Exam Report 2022

Yaokun Li (xnf483@alumni.ku.dk)

Contents

1 APpy: A simple parser generator Question 1.1

1.1 Design and Implementation

We are asked to implement a parser for grammar of an APpy specification. To elimate ambiguity in the grammar, I rewrite it. At the same time, using left recursive elimination skills, the original grammar can be transformed to following:

To skip useless newlines and comments, every time parsing a keyword or identity the program look ahead to skip following space and comment. Then we can then guarantee there will be no space before next expression.

For other low-level implementation details, I have commented in my code.

1.2 Testing

I tried to design test cases from the point of type ERHS. For every type of ERHS, I designed test cases to make sure that my program can parser these cases correctly.

I also write codes to test the given examples in the file src/Test.hs. My parser can parse boa.appy and calc.appy correctly. It proves that my program is able to handle comment and white space correctly.

2 APpy: A simple parser generator Question 1.2

I don't have enough time to implement the task in this part. Just leave some try in Transformer.hs. And I have commented them out.

3 APpy: A simple parser generator Assessment

3.1 Test

My program have passed all my test cases. As for the design of test cases, I designed test case covering all required API, making sure they can both execute and return correctly.

3.2 Completeness

Only the asked-for functionality in Question 1.1 are implemented. I don't have enough to do the Question 1.2.

3.3 Correctness

All implemented functionality work correctly and there is no known bugs.

```
Spec ::= preamble ERules.

ERules ::= ERule | ERule ERules.

ERule ::= LHS "::=" Alts ".".

LHS ::= name OptType| name | "_".

OptType ::= "{:" htext "}".

Alts ::= Seq |
        Seq "|" Alts.

Seq ::= Simple | Simplez "{" htext "}".

Simplez ::= | Simple Simplez.

Simple ::=
        Simple0 |
        Simple0 "?" |
        Simple0 "*" |
         "!" Simple0.

Simple0 ::= Atom Simple0Helper.

Simple0Helper = |"{?" htext "}" Simple0Helper.

Atom ::= name |
        tokLit |
        "@" |
        charLit |
        "(" Alts ")".
```

Figure 1: Grammar

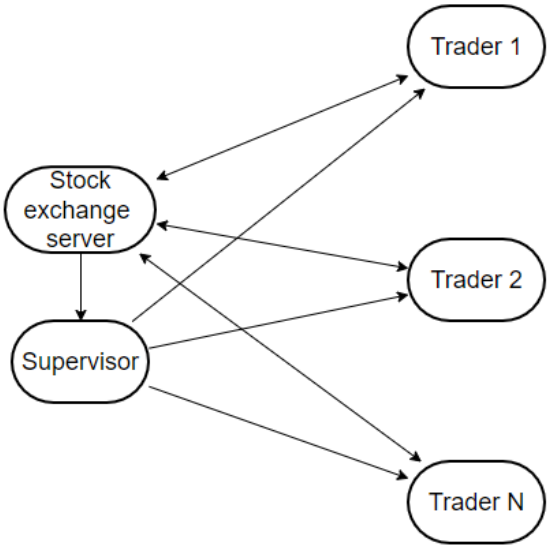

Figure 2: Stock exchange system processes

4 Stock Exchange Question 2.1

4.1 Overall

We are asked to implement a system allowing users to open accounts that can hold money and stocks. The fundamental event in the system is a trade, which happens when one user has made an offer at the exchange while another user has installed a trader that accepts the given offer.

My implementation is complete and can support all required API. The system is robust enough in the face of trader strategies that take a long time, or even forever, to compute a decision. In other words, my system does not make much assumptions to make the task easier. My stock exchange implementation can handle following cases:

1. Shutting down the stock exchange will not be delayed because of slow traders.
2. If a trader has accepted and is able to execute a trade, the stock exchange will execute the trade without waiting for the decisions of other—possibly slow—traders.
3. Any execution of a trader strategy function will be stopped when the outcome no longer has any relevance, e.g. if the offer has been rescinded or the trader has been removed.

4.2 Implementation Detail

4.2.1 Processes

To support time-consuming trader strategy function, the stock exchange system will introduce other processes to handle it. First, when then system launches, it will will also start a supervisor process, which is used to manage all trader processes (The implementation of this is in erlst.erl/launch()). When new trader is added into our system, the stock exchange system will launch a new process which will run the strategy function on current offers, **which means my system will use 1+1+N processes in total, where N means current alive trader number**. These three kinds of processes are implemented in different files: stock exchange server in erlst.erl, supervisor in sup.erl, trader in trader.erl. And both stock exchange server (erlst.erl) and trader (trader.erl) are implemented through gen_server. The supervisor (sup.erl) is implemented through supervisor behaviour.

As for the responsibility of each process, **the trader process only execute strategy function on offers and if any offer is accepted, send request to stock exchange server. The exchange server will decide whether the trade will be executed.** If the offer is rejected, the trader process just delete it from its own maintained data.

The stock exchange server basically is responsible for all the public API, including opening account, querying account balance, making offer, rescinding offer, adding and removing traders, and of course shutdown. As mentioned before, the server will decide whether the accepted trade will be executed for the reason that only the server process has stored all information it needs to judge whether the trade is valid.

The supervisor process is only existed for three things:

1. starting trader processes when launching.
2. killing trader processes when the stock exchange server shutdowns.
3. restarting trader processes, when rescinding offer.

4.2.2 Interactions between Processes

When there is requirement of message passing between trader processes and stock exchange server process, they call the corresponding API directly. To enable the communication between server and trader, I add one more API in erlst: accept_offer. **When trader accepts offer, it will call this function to notify server**. And the trader only needs to provide one API function : new_offer. When server get new offer, it will use this function to notify all traders.

When making new offer, the server will notify all traders with new offer. If the trader is currently running strategy function, they will get this new offer until the end of the strategy function. Otherwise, they will get the offer and run it immediately.

When rescinding offer, the server will terminate all trader process and restart them with updated offers. The reason for this implementation is for simpleness. To achieve only stopping the trader running strategy on the rescinded offer,

the server has to store the offer every trader is running on, which require more message passing. It will definitely introduce complexity into the system. So I just stop all trader processes and restart them, it's not elegant but simple.

When creating trader, the server will start a new trader process running the corresponding strategy on current offer sets through supervisor.

When removing trader, the server will just kill the trader process through supervisor whether the trader is running or not.

when shutdown, the server will kill all traders through supervisor first and then stop itself.

4.2.3 Data Structure

Most of the data are stored in the state of server process. To illustrate the data structure, I put a figure (Figure 2) of the data in server from logging directly. **The state of server process consists of 5 parts: account data, offer data, trader data, supervisor pid and executed trade number.** The structure of account data , offer data an trader data are alike: they are all modeled as map, using corresponding id as key, and value stores all information.

As for trader process, it keeps strategy function, server pid, current offers, accountId, traderId. Supervisor process keeps nothing.

4.3 Testing

I have written test cases covering API including launch(), shutdown(S), open_account(S, Holdings),account_balance(Acct),make_offer(Acct, Offer),rescind_offer(Acct, OfferID),add_trader(Acct, Strategy),remove_trader(Acct, TraderID). They are in file test_eunit_erlst.erl. I use assert operation to make sure my code are correct. However, the test including multi processes, sometimes the order of execution is uncertain. To make sure my code are right, I test them through observing the log (So, you will see a lot of logging if you run my codes).

I have designed strategy function that last long time or just cause error to test whether my system can handle it. The result proves it is robust enough to deal with all of these cases. My test cases also show that following requirements are all met:

1. Shutting down the stock exchange should not be delayed because of slow traders.
2. If a trader has accepted and is able to execute a trade, the stock exchange should execute the trade without waiting for the decisions of other—possibly slow—traders.
3. Any execution of a trader strategy function should be stopped when the outcome no longer has any relevance, e.g. if the offer has been rescinded or the trader has been removed.

```
{se_server_data,{account_data,13,
                              #{{1,<0.596.0>} => {19,[{c,20},{e,107},{b,6}]},
                                {2,<0.596.0>} => {4,[{c,62},{d,64},{e,25}]},
                                {3,<0.596.0>} =>
                                    {6,[{a,11},{b,81},{d,29},{e,66}]},
                                {4,<0.596.0>} =>
                                    {6,[{a,21},{c,20},{d,37},{e,28}]},
                                {5,<0.596.0>} =>
                                    {14,[{a,38},{b,27},{c,41},{e,32}]},
                                {6,<0.596.0>} =>
                                    {10,[{a,41},{d,32},{e,4},{b,57}]},
                                {7,<0.596.0>} =>
                                    {6,[{a,53},{b,73},{c,22},{d,64}]},
                                {8,<0.596.0>} =>
                                    {14,[{a,39},{b,86},{c,57},{d,35}]},
                                {9,<0.596.0>} => {19,[{a,76},{b,15},{e,88}]},
                                {10,<0.596.0>} =>
                                    {23,[{a,41},{b,20},{c,48},{e,36}]},
                                {11,<0.596.0>} =>
                                    {8,[{a,19},{b,51},{c,4},{d,21}]},
                                {12,<0.596.0>} => {14,[{b,2},{d,1},{e,25}]},
                                {13,<0.596.0>} => {8,[{d,13}]}}},
               {offer_data,8,
                           #{{2,<0.596.0>} => {{7,<0.596.0>},{e,4}},
                             {3,<0.596.0>} => {{8,<0.596.0>},{a,7}},
                             {4,<0.596.0>} => {{9,<0.596.0>},{c,12}},
                             {5,<0.596.0>} => {{10,<0.596.0>},{a,25}},
                             {6,<0.596.0>} => {{11,<0.596.0>},{d,20}},
                             {7,<0.596.0>} => {{12,<0.596.0>},{d,12}},
                             {8,<0.596.0>} => {{13,<0.596.0>},{c,9}}}},
               {trader_data,5,
                            #{1 =>
                                  {<0.598.0>,#Fun<test_erlst.16.54227401>,
                                   {1,<0.596.0>}},
                              2 =>
                                  {<0.599.0>,#Fun<test_erlst.14.54227401>,
                                   {2,<0.596.0>}},
                              3 =>
                                  {<0.600.0>,#Fun<test_erlst.13.54227401>,
                                   {3,<0.596.0>}},
                              4 =>
                                  {<0.601.0>,#Fun<test_erlst.14.54227401>,
                                   {4,<0.596.0>}},
                              5 =>
                                  {<0.602.0>,#Fun<test_erlst.14.54227401>,
                                   {5,<0.596.0>}}}},
               <0.597.0>,1}
```

Figure 3: Stock exchange system data structure

5 Stock Exchange Question 2.2

5.1 Overall

All the requirement are met. mkstrategy(Opr), reliable_strategy(), prop_value_preservation and prop_total_trades are all implemented. All the unit test, which I mentioned in 2.3 Testing, are implemented in file test_eunit_erlst.erl, the four functions just mentioned are in test_erlst.erl

5.2 Implementation Detail

The core of designing property-based testing is how to simulating the process of trade. My implementation first generate a stock exchange server using erlst:launch, then generate offer and trader randomly. When generating offers and traders, account will be generated. Then the trader will execute strategy automatically. To test remove_trader, my implementation will also remove one trader with half probability. The test case need to query account balance from the server. To make sure there is no mistakes caused by multi-processes, my property-based function needs to remove all trader first, then sleep 100ms and finally query the balance information.

As for the quality of the generated test case, I run collect function in eqc_gen to report the quality of test case. And I put part of the result in figure 2. As you can see, 16% of the trader list are empty, most trader list is filled with at least one trader. It shows that I have tested trader with multi traders and trader strategies. The quality of generated offer list is similar.

For test cases that does not cover here, it is tested by unit test. You can find the report of unit test in erlang part1 testing.

My program can pass all test cases.

6 Stock Exchange Assessment

6.1 Test

My program have passed all my test cases. As for the design of test cases, I designed test case covering all required API, making sure they can both execute and return correctly. I have designed property-based test to further test my program.

6.2 Completeness

All the asked-for functionality are implemented.

6.3 Correctness

All implemented functionality work correctly and there is no known bugs.

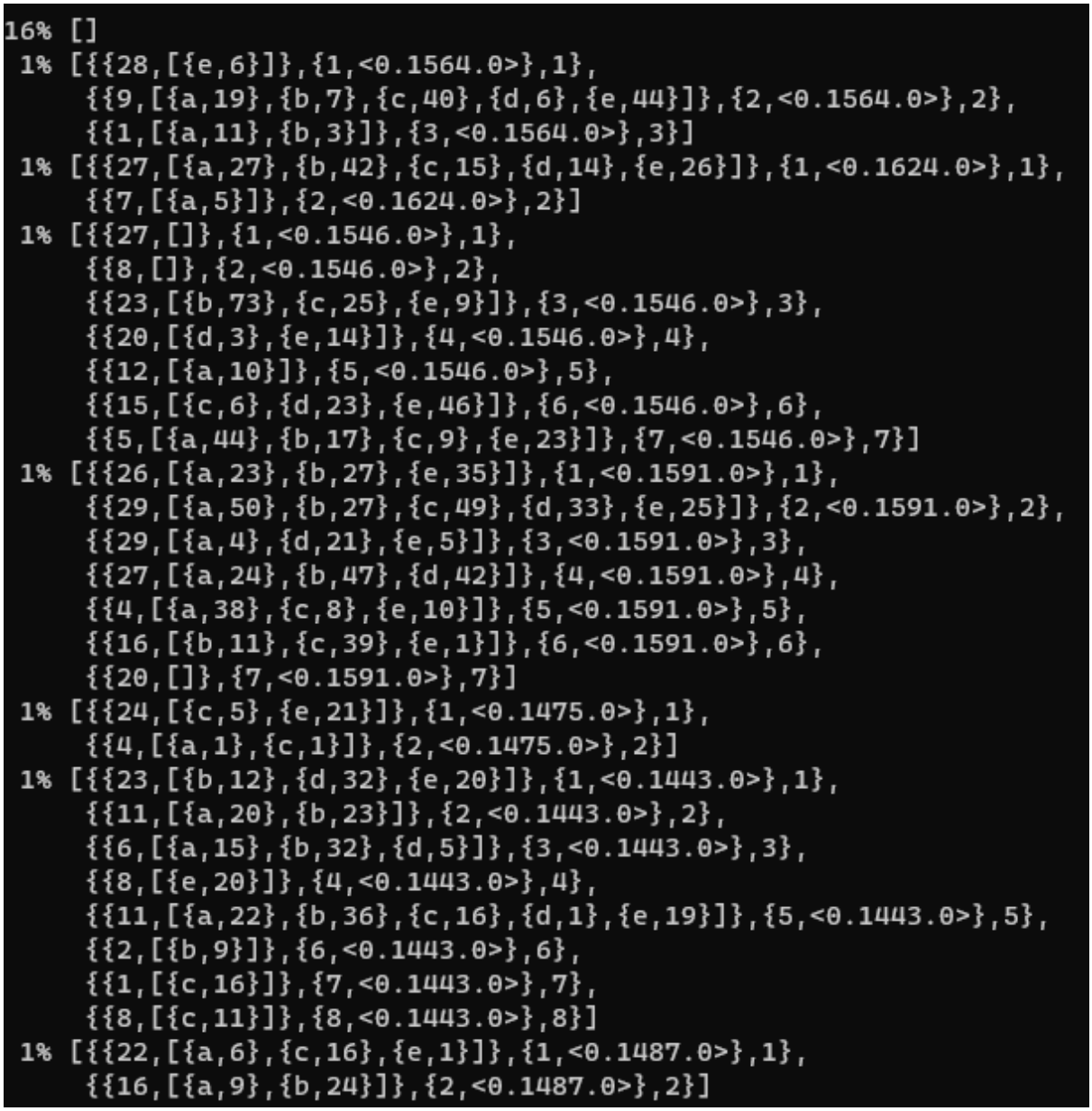

Figure 4: Stock exchange system test cases quality: quality of trader list. The first element of the generated trader is the holding of corresponding account, the second element is the accountId of the corresponding account, and the last is traderId.

6.4 Robustness

Due to the multi-processes design, my program is robust enough to handle long-running strategy function and error strategy function. And my program does not waste time calculating offer when the outcome no longer has any relevance. And when shutdown, my program will not be blocked by long-running strategy function and terminates immediately.

7 Code

7.1 Haskell

7.1.1 ParserImpl.hs

```haskell
-- Put yor parser implementation in this file
{-# OPTIONS_GHC -Wno-unrecognised-pragmas #-}
{-# HLINT ignore "Use when" #-}
module ParserImpl (parseSpec) where

import Definitions
import Text.ParserCombinators.ReadP

import Data.Char
import qualified Data.Map as Map
import GHC.Conc (pseq)
import Data.Functor.Contravariant (phantom)

type ParseError = String
type Parser a = ReadP a

parseSpec :: String -> EM (String, EGrammar)
parseSpec program = case readP_to_S pSpec program of
[] -> Left "empty parsing!!!"
x -> case x of
[ (e, "")] -> Right e
ls -> Left ("error! and the current parsing result is : " ++ show ls)

-- Spec ::= preamble ERules.
pSpec :: Parser (String, EGrammar)
pSpec = do
s <- pPreamble
skipSpaceAndComment
erules <- pERules
eof
return (s, erules)

{-
```

```
    if there is preamble then parse it, otherwise return empty string
       as preamble
    -}
pPreamble :: Parser String
pPreamble = do
manyTill get (string "---\n")
<++
return ""

-- ERules ::= ERule | ERule ERules.
pERules :: Parser [ERule]
pERules = do
rule <- pERule
return [rule]
+++
do
rule <- pERule
rules <- pERules
return $ rule:rules

-- ERule ::= LHS "::=" Alts ".".
pERule :: Parser ERule
pERule = do
lhs <- pLHS
string "::="
skipSpaceAndComment
alts <- pAlts
string "."
skipSpaceAndComment
return (lhs, alts)

-- LHS ::= name OptType | name | "_".
pLHS :: Parser (NTName, RKind, Maybe Type)
pLHS = do
x <- string "_"
skipSpaceAndComment
return ("_", getLHSKind x, Nothing)
+++
do
(SNTerm name) <- pName
return (name, getLHSKind name, Nothing)
+++
do
(SNTerm name) <- pName
optType <- pOptType
return (name, getLHSKind name, Just $ AUser optType)

-- get corresponding RKind according the name of LHS
getLHSKind :: String -> RKind
```

```
getLHSKind lhsName = case lhsName of
"_" -> RSep
x:_ | isLower x-> RToken
_ -> RPlain


-- OptType ::= "{:" htext "}".
pOptType :: Parser HText
pOptType = do
string "{:"
    skipSpaceAndComment
    htext <- pHtext False
    string "}"
skipSpaceAndComment
return htext

{-
    Any sequence of arbitrary characters, including any leading
       whitespace. Characters
    { and } to be included in the sequence must be written as {{ and
       }}, respectively.
    When following an opening {, the htext must not start with a : or
       ?.

       Bool : Whether the htext is following an opening {
          -}
       pHtext :: Bool -> Parser HText
       pHtext isFollowPara = if isFollowPara
       then do
       c <- satisfy (\x -> x /= ':' && x /= '?')
       s <- pHtextGetString [c]
       skipSpaceAndComment
       return s
       else do
       s <- pHtextGetString ""
       skipSpaceAndComment
       return s

       pHtextGetChar :: Parser Char
       pHtextGetChar = do
       string "{{"
              return '{'
                  <++
                  do
                  string "}}"
          return '}'
       <++
       do
       s <- look
```

```
        if head s == '{' || head s == '}' then
        return pfail "{ or } should be doubled in htext"
        else
        get

        pHtextGetString :: String -> Parser String
        pHtextGetString s = do
        c <- pHtextGetChar
        pHtextGetString (s ++ [c])
        <++ do return s


        -- Alts ::= Seq |Seq "|" Alts.
        pAlts :: Parser ERHS
        pAlts = do
        seq <- pSeq
        string "|"
        skipSpaceAndComment
        EBar seq <$> pAlts
        +++
        do
        pSeq

        -- Seq ::= Simple | Simplez "{" htext "}".
        pSeq :: Parser ERHS
        pSeq = do
        pSimple
        +++
        do
        simplez <- pSimplez
        string "{"
            skipSpaceAndComment
            htext <- pHtext True
            string "}"
        skipSpaceAndComment
        return $ ESeq simplez htext

        -- Simplez ::= | Simple Simplez.
        pSimplez :: Parser [ERHS]
        pSimplez = do
        many pSimple

        -- Simple ::= Simple0 | Simple0 "?" | Simple0 "*" | "!"
            Simple0.
        pSimple:: Parser ERHS
        pSimple = do
        pSimple0
        +++
        do
```

```
        simple0 <- pSimple0
        string "?"
        skipSpaceAndComment
        return $ EOption simple0
        +++
        do
        simple0 <- pSimple0
        string "*"
        skipSpaceAndComment
        return $ EMany simple0
        +++
        do
        string "!"
        skipSpaceAndComment
        ENot <$> pSimple0

        -- Simple0 ::= Atom Simple0Helper.
        pSimple0 :: Parser ERHS
        pSimple0 = do
        atom <- pAtom
        pSimple0Helper atom

        -- Simple0Helper = "{?" htext "}" Simple0Helper.
        pSimple0Helper :: ERHS -> Parser ERHS
        pSimple0Helper erhs = do
        string "{?"
            skipSpaceAndComment
            htext <- pHtext False
            string "}"
        skipSpaceAndComment
        simple0Helper <- pSimple0Helper erhs
        return $ EPred simple0Helper htext
        +++
        return erhs

        -- Atom ::= name | tokLit | "@" | charLit | "(" Alts ")".
        pAtom :: Parser ERHS
        pAtom = do
        ESimple <$> pName
        +++
        do
        ESimple <$> pTokLit
        +++
        do
        string "@"
        skipSpaceAndComment
        return $ ESimple SAnyChar
        +++
        do
```

```
        ESimple <$> pCharLit
        +++
        do
        string "("
        skipSpaceAndComment
        alts <- pAlts
        string ")"
        skipSpaceAndComment
        return alts

        -- Any printable character (including '), enclosed in
            single-quotes.
        pCharLit :: Parser Simple
        pCharLit = do
        s <-pCharLitString
        skipSpaceAndComment
        return $ SChar s

        pCharLitString :: Parser Char
        pCharLitString = do between (char '\'') (char '\'') (satisfy
            isPrint)

        -- Any non-empty sequence of printable characters, enclosed in
            double-quotes. If the
        -- sequence is itself to contain a double-quote, it must be
            written as "".
        pTokLit :: Parser Simple
        pTokLit = do
        s <- pTokLitString
        skipSpaceAndComment
        return $ SLit s

        pTokLitString :: Parser String
        pTokLitString = do
        string "\""
        tokLit <- pTokLitStringHelper
        string "\""
        return tokLit

        pTokLitStringHelper :: Parser String
        pTokLitStringHelper = do
        x <- getOneChar
        getMoreChar x

        getOneChar :: Parser String
        getOneChar = do
        string "\"\""
        return "\""
        <++
```

```
        do
        string "\""
        return pfail "\" in toklit should be double"
        <++ do
        a <- satisfy (\x -> isPrint x && x /= '\"' )
        return [a]

        getMoreChar :: String -> Parser String
        getMoreChar s = do
        x <- getOneChar
        getMoreChar (s ++ x)
        +++ do return s


        -- Name: Any sequence of (Unicode) letters, digits, and
            underscores, starting with a letter.
        -- There are no reserved names.
        pName :: Parser Simple
        pName = do
        s <- pNameString
        skipSpaceAndComment
        return $ SNTerm s

        pNameString :: Parser String
        pNameString = do
        a <- satisfy isLetter
        as <- munch (\x -> (x =='_' || isLetter x || isDigit x) &&
            isPrint x)
        return $ a:as

        skipSpaceAndComment :: Parser()
        skipSpaceAndComment = do
        skipSpaces
        s <- look
        if isComment s then
        do
        munch1 (/= '\n')
        skipSpaceAndComment
        else return ()
        -- return ()

        isComment :: String -> Bool
        isComment s = length s >= 2 && head s == '-' && head (tail s)
            == '-'
```

7.1.2 BlackBox.hs

```
-- Sample black-box test suite. Feel free to adapt, or start from
    scratch.

-- Do NOT import from your ModImpl files here. These tests should
    work with
-- any implementation of the APpy APIs. Put any white-box tests in
-- suite1/WhiteBox.hs.
import Definitions
import Parser
import Transformer

import Test.Tasty
import Test.Tasty.HUnit

main :: IO ()
main = defaultMain $ localOption (mkTimeout 1000000) tests

tests = testGroup "Smoke tests" [
testCase "Parser" $ parseSpec str @?= Right ("", eg),
testCase "Parser_testEBar1" $ parseSpec str2 @?= Right ("", result2),
testCase "Parser_testEBar2" $ parseSpec str3 @?= Right ("", result3),
testCase "Parser_testESeq1" $ parseSpec str4 @?= Right ("", result4),
testCase "Parser_testESeq2" $ parseSpec str5 @?= Right ("", result5),
testCase "Parser_testEOption" $ parseSpec str6 @?= Right ("",
    result6),
testCase "Parser_testEOption2" $ parseSpec str7 @?= Right ("",
    result7),
testCase "Parser_testEMany1" $ parseSpec str8 @?= Right ("", result8),
testCase "Parser_testEMany2" $ parseSpec str9 @?= Right ("", result9),
testCase "Parser_testEPred" $ parseSpec str10 @?= Right ("",
    result10),
testCase "Parser_testENot" $ parseSpec str11 @?= Right ("", result11),
testCase "Transformer.convert" $
convert eg @?= Right g] -- assumes that convert preserves input rule
    order
where
str = "---\n S ::= S \"a\" {_1+1} | \"b\" {0}.\n _ ::= {()}."
str2 = "---\n Stmts {:[Stmt]} ::= Stmt    {[_1]}\n| Stmt \";\" Stmts
    {_1 : _3}."
result2 = [(("Stmts",RPlain,Just (AUser "[Stmt]")),EBar (ESeq
    [ESimple (SNTerm "Stmt")] "[_1]") (ESeq [ESimple (SNTerm
    "Stmt"),ESimple (SLit ";"),ESimple (SNTerm "Stmts")] "_1 : _3"))]
str3 = "---\n Stmts {:[Stmt]} ::= Abcd    {[_1]}\n| Stmt \";\" Stmts
    {_1 : _3}."
result3 = [(("Stmts",RPlain,Just (AUser "[Stmt]")),EBar (ESeq
    [ESimple (SNTerm "Abcd")] "[_1]") (ESeq [ESimple (SNTerm
    "Stmt"),ESimple (SLit ";"),ESimple (SNTerm "Stmts")] "_1 : _3"))]
```

```
str4 = "Whitespace {:()} ::= WSElt* !WSElt {()}.\n WSElt {:()} ::=\n
    @{?isSpace} {()}\n| '#' Comment {()}.\n"
result4 = [(("Whitespace",RPlain,Just (AUser "()")),ESeq [EMany
    (ESimple (SNTerm "WSElt")),ENot (ESimple (SNTerm "WSElt"))]
    "()"),(("WSElt",RPlain,Just (AUser "()")),EBar (ESeq [EPred
    (ESimple SAnyChar) "isSpace"] "()") (ESeq [ESimple (SChar
    '#'),ESimple (SNTerm "Comment")] "()"))]

str5 = "Whitespace {:()} ::= XWSElt* !WSElt {()}.\n WSElt {:()} ::=\n
    @{?isSpace} {()}\n| '#' Comment {()}.\n"
result5 = [(("Whitespace",RPlain,Just (AUser "()")),ESeq [EMany
    (ESimple (SNTerm "XWSElt")),ENot (ESimple (SNTerm "WSElt"))]
    "()"),(("WSElt",RPlain,Just (AUser "()")),EBar (ESeq [EPred
    (ESimple SAnyChar) "isSpace"] "()") (ESeq [ESimple (SChar
    '#'),ESimple (SNTerm "Comment")] "()"))]

str6 = "---\nWhitespace {:()} ::= KU? University {_1 _2} | Shanghai
    {_1}."
result6 = [(("Whitespace",RPlain,Just (AUser "()")),EBar (ESeq
    [EOption (ESimple (SNTerm "KU")),ESimple (SNTerm "University")]
    "_1 _2") (ESeq [ESimple (SNTerm "Shanghai")] "_1"))]

str7 = "---\nWhitespace {:()} ::= ShanghaiTech? University {_1 _2} |
    Shanghai {_1}."
result7 = [(("Whitespace",RPlain,Just (AUser "()")),EBar (ESeq
    [EOption (ESimple (SNTerm "ShanghaiTech")),ESimple (SNTerm
    "University")] "_1 _2") (ESeq [ESimple (SNTerm "Shanghai")] "_1"))]

str8 = "---\nWhitespace {:()} ::= KU* University {_1 _2} | Shanghai
    {_1}."
result8 = [(("Whitespace",RPlain,Just (AUser "()")),EBar (ESeq [EMany
    (ESimple (SNTerm "KU")),ESimple (SNTerm "University")] "_1 _2")
    (ESeq [ESimple (SNTerm "Shanghai")] "_1"))]

str9 = "---\nWhitespace {:()} ::= KU* niversity {_1 _2}."
result9 = [(("Whitespace",RPlain,Just (AUser "()")),ESeq [EMany
    (ESimple (SNTerm "KU")),ESimple (SNTerm "niversity")] "_1 _2")]

str10 = "---\nNLZDigits {:String} ::= '0'                    {[_1]}|
    Digit{?/='0'} Digit* !Digit {_1:_2}."
result10 = [(("NLZDigits",RPlain,Just (AUser "String")),EBar (ESeq
    [ESimple (SChar '0')] "[_1]") (ESeq [EPred (ESimple (SNTerm
    "Digit")) "/='0'",EMany (ESimple (SNTerm "Digit")),ENot (ESimple
    (SNTerm "Digit"))] "_1:_2"))]

str11 = "---\nIdent1 {:String} ::=IdChar{?not.isDigit} IdChar*
    !IdChar{_1 : _2}."
result11 = [(("Ident1",RPlain,Just (AUser "String")),ESeq [EPred
    (ESimple (SNTerm "IdChar")) "not.isDigit",EMany (ESimple (SNTerm
```

```
    "IdChar")),ENot (ESimple (SNTerm "IdChar"))] "_1 : _2")]

eg = [(("S", RPlain, Nothing),
EBar (ESeq [ESimple (SNTerm "S"), ESimple (SLit "a")] "_1+1")
(ESeq [ESimple (SLit "b")] "0")),
(("_", RSep, Nothing), ESeq [] ("()"))]
g = [(("S", RPlain, Nothing),
[([SNTerm "S", SLit "a"], AUser "_1+1"),
([SLit "b"], AUser "0")]),
(("_", RSep, Nothing), [([], AUser "()")])]
```

7.2 Erlang

7.2.1 erlst.erl

```
-module(erlst).
-behaviour(gen_server).

% 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,
observe_state/1,
init/1,
handle_call/3,
accept_offer/4,
handle_cast/2]).

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

-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()).


%% account_map : key account_id(), value holdings()
-record(account_data, {current_account_num::integer(),
    account_map}).
%% offers_map : key offer_id(), value {account_id(), offer()}
-record(offer_data, {current_offer_num::integer(), offers_map}
    ).
%% trader_map : key trader_id(), value {trader_pid, Strategy,
    AccountId }
-record(trader_data, {current_trader_num::integer(),
    traders_map}).
%% supervisor {SupId, }
-record(se_server_data, {accounts, offers, traders,
    supervisor, executed_trades_num}).


-spec launch() -> {ok, stock_exchange()} | {error, term()}.
launch() ->
gen_server:start_link(?MODULE, [] ,[]).

%% for debug only, return State in gen_server for test use.
observe_state(S) ->
gen_server:call(S, {observe_state_offers}).

-spec shutdown(S :: stock_exchange()) -> non_neg_integer().
shutdown(S) ->
ExecutedNumber = gen_server:call(S, {shutdown}),
gen_server:stop(S),
ExecutedNumber.

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

-spec account_balance(Acct :: account_id()) -> holdings().
account_balance(Acct) ->
{_, ServerPid} = Acct,
%% todo : if ServerPid doesn't exist, crash; So should add
    try-catch here.
gen_server:call(ServerPid, {account_balance, Acct}).
```

```
-spec make_offer(Acct :: account_id(), Terms :: offer()) ->
    {ok, offer_id()} | {error, term()}.
make_offer(Acct, Offer) ->
{_, ServerPid} = Acct,
gen_server:call(ServerPid, {make_offer, Acct, Offer}).

-spec rescind_offer(Acct :: account_id(), Offer :: offer_id())
    -> ok.
rescind_offer(Acct, OfferId) ->
{_, ServerPid} = Acct,
gen_server:cast(ServerPid, {rescind_offer, Acct, OfferId}).

-spec add_trader(Acct :: account_id(), Strategy ::
    trader_strategy()) -> trader_id().
add_trader(AccountId, Strategy) ->
{_, Server} = AccountId,
gen_server:call(Server, {add_trader, Strategy, AccountId}).

-spec remove_trader(Acct :: account_id(), Trader ::
    trader_id()) -> ok.
remove_trader(AccountId, TraderId) ->
{_, Server} = AccountId,
gen_server:cast(Server, {remove_trader, TraderId}).

accept_offer(S, OfferElem, AccountId, TraderId) ->
gen_server:call(S, {accept_offer, OfferElem, AccountId,
    TraderId}).

init(_) ->
State = #se_server_data{
    accounts=#account_data{current_account_num = 0,
        account_map = maps:new()},
    offers=#offer_data{ current_offer_num=0, offers_map =
        maps:new()},
    traders = #trader_data{current_trader_num = 0, traders_map
        = maps:new()},
    supervisor = sup:start_link(),
    executed_trades_num = 0
},
{ok, State}.

check_accepted_offer_valid(State, OfferId, Offer,
    BuyerAccountId, SellerAccountId, TraderId) ->
case check_offer_exist(State, OfferId) of
false -> offer_do_not_exist;
true ->
case check_seller_has_stock(State, SellerAccountId, Offer) of
false -> seller_do_not_have_stock;
true ->
```

```
case check_buyer_has_enough_money(State, Offer,
    BuyerAccountId) of
false -> buyer_do_not_have_enough_money;
true ->
case check_valid_trader(State, TraderId) of
false -> offer_do_not_exist;
true -> ok
end
end
end
end.

check_buyer_has_enough_money(State, Offer, BuyerAccountId) ->
{ _, Price} = Offer,
case get_account_by_id(State, BuyerAccountId) of
account_do_not_exist ->
false;
{CurMoney, _} ->
if
CurMoney >= Price -> true;
true -> false
end
end.

check_valid_trader(State, TraderId) ->
case get_trader_by_id(State, TraderId) of
trader_do_not_exist ->
%%     io:format("[server process] trader do not exist ~n"),
false;
_ ->
%%     io:format("[server process] trader exist ~n"),
true
end.

check_offer_exist(State, OfferId) ->
case get_offer_by_id(State, OfferId) of
offer_do_not_exist -> false;
_ -> true
end .

check_seller_has_stock(State, SellerAccountId, Offer) ->
{StockName, _} = Offer,
case get_account_by_id(State, SellerAccountId) of
account_do_not_exist -> false;
{_, StockList} ->
case lists:keyfind(StockName, 1, StockList) of
false -> false;
{_, Amount} ->
if
```

```
        Amount > 0 -> true;
        true -> false
        end
        end
        end.

        execute_trade(State, SellerAccountId, BuyerAccountId, Offer,
            OfferId) ->
        NewState = delete_offer_by_id(State, OfferId),
        {StockName, Price} = Offer,
        NewState2 = update_seller_holding(NewState, SellerAccountId,
            StockName, Price),
        NewState3 = update_executed_trades_num(NewState2),
        update_buyer_holding(NewState3, BuyerAccountId, StockName,
            Price).

        %% for observe_state_offers request
        handle_call({observe_state_offers}, _From, State) ->
        OfferList =
            maps:to_list((State#se_server_data.offers)#offer_data.offers_map),
        {reply, OfferList, State};

        handle_call({accept_offer, OfferElem, AccountId, TraderId},
            _From, State) ->
        %%  io:format("[server process] get accept_offer request ~n"),
        {OfferId, {SellerAccountId, Offer}} = OfferElem,
        BuyerAccountId = AccountId,
        case check_accepted_offer_valid(State, OfferId, Offer,
            BuyerAccountId, SellerAccountId, TraderId) of
        ok ->
        %%     io:format("[server process] before make deal: ~n~p ~n",
            [State]),
        NewState = execute_trade(State, SellerAccountId,
            BuyerAccountId, Offer, OfferId),
        %%     io:format("[server process] after make deal: ~n~p ~n",
            [NewState]),
        {reply, delete_trader_offer, NewState};
        offer_do_not_exist ->
        %%     io:format("[server process] offer do not exist ~n"),
        {reply, delete_trader_offer, State};
        _ ->
        %%     io:format("[server process] check_accepted_offer_valid
            result: Other ~n"),
        {reply, keep_trader_offer, State}
        end;

        %% for open_account request
        handle_call({open_account , Holdings}, _From, State) ->
        {AccountId, NewState} = add_account(State, self(), Holdings),
```

```
%%  io:format("[server process] open_account ~p
    ~n",[AccountId]),
{reply, AccountId, NewState};

%% for shutdown request
handle_call({shutdown}, _From, State) ->
shutdown_all_traders(State),
NewState1 = set_accounts_map(State, maps:new()),
NewState2 = set_traders_map(NewState1, maps:new()),
ExecutedTradesNum = get_executed_trades_num(NewState2),
{reply, ExecutedTradesNum, NewState2};

%% for account_balance request
handle_call({account_balance , AccountId}, _From, State) ->
%%  io:format("accountId in handle_call : ~p ~n", [AccountId]),
Holdings = get_account_by_id(State, AccountId),
%% todo: what if no such account?
{reply, Holdings, State};

%% for make_offer request
handle_call({make_offer , Acct, Offer}, _From, State) ->
case check_valid_offer(Offer) of
true ->
{OfferId, NewState} = add_offer(self(), State, Acct, Offer),
%%     io:format("[server process] add new offer Offer: ~p
    ~n",[Offer]),
notify_all_trader_new_offer(NewState, OfferId, Offer, Acct),
{reply, {ok, OfferId}, NewState};
false ->
{reply, {error, invalid_offer}, State}
end;

handle_call({add_trader, Strategy, AccountId}, _From, State) ->
TraderId = get_current_trader_num(State) + 1,
%%  {_, TraderPid} = trader:start_trader(Strategy, self(),
    get_offers_map(State) , AccountId, TraderId),
Sup = get_supervisor(State),
ChildSpec = [Strategy, self(), get_offers_map(State) ,
    AccountId, TraderId],
Result = supervisor:start_child(Sup, ChildSpec),
{_, TraderPid} = Result,
NewState = set_current_trader_num(State, TraderId),
TraderValue = {TraderPid, Strategy, AccountId},
TraderMap = get_traders_map(NewState),
NewTradeMap = maps:put(TraderId, TraderValue, TraderMap),
NewState2 = set_traders_map(NewState, NewTradeMap),
%%  io:format("[server process] add trader successfully ~n"),
{reply, TraderId, NewState2}.
```

```
%% for open_account request
handle_cast({remove_trader, TraderId}, State) ->
case get_trader_by_id(State, TraderId) of
trader_do_not_exist ->
%%     io:format("[server process] remove_trader but
    trader_do_not_exist ~n"),
{noreply, State};
{TraderPid, _, _} ->
NewState = delete_trader_by_id(State, TraderId),
supervisor:terminate_child(get_supervisor(State), TraderPid),
%%     io:format("[server process] remove trader, newstate:
    ~n~p~n",[NewState]),
{noreply, NewState}
end;

%% for rescind_offer request
handle_cast({rescind_offer, _Acct, OfferId}, State) ->
%%  todo when rescind offer, should notify child process to
    stop running strategy on this offer.
NewState = delete_offer(State, OfferId),
shutdown_all_traders(NewState),
OldTraderMaps = get_traders_map(NewState),
OldTraderList = maps:to_list(OldTraderMaps),
NewState2 = set_traders_map(NewState, maps:new()),
NewState3 = restart_all_traders(NewState2, OldTraderList),
{noreply, NewState3}.

restart_trader({Strategy, AccountId, TraderId}, State) ->
Sup = get_supervisor(State),
ChildSpec = [Strategy, self(), get_offers_map(State) ,
    AccountId, TraderId],
Result = supervisor:start_child(Sup, ChildSpec),
%% io:format("[server process] start_child result:
    ~p~n",[Result]),
{_, TraderPid} = Result,
TraderValue = {TraderPid, Strategy, AccountId},
TraderMap = get_traders_map(State),
NewTradeMap = maps:put(TraderId, TraderValue, TraderMap),
NewState2 = set_traders_map(State, NewTradeMap),
%% io:format("[server process] add trader successfully ~n"),
{TraderId, NewState2}.

restart_all_traders(State, TraderList) ->
case TraderList of
[X|XS] ->
{TraderId, {_, Strategy, AccountId}} = X,
{_, NewState} = restart_trader({Strategy, AccountId,
    TraderId}, State),
restart_all_traders(NewState, XS);
```

```
        [] ->
        State
        end.

        shutdown_trader_list(Sup, TradersList) ->
        case TradersList of
        [X|XS] ->
        {_, {TraderPid, _, _}} = X,
        supervisor:terminate_child(Sup, TraderPid),
        shutdown_trader_list(Sup, XS);
        [] ->
        nothing
        end.

        shutdown_all_traders(State) ->
        Sup = get_supervisor(State),
        TradesMap = get_traders_map(State),
        TradersList = maps:to_list(TradesMap),
        shutdown_trader_list(Sup, TradersList).

        %%key offer_id(), value {account_id(), offer()}
        %%{trader_pid, Strategy, AccountId }
        notify_all_trader_new_offer(State, TraderList, OfferId, Offer,
            AccountId)->
        case TraderList of
        [X | XS] ->
        {_, TraderValue} = X,
        {TraderPid , _, _} = TraderValue,
        OfferElem = {OfferId, {AccountId, Offer}},
        trader:new_offer(TraderPid, OfferElem),
        notify_all_trader_new_offer(State, XS, OfferId, Offer,
            AccountId);
        [] ->
        nothing
        end.

        notify_all_trader_new_offer(State, OfferId, Offer, AccountId)
            ->
        TradersMap = get_traders_map(State),
        TradersList = maps:to_list(TradersMap),
        notify_all_trader_new_offer(State, TradersList, OfferId,
            Offer, AccountId).


        check_valid_offer(Offer) ->
        {StockName, Price} = Offer,
        case is_atom(StockName) of
        true ->
        case is_integer(Price) of
```

```
        true ->
        case Price >= 0 of
        true -> true;
        false -> false
        end;
        false -> false
        end;
        false -> false
        end.

        %% api for get data quickly from State
        get_accounts_data(State) ->
        State#se_server_data.accounts.

        get_offers_data(State) ->
        State#se_server_data.offers.

        get_traders_data(State)->
        State#se_server_data.traders.

        get_accounts_map(State) ->
        (get_accounts_data(State))#account_data.account_map.

        set_accounts_map(State, NewAccountMap) ->
        NewAccountData =
            (get_accounts_data(State))#account_data{account_map =
            NewAccountMap},
        State#se_server_data{accounts = NewAccountData}.

        get_offers_map(State) ->
        (get_offers_data(State))#offer_data.offers_map.

        %%set_offers_map(State, NewOffersMap) ->
        %%  NewOffersData =
            (get_offers_data(State))#offer_data{offers_map =
            NewOffersMap},
        %%  State#se_server_data{offers = NewOffersData}.

        get_traders_map(State) ->
        (get_traders_data(State))#trader_data.traders_map.

        set_traders_map(State, NewTradersMap) ->
        NewTradersData = (get_traders_data(State))#trader_data{
            traders_map = NewTradersMap},
        State#se_server_data{traders = NewTradersData}.

        get_current_trader_num(State) ->
        (get_traders_data(State))#trader_data.current_trader_num.
```

```
set_current_trader_num(State, Num) ->
NewTradersData = (get_traders_data(State))#trader_data{
    current_trader_num = Num},
State#se_server_data{traders = NewTradersData}.

get_current_account_num(State) ->
(get_accounts_data(State))#account_data.current_account_num.

set_current_account_num(State, Num) ->
NewAccountsData = (get_accounts_data(State))#account_data{
    current_account_num = Num},
State#se_server_data{accounts = NewAccountsData}.

%%get_current_offer_num(State) ->
%%  (get_offers_data(State))#offer_data.current_offer_num.

get_offer_by_id(State, OfferId) ->
OfferMap = get_offers_map(State),
maps:get(OfferId, OfferMap, offer_do_not_exist).

get_account_by_id(State, AccountId) ->
AccountMap = get_accounts_map(State),
maps:get(AccountId, AccountMap, account_do_not_exist).

get_trader_by_id(State, TraderId) ->
TraderMap = get_traders_map(State),
maps:get(TraderId, TraderMap, trader_do_not_exist).

delete_offer_by_id(State, OfferId) ->
OfferMap = get_offers_map(State),
NewOffersMap = maps:remove(OfferId, OfferMap),
NewOffersData = (get_offers_data(State))#offer_data{offers_map
    = NewOffersMap},
State#se_server_data{offers = NewOffersData}.

delete_trader_by_id(State, TraderId) ->
TraderMap = get_traders_map(State),
NewTraderMap = maps:remove(TraderId, TraderMap),
NewTraderData =
    (get_traders_data(State))#trader_data{traders_map =
    NewTraderMap},
State#se_server_data{traders = NewTraderData}.

get_supervisor(State) ->
State#se_server_data.supervisor.

set_account_by_id(State, SellerAccountId, AccountValue) ->
AccountMap = get_accounts_map(State),
```

```
    NewAccountMap = maps:update(SellerAccountId ,AccountValue,
        AccountMap),
    set_accounts_map(State, NewAccountMap).

    get_executed_trades_num(State)->
    State#se_server_data.executed_trades_num.

    update_executed_trades_num(State) ->
    NewNum = get_executed_trades_num(State) + 1,
    State#se_server_data{executed_trades_num = NewNum}.

    update_seller_holding(State, SellerAccountId, StockName,
        Price) ->
    Holding = get_account_by_id(State, SellerAccountId),
    {CurMoney, StockList} = Holding,
    {_, Amount} = lists:keyfind(StockName, 1, StockList),
    NewStockList = case Amount - 1 of
    0 -> lists:delete({StockName, Amount} ,StockList);
    _ -> lists:delete({StockName, Amount} ,StockList) ++
        [{StockName, Amount - 1}]
    end,
    NewHolding = {CurMoney + Price, NewStockList},
    set_account_by_id(State, SellerAccountId, NewHolding).

    update_buyer_holding(State, BuyerAccountId, StockName, Price)
        ->
    Holding = get_account_by_id(State, BuyerAccountId),
    {CurMoney, StockList} = Holding,
    case lists:keyfind(StockName, 1, StockList) of
    false ->
    NewStockList = StockList ++ [{StockName, 1}],
    NewHolding = {CurMoney - Price, NewStockList},
    set_account_by_id(State, BuyerAccountId, NewHolding);
    {_, Amount} ->
    NewStockList = lists:delete({StockName, Amount} ,StockList) ++
        [{StockName, Amount + 1}],
    NewHolding = {CurMoney - Price, NewStockList},
    set_account_by_id(State, BuyerAccountId, NewHolding)
    end.

    %% offer utils function
    add_offer(Server, State, Acct, Offer) ->
    Offers = State#se_server_data.offers,
    Id = Offers#offer_data.current_offer_num + 1,
    NewOfferMap = maps:put({Id, Server}, {Acct, Offer} ,
        Offers#offer_data.offers_map),
    NewOffers = Offers#offer_data{current_offer_num = Id,
        offers_map = NewOfferMap},
    NewState = State#se_server_data{offers = NewOffers},
```

```
        {{Id, Server}, NewState}.

        delete_offer(State, OfferId) ->
        OffersMap =
            (State#se_server_data.offers)#offer_data.offers_map,
        NewOffersMap = maps:remove(OfferId, OffersMap),
        State#se_server_data{offers
            =(State#se_server_data.offers)#offer_data{offers_map =
            NewOffersMap}}.

        %% account utils functions
        add_account(State, Server, Holdings) ->
        Id = get_current_account_num(State) + 1,
        NewAccountMap = maps:put({Id, Server}, Holdings ,
            get_accounts_map(State)),
        NewState1 = set_accounts_map(State, NewAccountMap),
        NewState2 = set_current_account_num(NewState1, Id),
        {{Id, Server}, NewState2}.
```

7.2.2 sup.erl

```
-module(sup).

-export([start_link/0]).

-behaviour(supervisor).
-export([init/1]).

-define(SERVER, ?MODULE).

start_link() ->
{ok, Sup} = supervisor:start_link( ?MODULE, []),
Sup.

%% @private
init(_Args) ->
SupFlags = #{strategy => simple_one_for_one, intensity => 1, period
    => 5},
ChildSpecs = [#{id => trader,
    start => {trader, start_trader, []},
    restart => permanent,
    shutdown => brutal_kill,
    type => worker,
    modules => [trader]}],

{ok, {SupFlags, ChildSpecs}}.
```

7.2.3 trader.erl

```
-module(trader).
-author("liyk1").
-behavior(gen_server).

%% API
-export([init/1, handle_call/3, handle_cast/2]).
-export([start_trader/5, new_offer/2]).


start_trader(Strategy, Sup, CurOffers, AccountId, TraderId) ->
{ok, Pid} = gen_server:start_link(?MODULE, {Sup, CurOffers, Strategy,
    AccountId, TraderId}, []),
try_accept_all_offers(Pid),
{ok, Pid}.

%% only pass AccountId to trader, the trader doesn't care the
    holdings in the account
init({Sup, CurOffers, Strategy, AccountId, TraderId}) ->
MyMap = maps:new(),
MyMap1 = maps:put("Sup", Sup, MyMap),
MyMap2 = maps:put("CurOffers", CurOffers, MyMap1),
MyMap3 = maps:put("Strategy", Strategy, MyMap2),
MyMap4 = maps:put("AccountId", AccountId, MyMap3),
MyMap5 = maps:put("TraderId", TraderId, MyMap4),
{ok, MyMap5}.

try_accept_all_offers(Trader) ->
gen_server:cast(Trader, {try_accept_all_offers}).

new_offer(Trader, OfferElem) ->
gen_server:cast(Trader, {new_offer, OfferElem}).

handle_call(_Request, _From, _State) ->
erlang:error(not_implemented).

handle_cast({try_accept_all_offers}, State) ->
Strategy = get_strategy_from_state(State),
CurOffers = get_cur_offers_from_state(State),
CurOffersList = maps:to_list(CurOffers),
NewState = implement_strategy_to_offers(State, CurOffersList,
    Strategy),
{noreply, NewState};

handle_cast({new_offer, OfferElem}, State) ->
```

```
%% io:format("[trader process ~p] new offer from server: ~p ~n",
    [self(), OfferElem]),
Strategy = get_strategy_from_state(State),
CurOffersMap = get_cur_offers_from_state(State),
{Key, Value} = OfferElem,
NewCurOffersMap = maps:put(Key , Value, CurOffersMap),
NewState = set_cur_offers(State, NewCurOffersMap),
CurOffersList = maps:to_list(NewCurOffersMap),
NewState2 = implement_strategy_to_offers(NewState, CurOffersList,
    Strategy),
{noreply, NewState2}.

accept_offer(State , OfferElem, OfferId) ->
Sup = get_sup_from_state(State),
AccountId = get_accounts_id_state(State),
TraderId = get_trader_id(State),
case erlst:accept_offer(Sup, OfferElem, AccountId, TraderId) of
keep_trader_offer ->
%%     io:format("[trader process ~p] get response from server:
    keep_trader_offer ~n", [self()]),
State;
delete_trader_offer->
%%     io:format("[trader process ~p] get response from server:
    delete_trader_offer ~n", [self()]),
delete_offer_by_id(State, OfferId)
end.

implement_strategy_to_offers(State, CurOffersList, Strategy) ->
case CurOffersList of
[X |XS] ->
{Key, Value} = X,
{_, Offer} = Value,
Result = try Strategy(Offer) of
accept ->
accept;
reject ->
reject
catch
_:_ ->
reject
end,
case Result of
accept ->
%%         io:format("[trader process ~p] accept one offer: ~p
    ~n",[self(), Offer]),
NewState = accept_offer(State, X, Key),
implement_strategy_to_offers(NewState, XS, Strategy);
reject ->
```

```
%%         io:format("[trader process ~p] reject one offer:~p
    ~n",[self(), Offer]),
NewState = delete_offer_by_id(State, Key),
implement_strategy_to_offers(NewState, XS, Strategy)
end;
[] ->
%%     io:format("[trader process ~p] implement_strategy_to_offers
    done ~n",[self()]),
State
end.

delete_offer_by_id(State, OfferId) ->
CurOffersMap = get_cur_offers_from_state(State),
NewCurOffersMap = maps:remove(OfferId, CurOffersMap),
set_cur_offers(State, NewCurOffersMap).

%% State util function
get_sup_from_state(State) ->
maps:get("Sup", State).

get_cur_offers_from_state(State) ->
maps:get("CurOffers", State).

set_cur_offers(State, CurOffersMap) ->
maps:update("CurOffers", CurOffersMap, State).

get_strategy_from_state(State) ->
maps:get("Strategy", State).

get_accounts_id_state(State) ->
maps:get("AccountId", State).

get_trader_id(State) ->
maps:get("TraderId", State).
```

7.2.4 test_eunit_erlst.erl

```
    -module(test_eunit_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_'.
% But you MUST have a module (this file) called test_erlst.

test_all() ->
%%  all this functions starting with 'test_' are for testing
    basic function in Q2.1
test_launch(),
test_open_account(),
test_account_balance1(),
test_make_offer1(),
test_make_offer2(),
test_make_offer3(),
test_rescind_offer(),
test_rescind_offer2(),
test_rescind_offer3(),
test_all_add_trader(),
test_remove_trader(),
test_shutdown(),
test_shutdown2(),
test_exception_strategy(),
ok.

test_everything() ->
test_all().

test_launch() ->
{ok, A} = erlst:launch(),
{ok, B} = erlst:launch(),
erlst:open_account(A, {10, [{a, 1}, {b, 2}]}),
erlst:open_account(B, {20, []}),
erlst:open_account(A, {10, [{a, 10}, {b, 20}]}),
io:format("test_lanunch1 ok ~n").

test_open_account() ->
{ok, A} = erlst:launch(),
{ok, B} = erlst:launch(),
erlst:open_account(A, {10, [{a, 1}, {b, 2}]}),
erlst:open_account(B, {20, []}),
erlst:open_account(A, {10, [{a, 10}, {b, 2320}]}),
erlst:open_account(A, {10, [{a, 1023}, {"c", 230}]}),
erlst:open_account(A, {10, [{a, 140}, {"d", 230}]}),
erlst:open_account(A, {10, [{"f", 130}, {b, 2320}]}),
io:format("test_open_account ok ~n").

test_account_balance1() ->
{ok, A} = erlst:launch(),
InputHolding1 = {100, [{a, 1}, {b, 2}]},
InputHolding2 = {1012332, [{a, 10}, {b, 2320}]},
Account1 = erlst:open_account(A, InputHolding1),
```

```
    Account2 = erlst:open_account(A, InputHolding2),
    Holding1 = erlst:account_balance(Account1),
    Holding2 = erlst:account_balance(Account2),
    ?assertMatch(Holding1, InputHolding1),
    ?assertMatch(Holding2, InputHolding2),
    io:format("test_account_balance1 ok ~n").

    test_make_offer1() ->
    {ok, A} = erlst:launch(),
    InputHolding1 = {100, [{a, 10}, {b, 25}]},
    Account1 = erlst:open_account(A, InputHolding1),
    {ok, {OfferId, Server}} = erlst:make_offer(Account1, {a, 12}),
    {ok, {OfferId2, Server2}} = erlst:make_offer(Account1, {a, 18}),
    ?assertMatch(OfferId, 1),
    ?assertMatch(Server, A),
    ?assertMatch(OfferId2, 2),
    ?assertMatch(Server2, A),
    io:format("test_make_offer1 ok ~n").

    test_make_offer2() ->
    {ok, A} = erlst:launch(),
    {ok, B} = erlst:launch(),
    InputHolding1 = {100, [{a, 10}, {b, 10}]},
    InputHolding2 = {200, [{a, 2}, {b, 20}]},
    Account1 = erlst:open_account(A, InputHolding1),
    Account2 = erlst:open_account(B, InputHolding2),
    {ok, {OfferId, Server}} = erlst:make_offer(Account1, {a, 12}),
    {ok, {OfferId2, Server2}} = erlst:make_offer(Account1, {a, 18}),
    ?assertMatch(OfferId, 1),
    ?assertMatch(Server, A),
    ?assertMatch(OfferId2, 2),
    ?assertMatch(Server2, A),

    {ok, {OfferId3, Server3}} = erlst:make_offer(Account2, {a, 100}),
    {ok, {OfferId4, Server4}} = erlst:make_offer(Account2, {a, 18}),
    ?assertMatch(OfferId3, 1),
    ?assertMatch(Server3, B),
    ?assertMatch(OfferId4, 2),
    ?assertMatch(Server4, B),

    io:format("test_make_offer2 ok ~n").

    test_make_offer3() ->
    {ok, A} = erlst:launch(),
    {ok, B} = erlst:launch(),
    InputHolding1 = {100, [{a, 1}, {b, 10}]},
    InputHolding2 = {200, [{a, 20}, {b, 20}]},
    Account1 = erlst:open_account(A, InputHolding1),
    Account2 = erlst:open_account(B, InputHolding2),
```

```
{ok, {OfferId, _}} = erlst:make_offer(Account1, {a, 12}),
Result = erlst:make_offer(Account1, {a, 18}),
io:format("Result : ~p ~n", [Result]),
?assertMatch(OfferId, 1),

{ok, {OfferId3, Server3}} = erlst:make_offer(Account2, {a, 100}),
{ok, {OfferId4, Server4}} = erlst:make_offer(Account2, {a, 18}),
?assertMatch(OfferId3, 1),
?assertMatch(Server3, B),
?assertMatch(OfferId4, 2),
?assertMatch(Server4, B),

io:format("test_make_offer3 ok ~n").

test_rescind_offer() ->
{ok, A} = erlst:launch(),
InputHolding1 = {100, [{a, 10}, {b, 10}]},
Account1 = erlst:open_account(A, InputHolding1),
{ok, {OfferId, Server}} = erlst:make_offer(Account1, {a, 12}),
{ok, {OfferId2, Server2}} = erlst:make_offer(Account1, {a, 18}),
?assertMatch(OfferId, 1),
?assertMatch(Server, A),
?assertMatch(OfferId2, 2),
?assertMatch(Server2, A),
erlst:rescind_offer(Account1, {OfferId, Server}),
io:format("test_rescind_offer ok ~n").

test_rescind_offer2() ->
{ok, A} = erlst:launch(),
{ok, B} = erlst:launch(),
InputHolding1 = {100, [{a, 10}, {b, 10}]},
InputHolding2 = {200, [{a, 2}, {b, 20}]},
Account1 = erlst:open_account(A, InputHolding1),
Account2 = erlst:open_account(B, InputHolding2),
{ok, {OfferId, Server}} = erlst:make_offer(Account1, {a, 12}),
{ok, {OfferId2, Server2}} = erlst:make_offer(Account1, {a, 18}),
?assertMatch(OfferId, 1),
?assertMatch(Server, A),
?assertMatch(OfferId2, 2),
?assertMatch(Server2, A),
erlst:rescind_offer(Account1, {OfferId, Server}),

OfferMap = maps:new(),
OfferMap1 = maps:put({2, A}, {{1,A},{a,18}}, OfferMap),
RightState = maps:to_list(OfferMap1),
State = erlst:observe_state(A),
?assertMatch(State, RightState),
```

```
{ok, {OfferId3, Server3}} = erlst:make_offer(Account2, {a, 100}),
{ok, {OfferId4, Server4}} = erlst:make_offer(Account2, {a, 18}),
?assertMatch(OfferId3, 1),
?assertMatch(Server3, B),
?assertMatch(OfferId4, 2),
?assertMatch(Server4, B),
io:format("test_rescind_offer2 ok ~n").

test_rescind_offer3() ->
{ok, A} = erlst:launch(),
InputHolding1 = {100, [{a, 10}]},
Account1 = erlst:open_account(A, InputHolding1),
{ok, OfferId1} = erlst:make_offer(Account1, {a, 1}),
erlst:make_offer(Account1, {a, 1}),
erlst:add_trader(Account1, fun({_StockName, Price}) ->
    timer:sleep(100), if Price < 2 -> accept; true -> reject end
    end),
erlst:rescind_offer(Account1, OfferId1),
io:format("[test process] test_rescind_offer3 ok ~n").

test_all_add_trader() ->
test_add_trader1(),
test_add_trader2(),
test_add_trader4(),
test_add_trader3().

%% seller and buyer are the same account
test_add_trader1() ->
{ok, A} = erlst:launch(),
InputHolding1 = {100, [{a, 10}, {b, 25}]},
Account1 = erlst:open_account(A, InputHolding1),
{ok, {OfferId, Server}} = erlst:make_offer(Account1, {a, 12}),
{ok, {OfferId2, Server2}} = erlst:make_offer(Account1, {a, 18}),
?assertMatch(OfferId, 1),
?assertMatch(Server, A),
?assertMatch(OfferId2, 2),
?assertMatch(Server2, A),
erlst:add_trader(Account1, fun(_Offer) -> accept end),
io:format("[test process] test_add_trader1 ok ~n").

%% In my design, the buyer and seller can be same account, so the
    behaviour of this
%% test is not deterministic, but we can make sure the final sum
    of money account1 and
%% account2 have is 100, and the sum of amount of stock a they
    have is 110.
test_add_trader2() ->
{ok, A} = erlst:launch(),
InputHolding1 = {100, [{a, 10}]},
```

```
InputHolding2 = {0, [{a, 100}]},
Account1 = erlst:open_account(A, InputHolding1),
Account2 = erlst:open_account(A, InputHolding2),
erlst:make_offer(Account1, {a, 3}),
erlst:make_offer(Account1, {a, 2}),
erlst:make_offer(Account2, {a, 1}),
erlst:make_offer(Account2, {a ,1}),
erlst:make_offer(Account2, {a ,1}),
erlst:make_offer(Account2, {a ,1}),
erlst:make_offer(Account2, {a ,1}),
erlst:add_trader(Account1, fun({_StockName, Price}) -> if Price <
    2 -> accept; true -> reject end end),
erlst:add_trader(Account2, fun(_Offer) -> accept end),
io:format("[test process] test_add_trader2 ok ~n").

strategy1({StockName, _Price}) ->
if
(StockName==b) -> accept;
true -> reject
end.

test_add_trader3() ->
{ok, A} = erlst:launch(),
InputHolding1 = {100, [{a, 10}]},
InputHolding2 = {0, [{b, 100}]},
Account1 = erlst:open_account(A, InputHolding1),
Account2 = erlst:open_account(A, InputHolding2),
erlst:make_offer(Account1, {a, 2}),
erlst:add_trader(Account1, fun({StockName, Price}) ->
    strategy1({StockName, Price}) end),
erlst:add_trader(Account2, fun(_Offer) -> accept end),
erlst:make_offer(Account2, {b ,10}),
io:format("[test process] test_add_trader3 ok ~n").

test_add_trader4() ->
{ok, A} = erlst:launch(),
InputHolding1 = {12, [{a, 1},{b, 1}, {c, 2}]},
InputHolding2 = {0, [{b, 1}]},
Account1 = erlst:open_account(A, InputHolding1),
Account2 = erlst:open_account(A, InputHolding2),
erlst:make_offer(Account1, {a, 2}),
erlst:add_trader(Account1, fun({StockName, Price}) ->
    strategy1({StockName, Price}) end),
erlst:add_trader(Account2, fun(_Offer) -> accept end),
erlst:make_offer(Account2, {b ,10}),
erlst:make_offer(Account1, {c, 6}),
io:format("[test process] test_add_trader3 ok ~n").

strategy_sleep_50ms({_StockName, _Price}) ->
```

```
    timer:sleep(50),
    accept.

    test_remove_trader() ->
    {ok, A} = erlst:launch(),
    InputHolding1 = {12, [{a, 1},{b, 1}, {c, 2}]},
    Account1 = erlst:open_account(A, InputHolding1),
    erlst:make_offer(Account1, {a, 2}),
    TraderId = erlst:add_trader(Account1, fun({StockName, Price}) ->
        strategy_sleep_50ms({StockName, Price}) end),
    erlst:make_offer(Account1, {c, 6}),
    erlst:remove_trader(Account1, TraderId),
    io:format("[test process] test_remove_trader ok ~n").

    test_shutdown() ->
    {ok, A} = erlst:launch(),
    InputHolding1 = {12, [{a, 1},{b, 1}, {c, 2}]},
    Account1 = erlst:open_account(A, InputHolding1),
    erlst:make_offer(Account1, {a, 2}),
    erlst:add_trader(Account1, fun({StockName, Price}) ->
        strategy_sleep_50ms({StockName, Price}) end),
    erlst:make_offer(Account1, {c, 6}),
    ExecutedNum = erlst:shutdown(A),
    ?assertMatch(ExecutedNum , 0),
    io:format("[test process] test_shutdown ok ~n").

    test_shutdown2() ->
    {ok, A} = erlst:launch(),
    InputHolding1 = {12, [{a, 1},{b, 1}, {c, 2}]},
    Account1 = erlst:open_account(A, InputHolding1),
    erlst:make_offer(Account1, {a, 2}),
    erlst:add_trader(Account1, fun({StockName, Price}) ->
        strategy_sleep_50ms({StockName, Price}) end),
    erlst:make_offer(Account1, {c, 6}),
    timer:sleep(120),
    ExecutedNum = erlst:shutdown(A),
    ?assertMatch(ExecutedNum , 2),
    io:format("[test process] test_shutdown2 ok ~n").

    strategy_exception({_StockName, _Price}) ->
    io:format("[test process] strategy_exception ~n"),
    X = 100/0,
    X.

    test_exception_strategy() ->
    {ok, A} = erlst:launch(),
    InputHolding1 = {12, [{a, 1},{b, 1}, {c, 2}]},
    Account1 = erlst:open_account(A, InputHolding1),
    erlst:make_offer(Account1, {a, 2}),
```

```
    erlst:add_trader(Account1, fun({StockName, Price}) ->
        strategy_exception({StockName, Price}) end),
    erlst:make_offer(Account1, {c, 6}),
    io:format("[test process] test_exception_strategy ok ~n").
```

7.2.5 test_erlst.erl

```
    -module(test_erlst).
    %%-include_lib("eunit/include/eunit.hrl").
    -include_lib("eqc/include/eqc.hrl").

    -export([test_all/0, test_everything/0,generate_stock_name/0 ]).
    -export([prop_value_preservation/0,generate_stock_exchange/0,
    test_strategy/0,generate_strategy/0, prop_total_trades/0,
    prop_value_preservation_collect_traderlist/0,
    prop_value_preservation_collect_offerlist/0]). % 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_'.
    % But you MUST have a module (this file) called test_erlst.

    test_all() ->
    %%  all this functions starting with 'test_' are for testing
        basic function in Q2.1
    eqc:quickcheck(prop_value_preservation()),
    eqc:quickcheck(prop_total_trades()),
    test_eunit_erlst:test_all(),
    ok.

    test_everything() ->
    test_all().

    %% TraderList [{InitialHolding, AccountId, TraderId}]
    %% offerList [{Holding, AccountId, {{StockName, Price} ,OfferId}}]
    prop_value_preservation() ->
    ?FORALL(
    {S, TraderList, OfferList, RemoveTrader, RemoveOffer},
    generate_offer_and_trader2(),
    begin
    TraderHolingMap =
        calculate_initial_holding_in_traderList(TraderList,
        maps:new()),
```

```
    AllHoldingMap = calculate_initial_holding_in_offerList(OfferList,
        TraderHolingMap),
    %% we have time-consuming strategy function, so we need to wait at
        least 1000.
    remove_trader_from_server(RemoveTrader, TraderList),
    %%     remove_offer_from_server(RemoveOffer, OfferList),
    timer:sleep(100),
    remove_all_trders(TraderList),
    timer:sleep(100),
    AccountIdList = collect_all_accountId(TraderList, OfferList),
    HoldingList = get_holding_list(AccountIdList),
    NewAllHoldingMap = calculate_initial_holdinglist(HoldingList,
        maps:new()),
    io:format("[test process] NewAllHoldingMap
        ~p~n",[NewAllHoldingMap]),
    io:format("[test process] AllHoldingMap ~p~n",[AllHoldingMap]),
    erlst:shutdown(S),
    io:format("------------------------------------------------------------~n"),
    NewAllHoldingMap == AllHoldingMap
    end
    ).

    remove_trader_from_server(B, TraderList) ->
    if B ->
    case TraderList of
    [X|_XS] ->
    {_, AccountId, TraderId} = X,
    io:format("[test process] remove trader ~n"),
    erlst:remove_trader(AccountId, TraderId);
    [] -> nothing
    end;
    true -> nothing
    end.


    remove_offer_from_server(B, OfferList) ->
    if B ->
    case OfferList of
    [X|_XS] ->
    {_, AccountId, {_, OfferId}} = X,
    io:format("[test process] remove offer ~n"),
    erlst:rescind_offer(AccountId, OfferId);
    [] -> nothing
    end;
    true -> nothing
    end.

    prop_value_preservation_collect_traderlist() ->
    ?FORALL(
```

```
{_S, TraderList, _OfferList,_,_},
generate_offer_and_trader2(),
collect(TraderList, true)
).

prop_value_preservation_collect_offerlist() ->
?FORALL(
{_S, _TraderList, OfferList,_,_},
generate_offer_and_trader2(),
collect(OfferList, true)
).

prop_total_trades() ->
?FORALL(
{S, TraderList, OfferList, RemoveTrader, RemoveOffer},
generate_offer_and_trader2(),
begin
MakeOfferNum = length(OfferList),
remove_trader_from_server(RemoveTrader, TraderList),
%%     remove_offer_from_server(RemoveOffer, OfferList),
timer:sleep(100),
FinalTradeNum = erlst:shutdown(S),
MakeOfferNum >= FinalTradeNum
end
).

%% the name of the stock can only be [a,b,c,d,e]
generate_stock_name() ->
eqc_gen:elements([a,b,c,d,e]).

generate_pos_int() ->
?LET(N, eqc_gen:nat(), case N of
0 -> 1;
X -> X
end ).

%% [{StockName, Amount}]
generate_stock_list() ->
eqc_gen:list({generate_stock_name(), generate_pos_int()}).

transfer_holding(StockList, HoldingMap) ->
case StockList of
[X|XS] ->
{StockName, Amount} = X,
CurrentAmount = maps:get(StockName, HoldingMap, 0),
NewMap = maps:put(StockName, Amount + CurrentAmount, HoldingMap),
transfer_holding(XS, NewMap);
[] -> HoldingMap
end.
```

```
%% holding:{CurMoney, [{StockName, Amount}]}
generate_holdings() ->
?LET({Money ,StockList}, {generate_pos_int() ,
    generate_stock_list()},
{Money, maps:to_list(transfer_holding(StockList, maps:new()))}).

generate_stock_exchange() ->
?LET({_, S}, ?LAZY(erlst:launch()), S).

%% return: {Holding, Account}
generate_account(S) ->
?LET(Holding, generate_holdings(), {Holding, erlst:open_account(S,
    Holding)}).

%% should limit the frequency of negative number.
%% most cases should be positive.
generate_stock_price() ->
eqc_gen:frequency([{1, eqc_gen:int()}, {9, eqc_gen:nat()}]).

%%return {{StockName, Price} ,OfferId}
generate_offer(Acct) ->
?LET({StockName, Price}, {generate_stock_name(),
    generate_stock_price()},
case erlst:make_offer(Acct, {StockName, Price}) of
{ok, OfferId} -> {{StockName, Price} ,OfferId};
{error, Reason} -> Reason
end ).

%% return {Holding, AccountId, {{StockName, Price} ,OfferId}}
generate_offer_with_server(S) ->
?LET({Holding, AccountId} , generate_account(S), {Holding,
    AccountId, generate_offer(AccountId)}).

%% return [{Holding, AccountId, {{StockName, Price} ,OfferId}}]
generate_offer_list_with_server(S) ->
eqc_gen:list(generate_offer_with_server(S)).


reliable_strategy() ->
eqc_gen:elements([
{call,test_erlst,mkstrategy,[buy_everything]},
{call,test_erlst,mkstrategy,[{buy_price_less_than, 20}]},
{call,test_erlst,mkstrategy,[{buy_only, a}]},
{call,test_erlst,mkstrategy,[{buy_only, b}]},
{call,test_erlst,mkstrategy,[{buy_only, c}]},
{call,test_erlst,mkstrategy,[{buy_only, d}]},
{call,test_erlst,mkstrategy,[{buy_only, e}]},
```

```
{call,test_erlst,mkstrategy,[{both, {buy_price_less_than, 8},
    {buy_only, a}}]},
{call,test_erlst,mkstrategy,[long_time_strategy_accept_everything]},
{call,test_erlst,mkstrategy,[long_time_strategy_reject_everything]}
]).

test_strategy() ->
?LAZY(eqc_symbolic:eval(reliable_strategy())).

generate_strategy() ->
?LET(Func, eqc_symbolic:eval(test_strategy()), Func).

mkstrategy_helper() ->
?LET(Symb, reliable_strategy(), case Symb of
{_,_,_, [Lists]} -> mkstrategy(Lists);
_ -> fun(_Offer) -> accept end
end ).

mkstrategy(Opr) ->
io:format("[test process] Opr ~p~n",[Opr]),
case Opr of
buy_everything ->
Func = fun(_Offer) -> accept end,
io:format("[test process] Func ~p~n",[Func]),
Func;
{buy_price_less_than, Price} ->
fun({_Stock, Price2}) ->
if
Price2 < Price -> accept;
true-> reject
end
end;
{buy_only, StockName} ->
fun({Stock, _Price2}) ->
case Stock of
StockName -> accept;
_ -> reject
end
end;
{both, Para1, Para2} ->
Func1 = mkstrategy(Para1),
Func2 = mkstrategy(Para2),
fun(Offer) ->
case Func1(Offer) of
accept -> case Func2(Offer) of
accept -> accept;
reject -> reject
end;
reject -> reject
```

```
    end
    end;
    long_time_strategy_accept_everything ->
    fun(_Offer) ->
    timer:sleep(10),
    accept
    end;
    long_time_strategy_reject_everything ->
    fun(_Offer) ->
    timer:sleep(10),
    reject
    end
    end.

    generate_trader(Acct) ->
    ?LET(Strategy, mkstrategy_helper(),
    ?LAZY(erlst:add_trader(Acct, Strategy))).

    %% return {InitialHolding,AccountId, TraderId}
    generate_trader_with_server(S) ->
    ?LET({Holding, AccountId}, generate_account(S), {Holding,
        AccountId, generate_trader(AccountId)}).

    %% return [{InitialHolding,AccountId, TraderId}]
    generate_trader_list_with_server(S) ->
    eqc_gen:list(generate_trader_with_server(S)).


    generate_offer_and_trader2() ->
    ?LET(S, generate_stock_exchange(), {S,
        generate_trader_list_with_server(S),
        generate_offer_list_with_server(S), eqc_gen:bool(),
        eqc_gen:bool()}).

    remove_all_trders(TraderList) ->
    lists:foldl(
    fun(X, _Acc) ->
    {_, AccountId, TraderId} = X,
    erlst:remove_trader(AccountId, TraderId),
    []
    end,
    [],
    TraderList).

    get_holding_list(AccountIdList) ->
    lists:foldl(
    fun(X, Acc) ->
    Holdings = erlst:account_balance(X),
    Acc ++ [Holdings]
```

```
    end,
    [],
    AccountIdList).


    collect_all_accountId(TraderList, OfferList) ->
    AccountList1 = lists:foldl(
    fun(X, Acc) ->
    {_, AccountId, _} = X,
    Acc ++ [AccountId]
    end,
    [],
    TraderList),
    AccountList2 = lists:foldl(
    fun(X, Acc) ->
    {_, AccountId, _} = X,
    Acc ++ [AccountId]
    end,
    [],
    OfferList),
    AccountList1 ++ AccountList2.

    calculate_initial_holding_in_offerList(OfferList, HoldingMap) ->
    case OfferList of
    [X|XS] ->
    {InitialHolding, _, _} = X,
    NewMap = calculate_initial_holding(InitialHolding, HoldingMap),
    calculate_initial_holding_in_offerList(XS, NewMap);
    [] -> HoldingMap
    end.

    %% TraderList : [{InitialHolding, AccountId, TraderId}]
    %% OfferList : [{InitialHolding, AccountId, {{StockName, Price}
        ,OfferId}}]
    calculate_initial_holding_in_traderList(TraderList, HoldingMap) ->
    case TraderList of
    [X|XS] ->
    {InitialHolding, _, _} = X,
    io:format("[test process] InitialHolding ~p~n",[InitialHolding]),
    NewMap = calculate_initial_holding(InitialHolding, HoldingMap),
    calculate_initial_holding_in_traderList(XS, NewMap);
    [] -> HoldingMap
    end.

    calculate_initial_holding(Holding, HoldingMap) ->
    {CurMoney, StockList} = Holding,
    NewMap = calculate_initial_stock_list(StockList, HoldingMap),
    CurAllMoney = maps:get("Money", NewMap, 0),
    maps:put("Money", CurAllMoney + CurMoney, NewMap).
```

```
calculate_initial_holdinglist(HoldingList, HoldingMap) ->
case HoldingList of
[X|XS] ->
{CurMoney, StockList} = X,
NewMap = calculate_initial_stock_list(StockList, HoldingMap),
CurAllMoney = maps:get("Money", NewMap, 0),
NewMap2 = maps:put("Money", CurAllMoney + CurMoney, NewMap),
calculate_initial_holdinglist(XS, NewMap2);
[] ->
HoldingMap
end.

calculate_initial_stock_list(StockList, HoldingMap) ->
case StockList of
[X|XS] ->
{StockName, Amount} = X,
OldAmount = maps:get(StockName, HoldingMap, 0),
NewMap = maps:put(StockName, OldAmount + Amount, HoldingMap),
calculate_initial_stock_list(XS, NewMap);
[]-> HoldingMap
end.
```