looked into how it behaves with more processes running on the server.

Appendix

ParserImpl.hs

```haskell
-- Put yor parser implementation in this file
module ParserImpl where

import Definitions
import Data.Char
import Text.ParserCombinators.Parsec  -- exports a suitable type ParseError
import Text.Parsec.Char

type Preamble = String
type ParseError = String

-- used from an earlier assignment done with Mikkel
spaceTab :: GenParser Char st ()
spaceTab = try (do      tab
                        spaceTab)
        <|> try (do     endOfLine
                        spaceTab)
        <|> try (do     space
                        spaceTab)
        <|> do  spaces
                return ()

-- the exported funtion to create the start of the parser
parseSpec :: String -> EM (Preamble, EGrammar)
parseSpec s = do        case parse (spec "") "Error" s of
                            Left x -> Left (show x)
                            Right x -> return x

spec :: String -> GenParser Char st (Preamble, EGrammar)
spec s = do string "---" -- waiting for a string starting with the ---
            char '\n' -- expecting a newline right after
            spaceTab
            e <- eRules
            spaceTab
            eof
            return (s, e)
    <|> do  c <- anyChar -- reading the preamble
            s <- spec ((++) s [c])
            return s

-- Parsing eRules
eRules :: GenParser Char st EGrammar
eRules = do e <- eRule
            eRulesL [e]

-- left factoring of the erules
eRulesL :: EGrammar -> GenParser Char st EGrammar
eRulesL e = do  e2 <- eRules
                return ((++) e e2)
        <|> do  return e

-- erule parsing
eRule :: GenParser Char st ERule
eRule = do  lhs <- lHSStart
            return lhs

-- find lhs left hand side with either type or not
lHSStart :: GenParser Char st ERule
lHSStart = try (do  (n, kind) <- lhsName
                    spaceTab
                    opt_type <- optType
                    spaceTab
                    lHSEnd n kind (Just opt_type))
        <|> do  (n, kind) <- lhsName
                spaceTab
                lHSEnd n kind (Nothing)

-- Finding the right hand side ERule
lHSEnd :: String -> RKind -> Maybe Type -> GenParser Char st ERule
lHSEnd n k t = do   string "::="
                    spaceTab
                    a <- alts
                    char '.'
                    spaceTab
                    return ((n, k, t), a)

-- finding the type of the rule
optType :: GenParser Char st Type
optType = do    string "{:"
                spaceTab
                s <- many1 (satisfy (\c -> c /= '}'))
                spaceTab
                char '}'
                spaceTab
                return (AUser s)

-- for finding the alts in the rule
alts :: GenParser Char st ERHS
alts = do   erhs <- seqe
            spaceTab
            altsL erhs

-- left factoring of the alts rule
altsL :: ERHS -> GenParser Char st ERHS
altsL erhs = do     char '|'
                    erhs2 <- seqe
                    altsL (EBar erhs erhs2)
        <|> do  return erhs

-- finding the different sequences of the
seqe :: GenParser Char st ERHS
seqe = try (do  simz <- simplez
                char '{'
                h <- htext
                char '}'
                return (ESeq simz h))
        <|> do  sim <- simple
                return sim

-- simplez parser
simplez :: GenParser Char st [ERHS]
simplez = try (do   sim <- simple
                    spaceTab
                    simz <- simplez
                    spaceTab
                    return ((++) [sim] simz))
        <|> do  return []

-- Simple with a left factorized version
simple :: GenParser Char st ERHS
simple = do char '!'
            sim0 <- simple0
            return (ENot sim0)
    <|> do  s <- simple0
            simpleL s

-- The other part of the left factorized version of simple
simpleL :: ERHS -> GenParser Char st ERHS
simpleL erhs =  do  char '?'
                    return (EOption erhs)
            <|> do  char '*'
                    return (EMany erhs)
            <|> return erhs
```

```
-- Simple0 after a left recursion elimination done on it,
so it removes the infinite loop
simple0 :: GenParser Char st ERHS
simple0 = do     l <- atom
                 sim0 <- simple0' l
                 return sim0

-- The new rule according to the left recursion elimination
simple0' :: ERHS -> GenParser Char st ERHS
simple0' a = try (do     string "{?" -- to parse predicates
                         s <- htext
                         char '}'
                         sim0 <- simple0' a
                         return (EPred sim0 s))
        <|> return a

-- Describing
atom :: GenParser Char st ERHS
atom =  try (do c <- charLit                          -- to do charlits
                return (ESimple (SChar c)))
    <|> try (do string "\"@\""
                return (ESimple SAnyChar))
    <|> try (do s <- tokLit
                return (ESimple (SLit s)))
    <|> try (do char '('
                alt <- alts -- nested stuff
                char ')'
                return alt)
    <|> do  n <- name
            return (ESimple (SNTerm n))

charLit :: GenParser Char st Char
charLit = do     char '\''
                 c <- anyChar
                 char '\''
                 return c

tokLit :: GenParser Char st String
        -- for a longer sequence of text
tokLit = try ( do    char '\"'
                     -- to make sure its a printable character
                     c <- satisfy (\c -> isPrint c)
                     s <- many1 (satisfy (\c -> isPrint c && c /= '\"'))
                     char '\"'
                     return ((++) [c] s))
-- checking for a single character (The one above should be enough >>
```

```haskell
and this one never reached, if it would pass thih one)
            <|> do  char '\"'
                    c <- satisfy (\c -> isPrint c)
                    char '\"'
                    return [c]

htext :: GenParser Char st String
htext = do  --c <- satisfy (\c -> c /= ':' && c /= '?' && c /= '}')
            s <- many1 (satisfy (\c -> c /= '}'))
            return (s)

name :: GenParser Char st String -- for use throughout the implementation
name = do   c <- satisfy (\c -> isLetter c || c == '_')
            s <- nameEnd c
            return s

-- shared part of the name that both the general and the lhs part uses
nameEnd :: Char -> GenParser Char st String
nameEnd c = try (do s <- many1 (
    satisfy (\cx -> isAlphaNum cx || cx == '_' || isDigit cx)
    )
                    return ([c] ++ s))
            <|> return [c]

-- is used to get the rkind of for the lhs name
lhsName :: GenParser Char st (String, RKind)
lhsName = try (do   c <- char '_'
                    s <- nameEnd c
                    return (s, RSep))
        <|> try (do c <- satisfy (\c -> isLower c)
                    s <- (nameEnd c)
                    return (s, RToken))
        <|> do  s <- name
                return (s, RPlain)
```

TransformerImpl

```haskell
-- Put yor transformer implementation in this file
module TransformerImpl where

import Definitions


-- used to convert from EBNF (EGrammar) to BNF (Grammar)
-- should in the future implement simple checks on the grammar also.
```

```
convert :: EGrammar -> EM Grammar
convert [] = return []
convert eGrammar =
    case eGrammar of
    x:xs -> case x of
-- creating our current rules and all the new rules that is found
-- doing the conversion.
        ((name, kind, t), erhs) -> do  let (a,b) = createRule erhs name "" in
            (do  ys <- convert xs; return (([((name, kind, t), b)] ++ (a)) ++ ys))

createRule :: ERHS -> String -> String -> ([Rule], [([Simple], Action)])
createRule erhs name htext =
        case erhs of
-- since we are sending another eseq all the time we gotta
-- catch when there is none left
            ESeq [] text -> ([], [])
            -- calculating the nested conversion
            ESeq (x:xs) text -> case nested x name text of
-- sending back to find the next on from the ESeq the same way as above
                (newRule, oldRule) -> case createRule (ESeq xs text) (name) text of
-- adding together the current rule we are building
                    (newRule2, oldRule2) -> case oldRule2 ++ oldRule of
                        [] -> (newRule ,[([], AUser text)])
                        y:ys ->
                            -- adding together the rules
                            ((newRule ++ newRule2),
                            [
-- this just adds together all of the simples into an array on the
--left hand side of the tuple
                                foldr
                                    (\ (sa, a) (sa2, a2) ->
                                        ((++) sa sa2, a)
                                    ) y ys
                            ])
            EBar erhs1 erhs2 ->
-- everything after here would be nested and therefore call each
-- side of the bar as nested
                case nested erhs1 name htext of
                    (newRules, oldRule) ->
                        case nested erhs2 name htext of
-- just adding all new rules from each side as well as the left sides together
                            (newRules2, oldRule2) ->
                                (newRules ++ newRules2, oldRule ++ oldRule2)
            EOption erhs2 -> eoptionCase erhs2 name htext
            EMany erhs2 -> emanyCase erhs2 name htext
            ENot erhs2 -> enotCase erhs2 name htext
            -- if its a simple just simply add it
            ESimple s -> ([], [([s], AUser htext)])

-- works mostly like the one above but takes hand of things when they are nested.
nested :: ERHS -> String -> String -> ([Rule], [([Simple], Action)])
nested erhs name htext =
    case erhs of
        ESeq [] text -> ([], [])
        ESeq (x:xs) text -> case nested x name text of
            (newRule, oldRule) -> case nested (ESeq xs text) name text of
                (newRule2, oldRule2) -> case oldRule2 ++ oldRule of
                    [] -> (newRule ,[([], AUser text)])
                    y:ys ->
                        ((newRule ++ newRule2),
                        [
                            -- same as in create rule
                            foldr
                                (\ (sa, a) (sa2, a2) ->
                                    ((++) sa sa2, a)
                                ) y ys
                        ])
        EBar erhs1 erhs2 ->
            case createRule erhs1 name htext of
                (newRules, oldRule) ->
                    case createRule erhs2 name htext of
                        (newRules2, oldRule2) ->
                            (
-- if an Ebar is used here a new rules should be made so
-- we add all new ones each side creates
-- and creates a new one for this
                        (newRules ++ newRules2) ++
                        [
                            ((mark_name name, RPlain, Nothing), oldRule ++ oldRule2)
                        ],
                        [
                            ([(SNTerm (mark_name name))], AVar (mark_name name))
                        ]
                            )
        EOption erhs2 -> eoptionCase erhs2 name htext
        EMany erhs2 -> emanyCase erhs2 name htext
        ENot erhs2 -> enotCase erhs2 name htext
        ESimple s -> ([], [([s], AUser htext)])
        a -> error (show a)

-- creates the rules for what to do if there exist an option in the EGrammar
eoptionCase :: ERHS -> String -> String -> ([Rule], [([Simple], Action)])
eoptionCase erhs name htext =
    case createRule erhs name htext of
        (newRules, oldRule) ->
(
    (
        -- adding the new rules calculated from the ERHS
        newRules ++
        [
-- The new rule created is the already created old rule but with a nothing
            ((opt_name name, RPlain, Nothing), (oldRule ++ [([], AUser "Nothing")]))
        ]
    ),
    [
        -- here we should just add the new stuff that is necesarry
        ([SNTerm (opt_name name)], AVar (opt_name name))
    ]
)

-- creates the rules for what to do if there exist an EMany in the EGrammar
emanyCase :: ERHS -> String -> String -> ([Rule], [([Simple], Action)])
emanyCase erhs name htext =
    case nested erhs name htext of
        (newRules, oldRule) ->
            case oldRule of
                x:xs -> -- Always consists of one element, from how it is build
                    case x of
                        (simples, action) ->
                            (
(
    newRules ++
    [
-- creating a new rule, using the simples from the
-- old rule and the action with the new one and an empty one
        ((many_name name, RPlain, Nothing),
        [(simples ++ [SNTerm (many_name name)], action), ([], AUser "Nothing")])
    ]
),
[
    -- just creating a simple call towards them
    ([SNTerm (many_name name)], AVar (many_name name))
]
                            )
-- creates the rules for what to do if there exist an ENot in the EGrammar
enotCase :: ERHS -> String -> String -> ([Rule], [([Simple], Action)])
enotCase erhs name htext =
    case nested erhs name htext of
```

```
        (newRules, oldRule) ->
            case oldRule of
                x:xs -> -- Always consists of one element, from how it is build
                    case x of
                        (simples, action) ->
                            (
                                (
                                    newRules
                                ),
                                [
                                    (setSNotOnArray simples, AVar (many_name name))
                                ]
                            )

-- set SNot infron of all the simples given as input
setSNotOnArray :: [Simple] -> [Simple]
setSNotOnArray [] = []
setSNotOnArray (s:sx) = [SNot s] ++ (setSNotOnArray sx)

-- adds 'opt to a given string
opt_name :: String -> String
opt_name s = s ++ "'opt"

-- adds 'm to a given string
many_name :: String -> String
many_name s = s ++ "'m"

-- adds ' to a given string
mark_name :: String -> String
mark_name s = s ++ "'"

-- (RLHS, [([Simple]{-seq-}, Action)]{-alts-})
-- convert [(("_", RPlain, Nothing ), ESimple (SLit "123"))]
-- convert [(("_", RPlain, Nothing ), ESeq [ESimple (SLit "123")] "123")]

{--
Supposed to be removing lre is not doing it
correctly and creates a mess and is not working at all...
--}
lre :: Grammar -> EM Grammar
lre [] = return []
lre grammar =
    case grammar of
        x:xs ->
            case x of
                ((name, kind, t), xs) ->
```

```haskell
                -- gets new rules as well as how to build this rule
                    case lreHelper name kind t xs of
                        (rules, rhs) ->
                            return ([((name, kind, t), rhs)] ++ rules)

-- gonna go over each element of the array to check for left recursion
lreHelper :: String -> RKind -> Maybe Type ->
    [([Simple], Action)] -> ([Rule], [([Simple], Action)])
lreHelper name kind t rhs =
    case rhs of
        x:xs ->
            case lreHelper2 name kind t x of
                (rules, simples, simple, action) ->
                    ((rules, (lreAdd [(simples, action)] simple) ++
                    [([simple], action)]))

-- Gonna use an element to eliminate the left recursion
lreHelper2 :: String -> RKind -> Maybe Type ->
    ([Simple],Action) -> ([Rule], [Simple], Simple, Action)
lreHelper2 name kind t elem =
    case elem of
        ((x:xs), action) ->
            case x of
                SNTerm name ->
                    (
                        [
                            -- A'
                            ((name, kind, t),
                            [(
                            (xs ++ [SNTerm (name ++ "'v")]), action),
                            ([], AUser "Nothing")])
                        ],
                        xs,
                        SNTerm (name ++ "'v"),
                        action
                    )

-- adds a simple to each array of [Simple] in [([Simple], Action)]
lreAdd :: [([Simple], Action)] -> Simple -> [([Simple], Action)]
lreAdd [] s = []
lreAdd (x:xs) s =
    case x of
        (ys, action) ->
            if length ys > 0    then [(ys ++ [s], action)] ++ (lreAdd xs s)
                                else [([], action)] ++ (lreAdd xs s)
```

```
{--
Not implemented will just return the same grammar as is given.
--}
lfactor :: Grammar -> EM Grammar
lfactor grammar = return grammar


{--
[
    (
        ("digit",RToken,Nothing),
        [
            (
                [
                    SNTerm "a"
                ],
                AUser ""
            ),
            (
                [
                    SNTerm "digit'"
                ],
                AVar "digit'"
            )
        ]
    ),
    (
        ("digit'",RPlain,Nothing),
        [
            (
                [
                    SNTerm "b"
                ],
                AUser ""
            ),
            (
                [
                    SNTerm "c"
                ],
                AUser ""
            ),
            (
                [
                    SNTerm "d"
                ],
                AUser ""
            )
        ]
    )
]
```

```
[
    (
        ("digit",RToken,Nothing),
        [
            (
                [
                    SNTerm "digit'",
                    SNTerm "digit'"
                ],
                AVar "digit'"
            )
        ]
    ),
    (
        ("digit'",RPlain,Nothing),
        [
            (
                [
                    SNTerm "c"
                ],
                AUser "abc"
            ),
            (
                [
                    SNTerm "b"
                ],
                AUser "abc"
            )
        ]
    ),
    (
        ("digit'",RPlain,Nothing),
        [
            (
                [
                    SNTerm "d"
                ],
                AUser "abc"
            ),
                (
                    [
                        SNTerm "e"
                    ],
                    AUser "abc"
                )
            ]
        )
]



[
    (
        ("digit",RToken,Nothing),
        [
            ([],AUser ""),
            (
                [
                    SNTerm "f",
                    SNTerm "c'v"
                ],
                AUser ""
            )
        ]
    ),
    (
        ("c",RToken,Nothing),
        [
            (
                [
                    SNTerm "f",
                    SNTerm "c'v"
                ],
                AUser ""
            ),
            (
                [],
                AUser "Nothing"
            )
        ]
    )
]
--}
```

erlst.erl

```erlang
-module(erlst).

% You are allowed to split your Erlang code in as many files as you
% find appropriate.
% However, you MUST have a module (this file) called erlst.

% Export at least the API:
-export([launch/0,
         shutdown/1,
         open_account/2,
         account_balance/1,
         make_offer/2,
         rescind_offer/2,
         add_trader/2,
         remove_trader/2
        ]).

% You may have other exports as well
-export([get_state/1]).

-import(rand, [uniform/3]).

-type stock_exchange() :: term().
-type account_id() :: term().
-type offer_id() :: term().
-type trader_id() :: term().
-type stock() :: atom().
-type isk() :: non_neg_integer().
-type stock_amount() :: pos_integer().
-type holdings() :: {isk(), [{stock(), stock_amount()}]}.
-type offer() :: {stock(), isk()}.
-type decision() :: accept | reject.
-type trader_strategy() :: fun((offer()) -> decision()).

%Simple function to gain state of a server as described in the server
get_state(S) ->
  S ! {self(), {get_state}},
  ok.

-spec launch() -> {ok, stock_exchange()} | {error, term()}.
%returning {ok, E} unless it cant start the stock exchange
launch() ->
    try
        E = spawn(fun() -> loop(1, [], #{}) end),
        {ok, E}
    catch
        % Catching all errors, for security measures, never
        % interested in throwing an exception that isnt handled
        _:Reason -> {error, Reason}
    end .

-spec shutdown(S :: stock_exchange()) -> non_neg_integer().
shutdown(S) ->
    request_reply(S, kill_process).

-spec open_account(S :: stock_exchange(), holdings()) -> account_id().
open_account(S, Holdings) ->
    request_reply(S, {open_acc, Holdings}).

-spec account_balance(Acct :: account_id()) -> holdings().
account_balance(Acc) ->
    request_reply(Acc, get_balance).

-spec make_offer(Acct :: account_id(), Terms :: offer()) ->
    {ok, offer_id()} | {error, term()}.
make_offer(Acc, Offer) ->
    request_reply(Acc, {make_offer, Offer}).

-spec rescind_offer(Acct :: account_id(), Offer :: offer_id()) -> ok.
rescind_offer(Acc, OfferId) ->
    Acc ! {self(), {rescind, OfferId}}, % non-blocking
    ok.

-spec add_trader(Acct :: account_id(), Strategy :: trader_strategy()) ->
    trader_id().
add_trader(Acc, Strategy) ->
  request_reply(Acc, {add_trader, Strategy}).

-spec remove_trader(Acct :: account_id(), Trader :: trader_id()) -> ok.
remove_trader(Acc, TraderId) ->
    Acc ! {self(), {remove_trader, TraderId}}, % non-blocking
    ok.

% Auxilliary function for requesting replies from the main server loop with
% Pid:Pid and with the request:Request
% Taken from course slides, and used in the same way as described there
request_reply(Pid, Request) ->
    Pid ! {self(), Request},
    receive
      {Pid, Response} ->
```

```
            Response
        end .

% Stock exchange server
loop(Counter, Offers, Accounts) ->
    receive
      {_, {get_state}} ->
        % Writes out the state to IO
        io:fwrite('Counter: ~w\nOffers: ~w\nAccounts: ~w\n',
        [Counter, Offers, Accounts]),
        loop(Counter, Offers, Accounts);

      {From, get_balance} -> %returns the balance from the Accounts mapping
        #{From := {Holdings}} = Accounts,
        From ! {self(), {holdings, Holdings}},
        loop(Counter, Offers, Accounts);

      {From, {open_acc, Holdings}} ->
        Self = self(),
        % Chosen to use each account as a different process
        ID = spawn(fun() -> account_loop(Self, []) end),
        % Adding the process to our map
        NewAccounts = Accounts#{ID => {Holdings}},
        From ! {self(), ID},
        loop(Counter, Offers, NewAccounts);

      {From, {make_offer, To, Offer, Seller}} ->
        % check they have the balance
        {StockName, Price} = Offer,
        % checking if the id has 1 of the given stock
        Stocks = id_has_stock(Accounts, Seller, StockName),
        if
            % if not then send that back
          Stocks == not_sufficient_amount ->
            From ! {self(), {offer_made, To, {error, not_sufficient_amount}}},
            loop(Counter, Offers, Accounts);
% otherwise change account state to remove one of the given stock
          true ->
            #{ Seller := {{Isk, _}}} = Accounts,
            NewAccounts = Accounts#{ Seller := {{Isk, Stocks}}},
            NewOffer = {Counter, Seller, Offer, undecided},
            NewOffers = lists:append(Offers, [NewOffer]),
            % After updating the accounts we are sending a signal to
            % All existing traders
            callWithOffer(maps:keys(Accounts), {broadcast_to_traders, NewOffer}),
            % sending to the account that the offer has been made
            From ! {self(), {offer_made, To, {ok, Counter}}},
            % Adding counter for new offer ID
            NewCounter = Counter + 1,
            loop(NewCounter, NewOffers, NewAccounts)
        end;

      {From, {accept, Id}} ->
        {_,Seller,{Stock,Price},Status} = lists:nth(Id, Offers),
        if
          % If status is undecided then buy
          Status == undecided ->
            NewOffers =
                update_array_at(
                Offers, Id, {Id, Seller, {Stock, Price}, accepted}
                ),
            NewAccounts = update_holding(Accounts, From, -Price, Stock),
            NewAccounts2 = update_isk_holding(NewAccounts, Seller, Price),
            loop(Counter, NewOffers, NewAccounts2);
          true ->
            loop(Counter, Offers, Accounts)
        end;

      {From, {rescind, OfferId}} ->
        {_,Seller,{Stock,Price},Status} = lists:nth(OfferId, Offers),
        if
          Status == undecided ->
            % add offter to be rescinded
            NewOffers = update_array_at(
                Offers, OfferId, {OfferId, Seller, {Stock, Price}, rescinded}
            ),
            #{Seller := {Holdings}} = Accounts,
            {ISK, Stocks} = Holdings,
            % finding the index of the stock in your holdings
            Index = get_stock_index(0, Stocks, Stock),
            if
              Index == nope ->
                % should be impossible to reach, as I
                % never remove it from your holdings
                loop(Counter, Offers, Accounts);
              true -> % if we can find it in holding then we can add it back
                Index2 = Index + 1,
                {Stock, Amount} = lists:nth(Index2, Stocks),
                NewStocks = update_array_at(Stocks, Index2, {Stock, Amount + 1}),
                NewAccounts = Accounts#{ Seller => {{ISK, NewStocks}}},
                loop(Counter, NewOffers, NewAccounts)
            end;
```

```
          true ->
            % already bought then we cant rescind
            loop(Counter, Offers, Accounts)
        end;

      {From, kill_process} ->
        % counting for the response otherwise
        % just sending to kill all processes, and itself.
        Count = calculate_transactions(0, Offers),
        From ! {self(), Count},
        callWithOffer(maps:keys(Accounts), kill_process),
        exit(self(), kill)
    end.


account_loop(Parent, Traders) ->
  receive
    {_, {get_state}} ->
      io:fwrite('Parent: ~w\nTraders: ~w\n', [Parent, Traders]),
      account_loop(Parent, Traders);

    {From, get_balance} ->
      % just sending to recieve the balance and send it back
      Parent ! {self(), get_balance},
      receive
        {Parent, {holdings, Holdings}} ->
          From ! {self(), Holdings}
      end,
      account_loop(Parent, Traders);

    {From, {add_trader, Strategy}} ->
      Self = self(),
      % creates a new trader process
      ID = spawn(fun() -> trader_loop(Self, Strategy, []) end),
      % appends to the list of traders
      NewTraders = lists:append(Traders, [ID]),
      From ! {self(), ID},
      account_loop(Parent, NewTraders);

    {From, {make_offer, Offer}} ->
      Parent ! {self(), {make_offer, From, Offer, self()}},
      account_loop(Parent, Traders);

    {From, {decision_made, Id, Type}} ->
      if
        %If trader accepts then send that if reject do nothing
        Type == accept ->
          Parent ! {self(), {accept, Id}};
        true -> nope
      end,
      account_loop(Parent, Traders);

    {Parent, {offer_made, To, Response}} ->
      To ! {self(), Response},
      account_loop(Parent, Traders);

    {From, {broadcast_to_traders, Obj}} ->
      {_, Seller, _, _} = Obj,
      if
        Seller == self() ->
          account_loop(Parent, Traders);
        true ->
          callWithOffer(Traders, {offer_check, Obj}),
          account_loop(Parent, Traders)
      end;

    {From, {rescind, OfferId}} ->
      Parent ! {self(), {rescind, OfferId}},
      account_loop(Parent, Traders);

    {From, {remove_trader, TraderId}} ->
      % gets index of the traders
      Index = get_index_from_content(1, Traders, TraderId),
      if
        Index == nope ->
          account_loop(Parent, Traders);
        true ->
          % removes element at position
          NewTraders = remove_elem_at(Traders, Index),
          TraderId ! {self(), kill_process},
          account_loop(Parent, NewTraders)
      end;

    {Parent, kill_process} ->
      % sends kill signal to all traders
      callWithOffer(Traders, kill_process),
      exit(self(), kill)

  end.

trader_loop(Parent, Strategy, OfferProcesses) ->
  receive
```

```
    {_, {get_state}} ->
      io:fwrite('Parent: ~w\nStrategy: ~w\nStrategy: ~w\n',
      [Parent, Strategy, OfferProcesses]),
      trader_loop(Parent, Strategy, OfferProcesses);

    {Parent, {offer_check, {Id, _, Offer, _}}} ->
      % spawns new process
      PId = spawn(fun() -> calculate_trade(Parent, Strategy, Offer, Id) end),
      NewOfferProcesses = lists:append(OfferProcesses, [{Id, PId, unfinished}]),
      trader_loop(Parent, Strategy, NewOfferProcesses);

    {Parent, kill_process} ->
      kill_all_processes(OfferProcesses)
  end.

% kills all processes in the array
kill_all_processes(OfferProcesses) when OfferProcesses == [] ->
  ok;

kill_all_processes(OfferProcesses) ->
  [Head|Tail] = OfferProcesses,
  {_, Pid, Status} = Head,
  if
    Status == unfinished ->
      exit(Pid, kill)
  end,
  kill_all_processes(Tail).

% the function in a process of a trader to calculate
calculate_trade(Account, Strat, Offer, Id) ->
  try
    Result = Strat(Offer),
    if
      Result == accept -> % accepted the offer
        Account ! {self(), {decision_made, Id, accept}};
      true -> % rejected the offer, send rejection
        Account ! {self(), {decision_made, Id, reject}}
    end
  catch % error send rejection
    _:_ -> Account ! {self(), {decision_made, Id, reject}}
  end.

calculate_transactions(Counter, Array) when Array == [] ->
  Counter;

calculate_transactions(Counter, Array) ->
```

```
  [Head|Tail] = Array,
  {_, _, {_, _}, Status} = Head,
  if
    Status == accepted ->
      NewCounter = Counter + 1,
      calculate_transactions(NewCounter, Tail);
    true -> calculate_transactions(Counter, Tail)
  end.

% sending object (tuple) to all in the array
callWithOffer(Accounts, _) when Accounts == [] ->
  ok;

callWithOffer(Accounts, Obj) ->
  [Head|Tail] = Accounts,
  Head ! {self(), Obj},
  callWithOffer(Tail, Obj).

% updating both ISK and stocks of a acc
update_holding(Accounts, Id, ISKDiff, Stock) ->
  #{Id := {Holdings}} = Accounts,
  {ISK, Stocks} = Holdings,
  NewISK = ISK + ISKDiff,
  NewStock = edit_stocks(Stocks, Stock, true),
  NewAccounts = Accounts#{Id => {{NewISK, NewStock}}},
  NewAccounts.

% only updates the ISK of an account
update_isk_holding(Accounts, Id, ISKDiff) ->
  #{Id := {Holdings}} = Accounts,
  {ISK, Stocks} = Holdings,
  NewISK = ISK + ISKDiff,
  NewAccounts = Accounts#{Id => {{NewISK, Stocks}}},
  NewAccounts.


% Checks if an id has a specific stock
id_has_stock(Accounts, Id, StockName) ->
  #{Id := {Holdings}} = Accounts,
  {ISK, Stocks} = Holdings,
  Index = stock_index_in_array(0, Stocks, StockName),
  if
    Index == not_found_or_not_sufficient_amount ->
      not_sufficient_amount;
    true ->
      Index2 = Index + 1,
      {Stock, Amount} = lists:nth(Index2, Stocks),
      update_array_at(Stocks, Index2, {Stock, Amount - 1})
  end.

% checking if u have more than zero of a stock
stock_index_in_array(_, Stocks, _) when Stocks == [] ->
  not_found_or_not_sufficient_amount;

stock_index_in_array(Counter, Stocks, Stock) ->
  [Head|Tail] = Stocks,
  {StockName, Amount} = Head,
  if
    StockName == Stock ->
      if
        Amount > 0 ->
          Counter;
          true -> not_found_or_not_sufficient_amount
      end;
    true -> stock_index_in_array(Counter+1, Tail, Stock)
  end.

% gets index of a stock
get_stock_index(_, Stocks, _) when Stocks == [] ->
  nope;

get_stock_index(Counter, Stocks, Stock) ->
  [Head|Tail] = Stocks,
  {StockName, _} = Head,
  if
    StockName == Stock -> Counter;
    true -> stock_index_in_array(Counter+1, Tail, Stock)
  end.

% gets an index from some content
get_index_from_content(Counter, Array, Content) when Array == [] ->
  nope;

get_index_from_content(Counter, Array, Content) ->
  [Head|Tail] = Array,
  if
    Content == Head -> Counter;
    true -> get_index_from_content(Counter+1, Tail, Content)
  end.

% removes an element at a position
remove_elem_at(Array, Index) ->
```

```
  lists:sublist(Array, Index - 1) ++ lists:nthtail(Index, Array).

% updates an element at nth position
update_array_at(Array, Index, Object) ->
  lists:sublist(Array, Index - 1) ++ [Object] ++ lists:nthtail(Index, Array).

% edit stocks to either contain the new or update it
% to contain it
edit_stocks(Stocks, Stock, Found) when Stocks == [] ->
  if
    Found == found ->
      [];
    true ->
     [{Stock, 1}]
  end;

edit_stocks(Stocks, Stock, Found) ->
  [Head|Tail] = Stocks,
  {StockName, Amount} = Head,
  if
    StockName == Stock ->
      lists:append([{StockName, Amount + 1}], edit_stocks(Tail, Stock, found));
    true ->
      lists:append([Head], edit_stocks(Tail, Stock, Found))
  end.

% c(erlst). {ok, S} = erlst:launch().
% erlst:shutdown(S).
% erlst:get_state(S).
% ACC = erlst:open_account(S, {100, [{abc, 100}]}).
% erlst:add_trader(ACC, fun(N) -> accept end).
% erlst:account_balance(ACC).
% erlst:make_offer(ACC, {abc, 100}).
```

test_erlst.erl

```
-module(test_erlst).

-include_lib("eunit/include/eunit.hrl").
-export([test_all/0, test_everything/0]).
-export([]). % Remember to export the other functions from Q2.2


% You are allowed to split your testing code in as many files as you
% think is appropriate, just remember that they should all start with
% 'test_'.
```

```erlang
% But you MUST have a module (this file) called test_erlst.
test_all() -> eunit:test(testsuite(), [verbose]).

testsuite() ->
  [ {"Basic behaviour", spawn,
    [ test_start_server(),
      test_add_account(),
      test_add_trader(),
      test_make_offer(),
      test_rescind_offer(),
      test_shutdown_server(),
      test_make_offer_rescind_with_trader()
    ]
  }
  ].

test_everything() ->
  test_all().

% c(erlst), c(test_erlst), test_erlst:test_all().

test_start_server() ->
  {"Starting server",
    fun () ->
      ?assertMatch({ok, _}, erlst:launch())
    end }.

test_shutdown_server() ->
  {"Starting server and shutting it down",
    fun () ->
      {ok, S} = erlst:launch(),
      ?assertMatch(true ,is_process_alive(S)),
      ?assertMatch(0, erlst:shutdown(S)),
      timer:sleep(500),
      ?assertMatch(false ,is_process_alive(S))
    end }.

test_add_account() ->
  {"Add account to stock exchange",
    fun () ->
      {ok, S} = erlst:launch(),
      Acc = erlst:open_account(S, {200, [{ll, 1}, {ff, 2}]}),
      ?assertMatch({200, [{ll, 1}, {ff, 2}]},erlst:account_balance(Acc))
    end }.

test_make_offer() ->
```

```
  {"Make an offer on the stock exchange",
    fun () ->
      {ok, S} = erlst:launch(),
      Acc = erlst:open_account(S, {200, [{ll, 1}, {ff, 2}]}),
      Acc2 = erlst:open_account(S, {1000, []}),
      Strat = fun(N) -> timer:sleep(5000), accept end,
      Strat2 = fun(N) -> accept end,
      TraderId = erlst:add_trader(Acc, Strat),
      TraderId2 = erlst:add_trader(Acc2, Strat2),
      ?assertMatch({ok, 1}, erlst:make_offer(Acc, {ll, 500})),
      timer:sleep(1000),
      ?assertMatch({500, [{ll, 1}]}, erlst:account_balance(Acc2))
    end }.

test_make_offer_rescind_with_trader() ->
  {"Make an offer and rescind it, before it can be bought",
    fun () ->
      {ok, S} = erlst:launch(),
      Acc = erlst:open_account(S, {200, [{ll, 1}, {ff, 2}]}),
      Acc2 = erlst:open_account(S, {1000, []}),
      Strat = fun(N) -> timer:sleep(1000), accept end,
      TraderId = erlst:add_trader(Acc, Strat),
      TraderId2 = erlst:add_trader(Acc2, Strat),
      {ok, OfferId} = erlst:make_offer(Acc, {ll, 500}),
      erlst:rescind_offer(Acc, OfferId),
      ?assertMatch(ok, erlst:rescind_offer(Acc, OfferId)),
      ?assertMatch(true ,is_process_alive(S)),
      ?assertMatch(true ,is_process_alive(S)),
      timer:sleep(500),
      ?assertMatch({200, [{ll, 1}, {ff, 2}]}, erlst:account_balance(Acc)),
      ?assertMatch({1000, []}, erlst:account_balance(Acc2)),
      erlst:shutdown(S),
      timer:sleep(500),
      ?assertMatch(false ,is_process_alive(S)),
      ?assertMatch(false ,is_process_alive(S))
    end }.

test_rescind_offer() ->
  {"Rescind offer from the stock exchange",
    fun () ->
      {ok, S} = erlst:launch(),
      Acc = erlst:open_account(S, {200, [{ll, 1}, {ff, 2}]}),
      Strat = fun(N) -> accept end,
      TraderId = erlst:add_trader(Acc, Strat),
      {ok, OfferId} = erlst:make_offer(Acc, {ll, 500}),
      ?assertMatch(ok, erlst:rescind_offer(Acc, OfferId))
    end }.

test_add_trader() ->
  {"Add trader to an account",
    fun () ->
      {ok, S} = erlst:launch(),
      Acc = erlst:open_account(S, {200, [{ll, 1}, {ff, 2}]}),
      Strat = fun(N) -> accept end,
      TraderId = erlst:add_trader(Acc, Strat),
      ?assertMatch(ok, erlst:remove_trader(Acc, TraderId))
    end }.
```