Final Exam

Advanced Programming

Exam ID: 25
Nov 10, 2022

Question 1: APPY: A simple parser generator

Question 1.1: The Parser module

Design

I first get rid of Left recursion and got the whole grammer as follows:

> *Spec ::= preamble ERules.*
>
> *ERules ::= ERule | ERule ERules.*

> *ERule ::= LHS "::=" Alts ".".*
>
> *LHS ::= name OptType | "_".*
>
> *OptType ::= | "{:" htext "}".*
>
> *Alts ::= Seq | Seq "|" Alts.*
>
> *Seq ::= Simple | Simplez "{" htext "}".*
>
> *Simplez ::= | Simple Simplez.*
>
> *Simple ::= Simple0 | Simple0 "?" | Simple0 "*" | "!" Simple0.*
>
> *-- Simple0 ::= Atom Simple0'.*
>
> *Simple0' ::= | "{?" htext "}" Simple0*
>
> *Atom ::= name | tokLit | "@" | charLit | "(" Alts ")".*

implement

I implemented the parsers from top to bottom, just follow the grammer and write them.

First I implemented skipmore to skip spaces and comment, this part of code I was referenced from 2021 exam sample code.Then I made token,symbol and schar from Parsernotes, using skipmore in token can make it skip spaces and comments before any token.

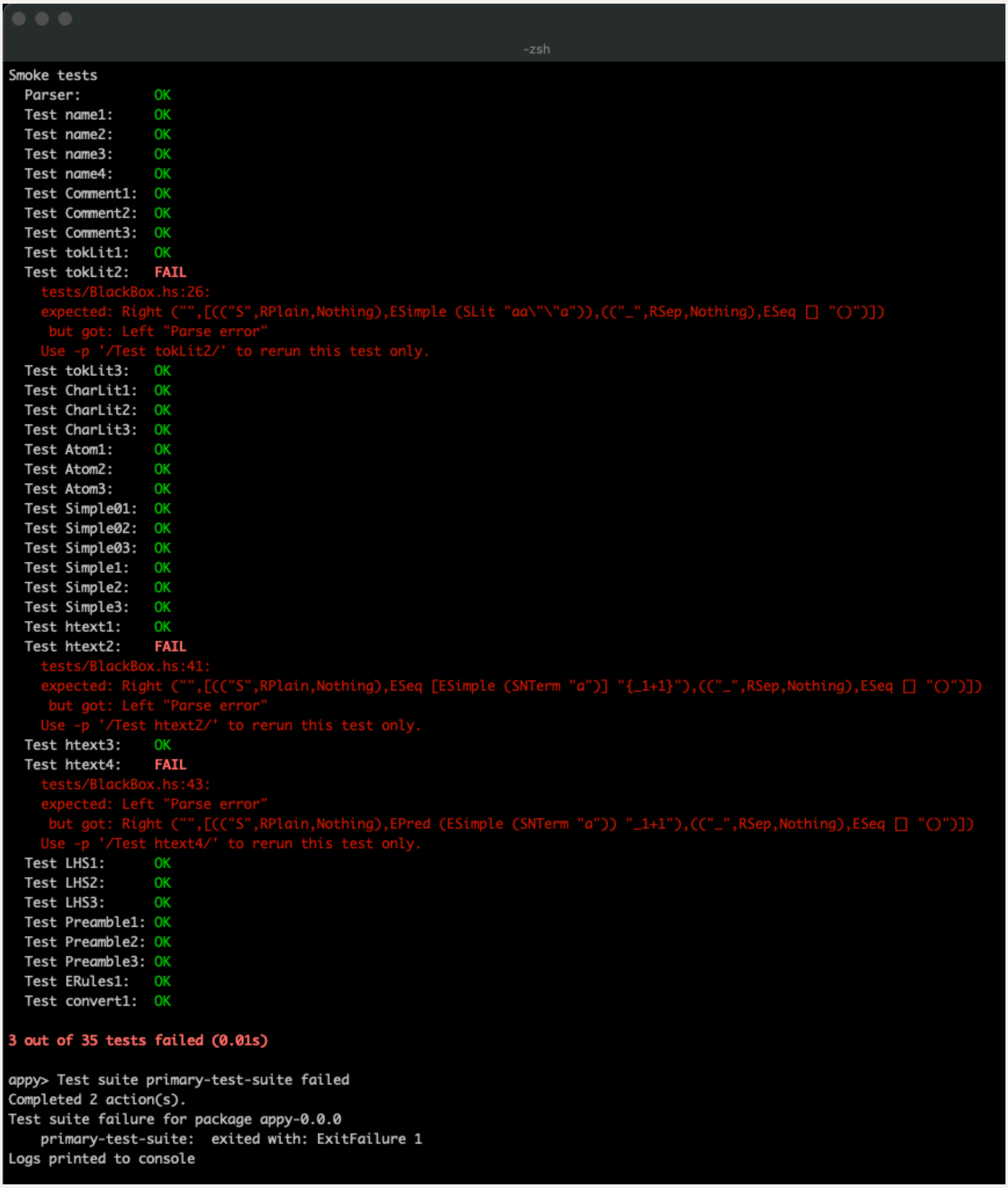
In parseLhs I used isUpper to check the head of LHS name, if it's Upper then it's a RPlain, or it's a RToken. So I just wrote them from parseErules to parseAtom. Then I made 4 small parsers to parse name(read a letter first and read a lot of letter,digit and underscores),tokit(begin with double-quotes and end with double-quotes,I didn't implement double double-quotes because I'm not quite sure how to judge if a double-quote is the ending of a TokLit or not),CharLit(begin with a single-quote and read a printable char and end with a single-quote),At(just return SAnyChar),parseHtext(read a lot of printable chars and end with '}', I didn't implement double '{'}' because I'm also not sure how to judge if it's an end or not).

Then, I implemented parsePreamble to read anything until it meets a '-', and read "---\n" and return what it just read.

While developing I found that some of my grammers may generate ambiguous results, so I used the last result in parseSpec. It's not the best way to handle it, but at least it can return the correct parsing answer.

Test

I made 34 unit tests for the parser, some of the results is failed because I didn't implyment those part, such as complicated htext and tokLit with double-quotes inside.But the rest are all performing well and following is the test result:



```
-zsh
Smoke tests
  Parser:        OK
  Test name1:    OK
  Test name2:    OK
  Test name3:    OK
  Test name4:    OK
  Test Comment1: OK
  Test Comment2: OK
  Test Comment3: OK
  Test tokLit1:  OK
  Test tokLit2:  FAIL
    tests/BlackBox.hs:26:
    expected: Right ("",[(("S",RPlain,Nothing),ESimple (SLit "aa\"\"a")),(("_",RSep,Nothing),ESeq □ "()")])
     but got: Left "Parse error"
    Use -p '/Test tokLit2/' to rerun this test only.
  Test tokLit3:  OK
  Test CharLit1: OK
  Test CharLit2: OK
  Test CharLit3: OK
  Test Atom1:    OK
  Test Atom2:    OK
  Test Atom3:    OK
  Test Simple01: OK
  Test Simple02: OK
  Test Simple03: OK
  Test Simple1:  OK
  Test Simple2:  OK
  Test Simple3:  OK
  Test htext1:   OK
  Test htext2:   FAIL
    tests/BlackBox.hs:41:
    expected: Right ("",[(("S",RPlain,Nothing),ESeq [ESimple (SNTerm "a")] "{_1+1}"),(("_",RSep,Nothing),ESeq □ "()")])
     but got: Left "Parse error"
    Use -p '/Test htext2/' to rerun this test only.
  Test htext3:   OK
  Test htext4:   FAIL
    tests/BlackBox.hs:43:
    expected: Left "Parse error"
     but got: Right ("",[(("S",RPlain,Nothing),EPred (ESimple (SNTerm "a")) "_1+1"),(("_",RSep,Nothing),ESeq □ "()")])
    Use -p '/Test htext4/' to rerun this test only.
  Test LHS1:     OK
  Test LHS2:     OK
  Test LHS3:     OK
  Test Preamble1: OK
  Test Preamble2: OK
  Test Preamble3: OK
  Test ERules1:  OK
  Test convert1: OK

3 out of 35 tests failed (0.01s)

appy> Test suite primary-test-suite failed
Completed 2 action(s).
Test suite failure for package appy-0.0.0
    primary-test-suite:  exited with: ExitFailure 1
Logs printed to console
```

Question 1.2: The Transformer module

I managed to implement a version of convert which can convert the test sample eg into g, it uses patton matching to get the rhs, and then I check the type of it, if it's ESimple then I will return a list with AUser "". If it's a ESeq I'll return the list generated by the helper function removeEsimple with Auser which is the second parameter of ESeq.If it's a EBar, I first check the two parameters of EBar, if they are both ESeq I will return a list with two ESeq's convert result.This simple convert function can successfully convert the sample eg into g.

Question 2: Stock Exchange

Question 2.1: The erlst module

Topics

- I support the complete API
- My trader can only process one Strategy at a time, so it needs to be not so slow.
- I uses 2 processes, one is the main exchange, one is the Strategy spawned process. Main exchange uses gen_server APIs to receive messages from user, Strategy send function result to main process through "!"
- The data is shown below as the status
- These parts are discussed in implement part

Design

I read through the questions first and constructed the status of the system. As there's 3 main data sets we need to save, I added 3 lists representing Accounts,Offers and Traders.Each Account has an account ID and a holding.

In designing account ID I found there're some functions that don't have server Pid as input, but have just an Acct. So I think AccountID needs to contain it's server Pid, in that way it can get the server which this account was opened in, so the accountID is {S,Id}.For the Id part we need a unique id for each account, there's multiple ways to do it, including setting a user count in state or generate a random unique number each time. I used the second one because I don’t want to add extra data in status which may result in the server running slow. I used erlang:now() to get the current time. I used erlang:localtime() first but during test I figured out that the minimum unit of time is seconds and if I call it several times by sequence erlang will do it in a second, and their ID will be the same. So I used now( ) instead although it's been deprecated. I used lists: concat( )to concat ‘A' and 3 results meaning it's an accountID. Also I used the same code in generating OfferID and TraderID with initials of O and T.

In designing Offers as we need to store an accountID with the offer to do the money/ stock transfer later I added AccountID into the offer tuple. Also we need to add AccountID into Trader to do the following trades.

As we need to return the number of trades I also add a number into the status to count for it, I will discuss it later.

The following is the status I designed:

> *State is [Num,Accounts, Offers, Traders]*
> *Num is the number of trades*
> *Accounts is [{{S,Id},{isk,[{stock,amount}.....]}}....]*
> *Offers is [{OfferId,AccountId,{stock,isk}}....]*
> *Traders is [{TraderId,AccountId,Strategy}....]*
> *Strategy = fun((Offer) -> Decision)*

implement

I used gen_server for the whole system and reviewed my code from assignment 6 to recall how to use APIs in gen_server.

Open_account

It will receive the server and new account's holding and make a call to the server. Server will first generate an ID and use a helper function checkHoldings(Holdings).This function will check the input Holding if the ISK >= 0. Then it will call another function checkStocks(). This function will check if each of the amounts >=0. Then if checkHoldings return true we will reply it’s AcctID and also add this item into State.If it’s false we will return {error,badHoldings} to show it’s not a valid holding.

Make_offer

I used a helper function check_Account_exist to run account_balance with a given Id, and judge its result and return true or false. If it's a valid Acct then we call the server with input Offer and Id from Acct. In server I first judge if the offer is a valid tuple, then if it's valid we check if the offer's price is valid.If all of these conditions are met we generate an OfferID and add the new offer into the state.

Add_trader

Then it comes the most complicated part of the server, the trader.

The task asked me to execute the trade whenever 4 conditions are met. I think we can define a function to execute each tradeable trade. As the 4 conditions may only be met when addoffer and addtrader is called, so I put the function above into handle_call of make_offer and add_trader.

My design of helper functions in doing this is as follows:

- check_offer_exists which will check if the offer list is empty.
- run_strategy: run a strategy on a single offer. In this function I spawned a new process and in this process it runs the strategy over offer and sends back the result after the function gets the result to run_strategy process. And in the main time run_strategy receives the result and returns it. In this part I didn't implement

> *"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."*

The reason of it is when I get the idea of doing it there's not enough time for me to change the whole structure, I think it maybe can be solved by using lists:foreach for each Strategy to spawn a new process and send back the result, and the main process receive the fastest result and return it's traderID and result.

- Map_offer: use one trider run al offer, return a list of results the Trader made to each of the offers,which may looks like this: [accept,reject,accept.....]. this list is called Map_Res.Here each result is a tuple, it's look like {reject,Buyer, {OfferId,Seller,{Stock,Price}}}.With these info we can proceed to handle changing accounts.

- Run_all_traders: use all traders to run all offer, each trader will return a list with result, so the function will return a list of result lists, which may looks like this: [[accept,reject],[reject,reject],[accept,accept]...],this list is called Map_Reses
- Remove_offer: it will receive Map_Res and Offers, remove those offers which is accept in Map_Res.
- Transfer_stock(Map_Res,Accounts): I used a helper function with 3 parameters in this part.As I remove offer based on Map_Res, sometimes Map_Res is accept, but the holdings of two side doesn't meet the requirement, so although the trade was not exec,the offer is also got removed. So this function will return new Map_Res and new Accounts.In helper function it has a new para NewMap_Res to collect each of new results.In this function it will go through each Map_Res, if it's accept it will check buyer's ISK and then seller's stock, then do the transfer or do nothing to the Accounts list.
- Run_all_Map_Res(Map_Reses,Accounts,Offers,Num):Recursive traversal Map_Reses and use above two functions to change Accounts and Offers. Also I changed the trade num by subtracting old Offers length with new Offers length and add the change to Original Num.Then it returns the final status after trading.

assessment

- Completeness :I completed all the APIs

- Correctness: I made 16 unit tests and 2 props and both passed(will discuss next)
- Robustness: I made several robust upgrade to erlst after running eqc and unit test,including fixed a bug that change_stock may return a list with {error,aaa}inside, fixed several functions may crush when receiving a tuple.Before these my erlst crushes a lot when running prop test but now it doesn't crush any more.However,because of time I didn't implement the robust requirement, but I think both of them can be resolved with supervisor behavior.
- Maintainability: I used a lot of helper functions in the code which makes it very easy to find bug and fix it. Also my code's neatly arranged and each section has it's name, like *Trader helper functions*,*server API* and so on.

2.2 Testing erlst

Topic

- I had implemented all required parts

Design

I implemented the function mkstrategy with 4 possible choices: {buy_everything,buy_cheap,buy_only,both}and each of them can return a strategy function.in both function it will only return accept when two functions generated by mkstrategy both return accept.

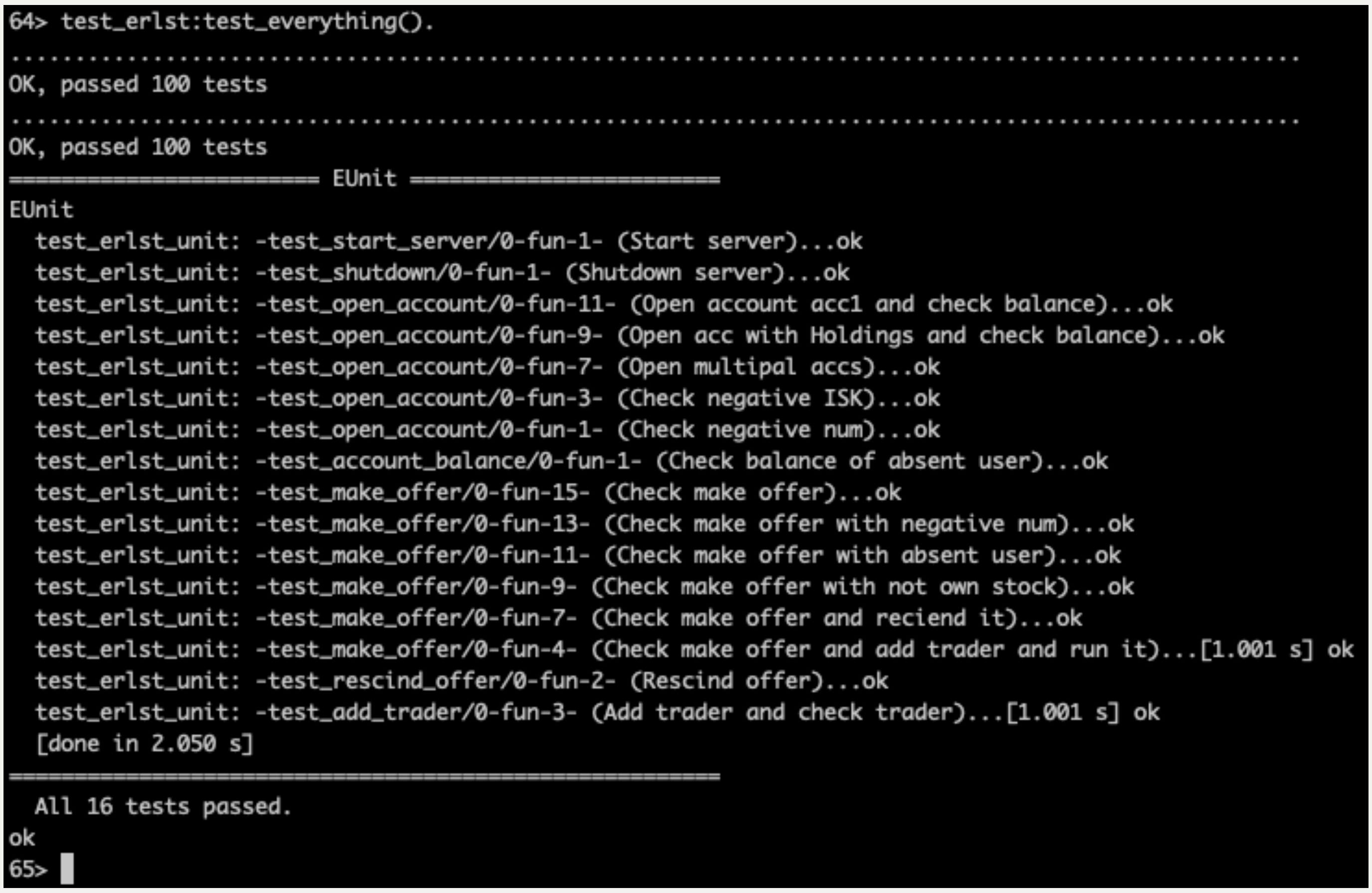
Then I implemented some other generators, stock_name can generate a name from[a,b,c,d,e,f], in stock_list,holdings and offer I used ?SUCHTHAT to check if the price > 0, this can make sure they can generate legal ISK and prices.

Then I implemented reliable_strategy generator. It used oneof to generate a symbolic call from four choices regarding to four choices of mkstrategy.

Then I implemented prop_value_preservation().This prop I need to check the whole amount of money and stock, I first added two accounts, and each of then make 4 offers and 4 traders.And then check the money and stock. I used two helper functions calc_Total_Money and calc_Total_Stock to get current whole money and stock.Then I implemented prop_total_trades(), it also gen 2 accounts and each make 4 offers and 4 traders.Then the number of make_offer calls is 8 and number of trades is the number shutdown returns.

I also added 16 unit tests to test some aspects prop may not be able to reach, like negative ISK, negative stock price or offer price and so on, and my erlst passed all the tests.

Following is the result of testing:



```
64> test_erlst:test_everything().
....................................................................................................
OK, passed 100 tests
....................................................................................................
OK, passed 100 tests
======================== EUnit ========================
EUnit
  test_erlst_unit: -test_start_server/0-fun-1- (Start server)...ok
  test_erlst_unit: -test_shutdown/0-fun-1- (Shutdown server)...ok
  test_erlst_unit: -test_open_account/0-fun-11- (Open account acc1 and check balance)...ok
  test_erlst_unit: -test_open_account/0-fun-9- (Open acc with Holdings and check balance)...ok
  test_erlst_unit: -test_open_account/0-fun-7- (Open multipal accs)...ok
  test_erlst_unit: -test_open_account/0-fun-3- (Check negative ISK)...ok
  test_erlst_unit: -test_open_account/0-fun-1- (Check negative num)...ok
  test_erlst_unit: -test_account_balance/0-fun-1- (Check balance of absent user)...ok
  test_erlst_unit: -test_make_offer/0-fun-15- (Check make offer)...ok
  test_erlst_unit: -test_make_offer/0-fun-13- (Check make offer with negative num)...ok
  test_erlst_unit: -test_make_offer/0-fun-11- (Check make offer with absent user)...ok
  test_erlst_unit: -test_make_offer/0-fun-9- (Check make offer with not own stock)...ok
  test_erlst_unit: -test_make_offer/0-fun-7- (Check make offer and reciend it)...ok
  test_erlst_unit: -test_make_offer/0-fun-4- (Check make offer and add trader and run it)...[1.001 s] ok
  test_erlst_unit: -test_rescind_offer/0-fun-2- (Rescind offer)...ok
  test_erlst_unit: -test_add_trader/0-fun-3- (Add trader and check trader)...[1.001 s] ok
  [done in 2.050 s]
=======================================================
  All 16 tests passed.
ok
65> 
```

assessment

- Completeness :I completed two props and some other helper functions

- Correctness: I used the props to test my own erlst and found several bugs in my system until all tests passed.

Appendix

ParserImpl

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

import Definitions
-- import ReadP or Parsec
import Text.ParserCombinators.ReadP
import Control.Applicative ((<|>))
import Data.Char

type Parser a = ReadP a

type ParseError = String


-- Spec ::= preamble ERules.
parseSpec :: String -> EM (String, EGrammar)
parseSpec str = case readP_to_S (do
  preamble <- parsePreamble
  eRules <- parseErules
  eof
  return (preamble, eRules)) str of
  [] -> Left "Parse error"
  x -> case last x of
    (x, "") -> return x
    _ -> Left "Parse error"


spec :: Parser (String, EGrammar)
spec = do
    preamble <- parsePreamble
    eRules <- parseErules
    return (preamble, eRules)

-- ERules ::= ERule | ERule ERules.
parseErules :: Parser [ERule]
parseErules = do
            skipmore
            e <- parseErule
            es <- parseErules
            return (e:es)
        <|> do
            e <- parseErule
            return [e]

-- ERule :: LHS "::=" Alts ".".
parseErule :: Parser ERule
parseErule = do
            (l1,l2,l3) <- parseLhs
            symbol "::="
            a <- parseAlts
            schar '.'
            return ((l1,l2,l3),a)

-- LHS ::= name OptType | "_".
parseLhs :: Parser RLHS
parseLhs = do
        skipmore
        n <- parseName
        o <- parseOptType
        if isUpper (head n)
            then return (n, RPlain, o)
            else return (n, RToken, o)
    <|> do
        skipmore
        symbol "_"
        return ("_",RSep,Nothing)

-- OptType ::= | "{:" htext "}".
parseOptType :: Parser (Maybe Type)
parseOptType = do
            schar '{'
            schar ':'
            h <- parseHtext
            schar '}'
            return (Just (AUser h))
        <|> return Nothing
```

```haskell
-- Alts ::= Seq | Seq "|" Alts.
parseAlts :: Parser ERHS
parseAlts = do
        skipmore
        s <- parseSeq
        symbol "|"
        EBar s <$> parseAlts
       <|> do
        skipmore
        parseSeq

-- Seq ::= Simple | Simplez "{" htext "}".
parseSeq :: Parser ERHS
parseSeq = do
        s <- parseSimplez
        schar '{'
        h <- parseHtext
        schar '}'
        return (ESeq s h)
    <|> parseSimple

-- Simplez ::= | Simple Simplez.
parseSimplez :: Parser [ERHS]
parseSimplez = do
            s <- parseSimple
            ss <- parseSimplez
            return (s:ss)
        <|> return []

-- Simple ::= Simple0 | Simple0 "?" | Simple0 "*" | "!" Simple0.
parseSimple :: Parser ERHS
parseSimple = do
            s <- parseSimple0
            schar '?'
            return (EOption s)
        <|> do
            s <- parseSimple0
```

```haskell
                schar '*'
                return (EMany s)
            <|> do
                schar '!'
                ENot <$> parseSimple0
            <|> parseSimple0

-- Simple0 ::= Atom Simple0'.
parseSimple0 :: Parser ERHS
parseSimple0 = do
                a <- parseAtom
                parseSimple0' a

-- Simple0' ::= | "{?" htext "}" Simple0'.
parseSimple0' :: ERHS -> Parser ERHS
parseSimple0' a = do
                    schar '{'
                    schar '?'
                    h <- parseHtext
                    schar '}'
                    parseSimple0' (EPred a h)
                <|> return a


-- Atom ::= name | tokLit | "@" | charLit | "(" Alts ")".
parseAtom :: Parser ERHS
parseAtom = do
            ESimple . SNTerm <$> parseName
        <|> do
            ESimple . SLit <$> parseTokLit
        <|> parseAt
        <|> do
            ESimple . SChar <$> parseCharLit
        <|> do
            schar '('
            a <- parseAlts
            schar ')'
            return a
```

```
parseName :: Parser String
parseName = token(do
        c <- satisfy isLetter
        cs <- munch (\c -> isLetter c || isDigit c || c == '_')
        return (c:cs))

parseTokLit :: Parser String
parseTokLit = token(do
        schar '"'
        cs <- munch (\c -> c /= '"')
        schar '"'
        return cs)



parseCharLit :: Parser Char
parseCharLit = do
            schar '\''
            c <- satisfy isPrint
            schar '\''
            return c

parseAt :: Parser ERHS
parseAt = do
        schar '@'
        return (ESimple SAnyChar)

parseHtext :: Parser String
parseHtext = munch (\c -> isPrint c  && c /= '}')




parsePreamble :: Parser String
parsePreamble = do
    p <- munch (/= '-')
    symbol "---\n"
    return p
```

```
-- reference from the last haskell lecture code
skipmore :: Parser ()
skipmore = do
    do
        skipSpaces;
        optional (
            do
                string "--";
                endComment;
                skipmore
            )
endComment :: Parser ()
endComment = do
    char '\n'
    return ()
    <++ do
        get;
        endComment



-- Reference from Parsernotes
token :: Parser a -> Parser a
token p = skipmore >> p
symbol :: String -> Parser String
symbol = token . string
schar :: Char -> Parser Char
schar = token . char
```

TransformerImpl

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

import Definitions

convert :: EGrammar -> EM Grammar
convert [] = Right []
```

```haskell
convert [(lhs, rhs)] = case rhs of
    EBar r1 r2 -> case (r1, r2) of
        (ESeq s1 u1, ESeq s2 u2) -> Right [(lhs, [(removeEsimple s1,
AUser u1), (removeEsimple s2, AUser u2)])];
        _ -> Left "Error"
    ESeq s u -> Right [(lhs, [(removeEsimple s, AUser u)])]
    ESimple s -> Right [(lhs, [([s], AUser "")])]
    _ -> Left "Error"
convert ((lhs, rhs):xs) =
    case convert [(lhs, rhs)] of
        Right r -> case convert xs of
            Right rs -> Right (r ++ rs)
            Left e -> Left e
        Left e -> Left e


removeEsimple :: [ERHS] -> [Simple]
removeEsimple [] = []
removeEsimple (x:xs) = case x of
    ESimple s -> s : removeEsimple xs


lre :: Grammar -> EM Grammar
lre = undefined

lfactor :: Grammar -> EM Grammar
lfactor = undefined
```

BlackBox.hs

```haskell
-- 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.
```

```haskell
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 "Test name1" $ parseSpec str1 @?= Right ("", eg2),
       testCase "Test name2" $ parseSpec str2 @?= Right ("", eg3),
       testCase "Test name3" $ parseSpec str3 @?= Right ("", eg4),
       testCase "Test name4" $ parseSpec str4 @?= Left "Parse error",
       testCase "Test Comment1" $ parseSpec str5 @?= Right ("", eg5),
       testCase "Test Comment2" $ parseSpec str6 @?= Right ("", eg6),
       testCase "Test Comment3" $ parseSpec str7 @?= Right ("", eg7),
       testCase "Test tokLit1" $ parseSpec str8 @?= Right ("", eg8),
       testCase "Test tokLit2" $ parseSpec str9 @?= Right ("", eg9),
       testCase "Test tokLit3" $ parseSpec str10 @?= Left "Parse
error",
       testCase "Test CharLit1" $ parseSpec str11 @?= Right ("",
eg11),
       testCase "Test CharLit2" $ parseSpec str12 @?= Left "Parse
error",
       testCase "Test CharLit3" $ parseSpec str13 @?= Left "Parse
error",
       testCase "Test Atom1" $ parseSpec str14 @?= Right ("", eg14),
       testCase "Test Atom2" $ parseSpec str15 @?= Right ("", eg15),
       testCase "Test Atom3" $ parseSpec str16 @?= Right ("", eg16),
       testCase "Test Simple01" $ parseSpec str17 @?= Right ("",
eg17),
       testCase "Test Simple02" $ parseSpec str18 @?= Right ("",
eg18),
       testCase "Test Simple03" $ parseSpec str19 @?= Right ("",
eg19),
       testCase "Test Simple1" $ parseSpec str20 @?= Right ("",
eg20),
       testCase "Test Simple2" $ parseSpec str21 @?= Right ("",
eg21),
       testCase "Test Simple3" $ parseSpec str22 @?= Right ("",
eg22),
       testCase "Test htext1" $ parseSpec str23 @?= Right ("", eg23),
       testCase "Test htext2" $ parseSpec str24 @?= Right ("", eg24),
       testCase "Test htext3" $ parseSpec str25 @?= Left "Parse
error",
       testCase "Test htext4" $ parseSpec str26 @?= Left "Parse
error",
       testCase "Test LHS1" $ parseSpec str27 @?= Right ("", eg27),
       testCase "Test LHS2" $ parseSpec str28 @?= Right ("", eg28),
       testCase "Test LHS3" $ parseSpec str29 @?= Right ("", eg29),
       testCase "Test Preamble1" $ parseSpec str30 @?= Right
("asdasdasd", eg30),
       testCase "Test Preamble2" $ parseSpec str31 @?= Right ("a s d
a s d a s d ", eg31),
       testCase "Test Preamble3" $ parseSpec str32 @?= Right ("a s d
a s d a s d ", eg32),
       testCase "Test ERules1" $ parseSpec str33 @?= Right ("",
eg33),
       testCase "Test convert" $ convert eg @?= Right g
       ]
       -- convert eg @?= Right g] -- assumes that convert preserves
input rule order
       where
       str = "---\n S ::= S \"a\" {_1+1} | \"b\" {0}.\n _ ::= {()}."
       eg = [(("S", RPlain, Nothing),
              EBar (ESeq [ESimple (SNTerm "S"), ESimple (SLit "a")]
"_1+1")
                           (ESeq [ESimple (SLit "b")] "0")),
              (("_", RSep, Nothing), ESeq [] ("()"))]
       -- Test name1
       str1 = "---\n S ::= a.\n _ ::= {()}."
       eg2 = [(("S", RPlain, Nothing),
              ESimple (SNTerm "a")),
              (("_", RSep, Nothing), ESeq [] ("()"))]
       -- Test name2
       str2 = "---\n S ::= asdf1234.\n _ ::= {()}."
       eg3 = [(("S", RPlain, Nothing),
              ESimple (SNTerm "asdf1234")),
              (("_", RSep, Nothing), ESeq [] ("()"))]
       -- Test name3
       str3 = "---\n S ::= a________1.\n _ ::= {()}."
       eg4 = [(("S", RPlain, Nothing),
              ESimple (SNTerm "a________1")),
              (("_", RSep, Nothing), ESeq [] ("()"))]
       -- Test name4
       str4 = "---\n S ::= 123.\n _ ::= {()}."
       -- Test Comment1
       str5 = "---\n S --ppp\n::= --ppp\nS \"a\" {_1+1} | \"b\"
{0}.\n _ ::= {()}."
       eg5 = [(("S", RPlain, Nothing),
              EBar (ESeq [ESimple (SNTerm "S"), ESimple (SLit "a")]
"_1+1")
                            (ESeq [ESimple (SLit "b")] "0")),
              (("_", RSep, Nothing), ESeq [] ("()"))]
       -- Test Comment2
       str6 = "---\n S --ppp\n--ppp\n--ppp\n::= S \"a\" {_1+1} |
\"b\" {0}.\n _ ::= {()}."
       eg6 = [(("S", RPlain, Nothing),
              EBar (ESeq [ESimple (SNTerm "S"), ESimple (SLit "a")]
"_1+1")
                            (ESeq [ESimple (SLit "b")] "0")),
              (("_", RSep, Nothing), ESeq [] ("()"))]
       -- Test Comment3
       str7 = "---\n S --ppp\n   --ppp\n   --ppp\n::= S \"a\" {_1+1}
|  --aqqwe\n \"b\" {0}.\n _ ::= {()}."
       eg7 = [(("S", RPlain, Nothing),
              EBar (ESeq [ESimple (SNTerm "S"), ESimple (SLit "a")]
"_1+1")
                            (ESeq [ESimple (SLit "b")] "0")),
              (("_", RSep, Nothing), ESeq [] ("()"))]
       -- Test tokLit1
       str8 = "---\n S ::= \"a\".\n _ ::= {()}."
       eg8 = [(("S", RPlain, Nothing),
               ESimple (SLit "a")),
               (("_", RSep, Nothing), ESeq [] ("()"))]
       -- Test tokLit2
       str9 = "---\n S ::= \"aa\"a\".\n _ ::= {()}."
       eg9 = [(("S",RPlain,Nothing),ESimple (SLit "aa\"\"a")),
(("_",RSep,Nothing),ESeq [] "()")]
       -- Test tokLit3
       str10 = "---\n S ::= \"a\" \"\"\".\n _ ::= {()}."
       -- Test CharLit1
       str11 = "---\n S ::= \'a\'.\n _ ::= {()}."
       eg11 = [(("S",RPlain,Nothing),ESimple (SChar 'a')),
(("_",RSep,Nothing),ESeq [] "()")]
       -- Test CharLit2
       str12 = "---\n S ::= \'a\' \'b\'.\n _ ::= {()}."
       -- Test CharLit3
       str13 = "---\n S ::= \'a\'\'.\n _ ::= {()}."
       -- Test Atom1
       str14 = "---\n S ::= a.\n _ ::= {()}."
       eg14 = [(("S",RPlain,Nothing),ESimple (SNTerm "a")),
(("_",RSep,Nothing),ESeq [] "()")]
       -- Test Atom2
       str15 = "---\n S ::= @.\n _ ::= {()}."
       eg15 = [(("S",RPlain,Nothing),ESimple SAnyChar),
(("_",RSep,Nothing),ESeq [] "()")]
       -- Test Atom3
       str16 = "---\n S ::= (a).\n _ ::= {()}."
       eg16 = [(("S",RPlain,Nothing),ESimple (SNTerm "a")),
(("_",RSep,Nothing),ESeq [] "()")]
       -- Test Simple01
       str17 = "---\n S ::= a.\n _ ::= {()}."
       eg17 = [(("S",RPlain,Nothing),ESimple (SNTerm "a")),
(("_",RSep,Nothing),ESeq [] "()")]
       -- Test Simple02
       str18 = "---\n S ::= @{?aaa}.\n _ ::= {()}."
       eg18 = [(("S",RPlain,Nothing),EPred (ESimple SAnyChar) "aaa"),
(("_",RSep,Nothing),ESeq [] "()")]
       -- Test Simple03
       str19 = "---\n S ::= @{? isNum }.\n _ ::= {()}."
```

```
        eg19 = [(("S",RPlain,Nothing),EPred (ESimple SAnyChar) " isNum
"),(("_",RSep,Nothing),ESeq [] "()")]
        -- Test Simple1
        str20 = "---\n S ::= a*.\n _ ::= {()}."
        eg20 = [(("S",RPlain,Nothing),EMany (ESimple (SNTerm "a"))),
(("_",RSep,Nothing),ESeq [] "()")]
        -- Test Simple2
        str21 = "---\n S ::= a?.\n _ ::= {()}."
        eg21 = [(("S",RPlain,Nothing),EOption (ESimple (SNTerm "a"))),
(("_",RSep,Nothing),ESeq [] "()")]
        -- Test Simple3
        str22 = "---\n S ::= !a.\n _ ::= {()}."
        eg22 = [(("S",RPlain,Nothing),ENot (ESimple (SNTerm "a"))),
(("_",RSep,Nothing),ESeq [] "()")]
        -- Test htext1
        str23 = "---\n S ::= a {_1+1}.\n _ ::= {()}."
        eg23 = [(("S",RPlain,Nothing),ESeq [ESimple (SNTerm "a")]
"_1+1"),(("_",RSep,Nothing),ESeq [] "()")]
        -- Test htext2
        str24 = "---\n S ::= a {{_1+1}}.\n _ ::= {()}."
        eg24 = [(("S",RPlain,Nothing),ESeq [ESimple (SNTerm "a")] "
{_1+1}"),(("_",RSep,Nothing),ESeq [] "()")]
        -- Test htext3
        str25 = "---\n S ::= a {:_1+1} b.\n _ ::= {()}."
        -- Test htext4
        str26 = "---\n S ::= a {?_1+1}.\n _ ::= {()}."
        -- Test LHS1
        str27 = "---\n S {:P}::= a.\n _ ::= {()}."
        eg27 = [(("S",RPlain,Just (AUser "P")),ESimple (SNTerm "a")),
(("_",RSep,Nothing),ESeq [] "()")]
        -- Test LHS2
        str28 = "---\n s {:P} ::= a.\n _ ::= {()}."
        eg28 = [(("s",RToken,Just (AUser "P")),ESimple (SNTerm "a")),
(("_",RSep,Nothing),ESeq [] "()")]
        -- Test LHS3
        str29 = "---\n _ ::= {()}."
        eg29 = [(("_",RSep,Nothing),ESeq [] "()")]
        -- Test Preamble1
        str30 = "asdasdasd---\n S ::= a.\n _ ::= {()}."
```

```
       eg30 = [(("S", RPlain, Nothing),
              ESimple (SNTerm "a")),
              (("_", RSep, Nothing), ESeq [] ("()"))]
       -- Test Preamble2
       str31 = "a s d a s d a s d ---\n S ::= a.\n _ ::= {()}."
       eg31 = [(("S", RPlain, Nothing),
              ESimple (SNTerm "a")),
              (("_", RSep, Nothing), ESeq [] ("()"))]
       -- Test Preamble3
       str32 = "a s d a s d a s d ---\n --asas\n S ::= a.\n _ ::=
{()}."
       eg32 = [(("S", RPlain, Nothing),
              ESimple (SNTerm "a")),
              (("_", RSep, Nothing), ESeq [] ("()"))]
       -- Test ERules1
       str33 = "---\n S ::= S \"a\" {_1+1} | \"b\" {0}.\n _ ::=
{()}."
       eg33 = [(("S", RPlain, Nothing),
              EBar (ESeq [ESimple (SNTerm "S"), ESimple (SLit "a")]
"_1+1")
                            (ESeq [ESimple (SLit "b")] "0")),
              (("_", RSep, Nothing), ESeq [] ("()"))]
       -- Transformer
       -- 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 "()")])]
```

erlst.erl

```erlang
-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,
         show/1
        ]).

% You may have other exports as well
-export([init/1,terminate/2,handle_call/3,handle_cast/2]).
-
export([removeoffer/3,run_strategy/2,check_offer_exists/1,run_all_tra
ders/2,run_all_Map_Res/4
    ,transfer_stock/2,change_stock/4]).

-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()).
```

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

-spec shutdown(S :: stock_exchange()) -> non_neg_integer().
shutdown(S) ->
    Ret = gen_server:call(S, shutdown),
    Ret.

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

-spec account_balance(Acct :: account_id()) -> holdings().
account_balance(Acct) ->
    case Acct of
        {Server,Id} ->
            gen_server:call(Server, {account_balance, Id});
        _ ->
            {error, badAcct}
    end.

-spec make_offer(Acct :: account_id(), Terms :: offer()) -> {ok,
offer_id()} | {error, term()}.
make_offer(Acct, Offer) ->
    case check_Account_exist(Acct) of
        true ->
            case Acct of
                {Server,Id} ->
                    gen_server:call(Server, {make_offer, Offer, Id});
                _ ->
                    {error, badAcct}
            end;
        false ->
            {error, notexist}
    end.
```

```
-spec rescind_offer(Acct :: account_id(), Offer :: offer_id()) -> ok.
rescind_offer(Acct, Offer) ->
    case check_Account_exist(Acct) of
        true ->
            case Acct of
                {Server,Id} ->
                    gen_server:cast(Server, {rescind_offer, Offer,
Id});
                _ ->
                    {error, badAcct}
            end;
        false ->
            {error, notexist}
    end.

-spec add_trader(Acct :: account_id(), Strategy :: trader_strategy())
-> trader_id().
add_trader(Acct, Strategy) ->
    case check_Account_exist(Acct) of
        true ->
            case Acct of
                {Server,Id} ->
                    gen_server:call(Server, {add_trader, Strategy,
Id});
                _ ->
                    {error, badAcct}
            end;
        false ->
            {error, notexist}
    end.


-spec remove_trader(Acct :: account_id(), Trader :: trader_id()) ->
ok.
remove_trader(Acct, Trader) ->
    case check_Account_exist(Acct) of
        true ->
            case Acct of
                {Server,Id} ->
```

```
                    gen_server:cast(Server, {remove_trader, Trader,
Id});
                _ ->
                    {error, badAcct}
            end;
        false ->
            {error, notexist}
    end.

show(S) ->
    gen_server:call(S, show).

% --------------------------server API----------------
% State is [Num,Accounts, Offers, Traders]
% Num is the number of trades
% Accounts is [{{S,Id},{isk,[{stock,amount}.....]}}....]
% Offers is [{OfferId,AccountId,{stock,isk}}....]
% Traders is [{TraderId,AccountId,Strategy}....]
% Strategy = fun((Offer) -> Decision), i.e. fun({"Erlang Inc",
Price}) when Price =< 42 -> accept; (_) -> reject end.
init([]) ->
    {ok, [0,[],[],[]]}.

handle_call({open_account, Holdings,Server}, _From, [Num, Accounts,
Offers, Traders]) ->
    {H,M,S} = erlang:now(),
    AccountId = lists:concat(['A',H,M,S]),
    case checkHoldings(Holdings) of
        true ->
            {reply, {Server,AccountId}, [Num,
[{{Server,AccountId},Holdings}|Accounts], Offers, Traders]};
        false ->
            {reply, {error, badHoldings}, [Num, Accounts, Offers,
Traders]}
    end;
handle_call(show, _From, [Num, Accounts, Offers, Traders]) ->
    io:format("Num: ~p~n",[Num]),
    io:format("Accounts: ~p~n",[Accounts]),
    io:format("Offers: ~p~n",[Offers]),
    io:format("Traders: ~p~n",[Traders]),
    {reply,{Accounts,Offers,Traders}, [Num, Accounts, Offers,
Traders]};
handle_call({make_offer,Offer,Id}, _From, [Num, Accounts, Offers,
Traders]) ->
    case Offer of
        {_, Price} ->
            if Price >= 0 ->
                {H,M,S} = erlang:now(),
                OfferId = lists:concat(['O',H,M,S]),
                NewOffers = [{OfferId,Id,Offer}|Offers],

                % Run Traders
                Map_Reses = run_all_traders(NewOffers, Traders),
                {NewAccounts, NewOffers1,NewNum} =
run_all_Map_Res(Map_Reses, Accounts, NewOffers,Num),
                {reply, {ok, OfferId}, [NewNum, NewAccounts,
NewOffers1, Traders]};
            true ->
                {reply, {error, badPrice}, [Num, Accounts, Offers,
Traders]}
            end;
        _ ->
            {reply, {error, badOffer}, [Num, Accounts, Offers,
Traders]}
    end;
handle_call({add_trader,Strategy,Id}, _From, [Num, Accounts, Offers,
Traders]) ->
    {H,M,S} = erlang:now(),
    TraderId = lists:concat(['T',H,M,S]),
    NewTraders = [{TraderId, Id, Strategy}|Traders],

    % Run Traders
    Map_Reses = run_all_traders(Offers, NewTraders),
    {NewAccounts, NewOffers,NewNum} = run_all_Map_Res(Map_Reses,
Accounts, Offers,Num),
    {reply, TraderId, [NewNum, NewAccounts, NewOffers, NewTraders]};
handle_call(shutdown, _From, [Num, Accounts, Offers, Traders]) ->
    {stop, normal, Num, [Num, Accounts, Offers, Traders]};
```

```
handle_call({account_balance,Id}, _From, [Num, Accounts, Offers,
Traders]) ->
    Balance = getBalance(Id, Accounts),
    {reply, Balance, [Num, Accounts, Offers, Traders]}.

handle_cast({rescind_offer,OfferId,Id}, [Num, Accounts, Offers,
Traders]) ->
    NewOffers = removeoffer(OfferId, Id, Offers),
    {noreply,[Num, Accounts, NewOffers, Traders]};
handle_cast({remove_trader,TraderId,Id}, [Num, Accounts, Offers,
Traders]) ->
    NewTraders = removetrader(TraderId, Id, Traders),
    {noreply,[Num, Accounts, Offers, NewTraders]}
.

terminate(_Reason, _State) ->
    ok.

% --------------------------helper functions-------------------------
---
checkHoldings(Holdings) ->
    case Holdings of
        {ISK, STOCKS} ->
            if ISK >= 0 ->
                checkStocks(STOCKS);
            true ->
                false
            end;
        _ ->
            false
    end.

% adasd
checkStocks(STOCKS) ->
    case STOCKS of
        [] ->
            true;
        {_, Amount} ->
            if Amount >= 0 ->
```

```erlang
                    true;
                true ->
                    false
                end;
            [{_,Amount}|STOCKS1] ->
                if Amount >= 0 ->
                    checkStocks(STOCKS1);
                true ->
                    false
                end
    end.


getBalance(Id, Accounts) ->
    case Accounts of
        [] ->
            {error, empty};
        [{{_,Id},{Balance,Share}}|_] ->
            {Balance,Share};
        [_|T] ->
            getBalance(Id, T)
    end.

removeoffer(OfferId, Id, Offers) ->
    case Offers of
        [] ->
            [];
        [{OfferId, Id, A}|_] ->
            Offers -- [{OfferId, Id, A}];
        [_|T] ->
            removeoffer(OfferId, Id, T)
    end.

removetrader(TraderId, Id, Traders) ->
    case Traders of
        [] ->
            [];
        [{TraderId, Id, A}|_] ->
            Traders -- [{TraderId, Id, A}];
        [_|T] ->
            removetrader(TraderId, Id, T)
    end.


check_Account_exist(Id) ->
    Balance = account_balance(Id),
    case Balance of
        {error, empty} ->
            false;
        _ ->
            true
    end.

% -----------------------------Trader helper functions---------------
-----------------
% condition1
check_offer_exists(Offers) ->
    case Offers of
        [] ->
            false;
        _ ->
            true
    end.

% Use Strategy handle one offer
run_strategy(Strategy, Offer) ->
    Me = self(),
    spawn(fun() ->
        case Strategy(Offer) of
            accept ->
                Me ! accept;
            reject ->
                Me ! reject;
            _ ->
                Me ! reject
        end
    end),
    receive
```

```erlang
        accept ->
            accept;
        reject ->
            reject
    end.

% One trider run all offer
map_offer(Offers, Trader) ->
    {_, Buyer, Strategy} = Trader,
    case check_offer_exists(Offers) of
        true ->
            lists:map(fun(Offer) ->
                {_,_,Temp} = Offer,
                Res = run_strategy(Strategy,Temp),
                {Res,Buyer,Offer}
            end, Offers);
        false ->
            []
    end.

% get all trader result
run_all_traders(Offers, Traders) ->
    lists:map(fun(Trader) ->
        map_offer(Offers, Trader)
    end, Traders).


% get new Offers
remove_offer(Map_Res,Offers) ->
    case Map_Res of
        [] ->
            Offers;
        [Res|T] ->
            case Res of
                {accept,_,{OfferId,Id,_}} ->
                    removeoffer(OfferId,Id,Offers);
                _ ->
                    remove_offer(T,Offers)
            end
    end.

% get new Map_Res and Accounts,return is {Map_Res,Accounts}
transfer_stock(Map_Res, Accounts) ->
    transfer_stock(Map_Res, Accounts, []).
transfer_stock([], Accounts, NewMap_Res) ->
    {NewMap_Res, Accounts};
transfer_stock(Map_Res, Accounts, NewMap_Res) ->
    case Map_Res of
        [{accept,Buyer,{OfferId,Seller,{Stock,Price}}}|T] ->
            Acc1 = change_ISK(add, Seller, Price, Accounts),
            case change_ISK(remove, Buyer, Price, Acc1) of
                {error,not_enough_ISK} ->
                    NewRes = {reject,Buyer,{OfferId,Seller,
{Stock,Price}}},
                    transfer_stock(T, Accounts, [NewRes|NewMap_Res]);
                Acc2 ->
                    Acc3 = change_stock(add, Buyer, Stock, Acc2),

                    case change_stock(remove, Seller, Stock, Acc3) of
                        {error, aaa} ->
                            NewRes = {reject,Buyer,{OfferId,Seller,
{Stock,Price}}},
                            transfer_stock(T, Accounts,
[NewRes|NewMap_Res]);
                        Acc4 ->
                            NewRes = {accept,Buyer,{OfferId,Seller,
{Stock,Price}}},
                            transfer_stock(T, Acc4,
[NewRes|NewMap_Res])
                    end
            end;
        [{reject,B,O}|T] ->
            NewRes = {reject,B,O},
            transfer_stock(T, Accounts, [NewRes|NewMap_Res])
    end.
```

```erlang
change_ISK(Label, Id, Price, Accounts) ->
    case Accounts of
        [] ->
            [];
        [{{S,Id}, {ISK, Holdings}}|T] ->
            case Label of
                add ->
                    [{{S,Id}, {ISK+Price, Holdings}}|T];
                remove ->
                    case ISK-Price < 0 of
                        true ->
                            {error, not_enough_ISK};
                        false ->
                            [{{S,Id}, {ISK-Price, Holdings}}|T]
                    end
            end;
        [H|T] ->
            [H|change_ISK(Label, Id, Price, T)]
    end.

change_stock(_, _, _, []) ->
    [];
% qwqw
change_stock(_,_,_,{error,_})->
    [];
change_stock(Label, Id, Stock, [{{S,Id}, {ISK, Holdings}}|T]) ->
    case Label of
        add ->
            NewHoldings = add_stock(Stock, Holdings),
            [{{S,Id}, {ISK, NewHoldings}}|T];
        remove ->
            case remove_stock(Stock, Holdings) of
                {error, not_enough_stock} ->
                    {error, aaa};
                NewHoldings ->
                    [{{S,Id}, {ISK,NewHoldings}}|T]
            end
    end;
change_stock(Label, Id, Stock, [H|T]) ->
```

```erlang
    case change_stock(Label, Id, Stock, T) of
        {error, aaa} ->
            {error, aaa};
        NewT ->
            [H|NewT]
    end.

remove_stock(Stock, Holdings) ->
    case Holdings of
        [] ->
            {error, not_enough_stock};
        [H|T] ->
            case H of
                {Stock, Amount} ->
                    case Amount of
                        1 ->
                            T;
                        _ ->
                            [{Stock, Amount-1}|T]
                    end;
                _ ->
                    [H|remove_stock(Stock, T)]
            end;
        _ ->
            {error, not_enough_stock}
    end.

add_stock(Stock, Holdings) ->
    case Holdings of
        [] ->
            [{Stock,1}];
        {Stock, Amount} ->
            {Stock, Amount+1};
        {_, _} ->
            [{Stock,1}|Holdings];
        [H|T] ->
            case H of
                {Stock, Amount} ->
                    [{Stock, Amount+1}|T];
                _ ->
                    [H|add_stock(Stock, T)]
            end
    end.

% get final Accounts and Offers
% Map_Reses = [[{accept,AccountId,Offer}...],...]
run_all_Map_Res(Map_Reses,Accounts,Offers,Num) ->
    case Map_Reses of
        [] ->
            {Accounts,Offers,Num};
        [Map_Res|T] ->
            OldNum = length(Offers),
            {NewMap_Res,NewAccounts}=
transfer_stock(Map_Res,Accounts),
            NewOffers = remove_offer([NewMap_Res],Offers),
            NewNum = length(NewOffers),
            Num2 = Num + OldNum - NewNum,
            run_all_Map_Res(T,NewAccounts,NewOffers,Num2)
    end.
```

Test_erlst_unit.erl

```
-module(test_erlst_unit).

-import(erlst, [launch/0, open_account/2, make_offer/2, add_trader/2,
show/1, run_all_traders/2]).
-export([test_all/0, test_everything/0]).
-export([]). % Remember to export the other functions from Q2.2

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

```erlang
test_all() ->
    [
        {"EUnit", spawn, [
            test_start_server(),
            test_shutdown(),
            test_open_account(),
            test_account_balance(),
            test_make_offer(),
            test_rescind_offer(),
            test_add_trader()
        ]}
    ].

test_everything() ->
    eunit:test(test_all(), [verbose]).

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

test_shutdown() ->
  {"Shutdown server",
    fun() ->
      {ok,S} = erlst:launch(),
      ?assertMatch(0, erlst:shutdown(S))
    end}.

test_open_account() ->
    [
        {"Open account acc1 and check balance",
        fun() ->
          {ok, S} = erlst:launch(),
          Acc1 = erlst:open_account(S, {2000, [{"a", 10}]}),
          B = erlst:account_balance(Acc1),
          ?assertMatch({2000,[{"a",10}]} , B)
        end},
        {"Open acc with Holdings and check balance", fun() ->
```

```
         {ok, S} = erlst:launch(),
         Holdings = {100, [{"a", 10}, {"b", 20}]},
         Acc1 = erlst:open_account(S, Holdings),
         B = erlst:account_balance(Acc1),
         ?assertMatch(Holdings , B)
         end},
       {"Open multipal accs", fun() ->
         {ok, S} = erlst:launch(),
         Holdings = {100, [{"a", 10}, {"b", 20}]},
         Acc1 = erlst:open_account(S, Holdings),
         Acc2 = erlst:open_account(S, Holdings),
         Acc3 = erlst:open_account(S, Holdings),
         B1 = erlst:account_balance(Acc1),
         B2 = erlst:account_balance(Acc2),
         B3 = erlst:account_balance(Acc3),
         erlst:show(S),
         ?assertMatch(Holdings , B1),
         ?assertMatch(Holdings , B2),
         ?assertMatch(Holdings , B3)
         end},
       {"Check negative ISK", fun() ->
           {ok, S} = erlst:launch(),
           Holdings = {-100, [{"a", 10}, {"b", 20}]},
           ?assertMatch({error, badHoldings}, erlst:open_account(S,
Holdings))
       end},
       {"Check negative num", fun() ->
           {ok, S} = erlst:launch(),
           Holdings = {100, [{"a", -10}, {"b", 20}]},
           ?assertMatch({error, badHoldings}, erlst:open_account(S,
Holdings))
       end}
     ].

test_account_balance() ->
    {"Check balance of absent user", fun() ->
      erlst:launch(),
      B = erlst:account_balance({{1,2}}),
      ?assertMatch({error,badAcct} , B)
    end}.

test_make_offer() ->
  [{"Check make offer",
    fun() ->
      {ok, S} = erlst:launch(),
      Acc2 = erlst:open_account(S, {2000, []}),
      ?assertMatch({ok,_}, erlst:make_offer(Acc2, {"a", 50}))
    end
    },
    { "Check make offer with negative num",
      fun() ->
        {ok, S} = erlst:launch(),
        Acc2 = erlst:open_account(S, {2000, []}),
        ?assertMatch({error, badPrice}, erlst:make_offer(Acc2, {"a",
-50}))
      end
    },
    { "Check make offer with absent user",
      fun() ->
        ?assertMatch({error, badAcct}, erlst:make_offer({{1,2}},
{"a", 50}))
      end
    },
    {"Check make offer with not own stock",
      fun() ->
        {ok, S} = erlst:launch(),
        Acc2 = erlst:open_account(S, {2000, [{"b", 50}]}),
        ?assertMatch({ok, _}, erlst:make_offer(Acc2, {"a", 50}))
      end
    },
    {"Check make offer and reciend it",
      fun() ->
        {ok, S} = erlst:launch(),
        Acc2 = erlst:open_account(S, {2000, [{"a", 50}]}),
        Offer = erlst:make_offer(Acc2, {"a", 50}),
        ?assertMatch({ok, _}, Offer),
        ?assertMatch(ok, erlst:rescind_offer(Acc2, Offer))
      end
```

```
     },
     {"Check make offer and add trader and run it",
       fun() ->
         {ok, S} = erlst:launch(),
         Acc1 = erlst:open_account(S, {2000, [{"a", 50}]}),
         Acc2 = erlst:open_account(S, {2000, [{"a", 50}]}),
         Strategy = fun({"a", Price}) ->
             if
                 Price < 100 -> accept;
                 true -> reject
             end
         end,
         erlst:add_trader(Acc1, Strategy),
         Offer = erlst:make_offer(Acc2, {"a", 50}),
         ?assertMatch({ok, _}, Offer),
         timer:sleep(1000),
         ?assertMatch({1950, [{"a", 51}]},
erlst:account_balance(Acc1)),
         ?assertMatch({2050, [{"a", 49}]},
erlst:account_balance(Acc2))
       end
     }].

test_add_trader() ->
   {"Add trader and check trader",
     fun() ->
       {ok, S} = erlst:launch(),
       Acc3 = erlst:open_account(S, {2000, [{"a", 2}]}),
       Acc4 = erlst:open_account(S, {1000, []}),
       Strategy = fun({"a", Price}) ->
             if
                 Price < 100 -> accept;
                 true -> reject
             end
         end,
       erlst:make_offer(Acc3, {"a", 50}),
       erlst:add_trader(Acc4, Strategy),
       timer:sleep(1000),
       ?assertMatch({2050,[{"a",1}]},erlst:account_balance(Acc3)),
       ?assertMatch({950,[{"a",1}]} ,erlst:account_balance(Acc4))
     end
     }.

 test_rescind_offer() ->
   {"Rescind offer",
     fun() ->
       {ok, S} = erlst:launch(),
       Acc2 = erlst:open_account(S, {2000, [{"a", 50}]}),
       Offer = erlst:make_offer(Acc2, {"a", 50}),
       ?assertMatch({ok, _}, Offer),
       ?assertMatch(ok, erlst:rescind_offer(Acc2, Offer))
     end
     }.
```

Test_erlst.erl

```
-module(test_erlst).

-export([test_all/0, test_everything/0]).
% -import(test_erlst_unit, [test_all/0, test_everything/0]).
% Remember to export the other functions from Q2.2
-export([prop_value_preservation/0, mkstrategy/1,
reliable_strategy/0,prop_total_trades/0]).

% -compile(export_all).

-include_lib("eqc/include/eqc.hrl").

test_all() ->
    eqc:quickcheck(prop_value_preservation()),
    eqc:quickcheck(prop_total_trades()),
    test_erlst_unit:test_everything().

test_everything() ->
    test_all().

mkstrategy(Opr) ->
```

```erlang
    case Opr of
        buy_everything ->
            fun({_, _}) -> accept end;
        {buy_cheap, Num} ->
            fun({_, Price}) ->
                if
                    Price =< Num -> accept;
                    true -> reject
                end
            end;
        {buy_only, Stocks} ->
            fun({Stock, _}) ->
                case lists:member(Stock, Stocks) of
                    true -> accept;
                    false -> reject
                end
            end;
        {both, S1, S2} ->
            fun(Offer) ->
                Str1 = mkstrategy(S1),
                Str2 = mkstrategy(S2),
                case {Str1(Offer), Str2(Offer)} of
                    {accept, accept} -> accept;
                    {reject, reject} -> reject;
                    _ -> reject
                end
            end
    end.

stock_name() -> eqc_gen:oneof([a,b,c,d,e,f]).
stock_names() -> eqc_gen:list(stock_name()).

stock_list() ->
    ?SUCHTHAT({_,Price},
        {stock_name(), eqc_gen:int()},
        Price > 0).

holdings() ->
    ?SUCHTHAT({Money, _},
               {eqc_gen:int(), stock_list()},
               Money > 0).

offer() ->
    ?SUCHTHAT({_,Price},
          {stock_name(), eqc_gen:int()},
          Price >= 0).

reliable_strategy() ->
    oneof([
            {call, test_erlst, mkstrategy, [buy_everything]},
            {call, test_erlst, mkstrategy, [{buy_cheap, int()}]},
            {call, test_erlst, mkstrategy, [{buy_only,
stock_names()}]},
            {call, test_erlst, mkstrategy, [{both,{buy_cheap,int()},
{buy_only, stock_names()}}]}
    ]).


calc_Total_Money(Accts) ->
    case Accts of
        [] -> 0;
        [{Balance,_}|T] -> case Balance+0 of
            0 -> calc_Total_Money(T);
            _ -> Balance+calc_Total_Money(T)
        end
    end.

calc_Total_Stock(Accts) ->
    case Accts of
        [] -> 0;
        {_,Holdings} -> Holdings;
        [{_,Holdings}|T] -> case Holdings of
            [] -> calc_Total_Stock(T);
            [{_,Num}|T2] -> Num+calc_Total_Stock(T);
            {_,Num} -> Num
        end
    end.
```

```
prop_value_preservation() ->
    ?FORALL(
        {Strategy, Holdings1,Holdings2,Offer},
        {reliable_strategy(), holdings(),holdings(), offer()},
        begin
            {ok,S} = erlst:launch(),
            A1 = erlst:open_account(S, Holdings1),
            A2 = erlst:open_account(S, Holdings2),
            erlst:make_offer(A1, Offer),
            erlst:make_offer(A1, Offer),
            erlst:make_offer(A1, Offer),
            erlst:make_offer(A1, Offer),
            erlst:make_offer(A2, Offer),
            erlst:make_offer(A2, Offer),
            erlst:make_offer(A2, Offer),
            erlst:make_offer(A2, Offer),
            A1Holding = erlst:account_balance(A1),
            A2Holding = erlst:account_balance(A2),
            Money1 = calc_Total_Money([A1Holding, A2Holding]),
            Stock1 = calc_Total_Stock([A1Holding, A2Holding]),
            erlst:add_trader(A1, eval(Strategy)),
            erlst:add_trader(A1, eval(Strategy)),
            erlst:add_trader(A1, eval(Strategy)),
            erlst:add_trader(A1, eval(Strategy)),
            erlst:add_trader(A2, eval(Strategy)),
            erlst:add_trader(A2, eval(Strategy)),
            erlst:add_trader(A2, eval(Strategy)),
            erlst:add_trader(A2, eval(Strategy)),
            A1Holding1 = erlst:account_balance(A1),
            A2Holding1 = erlst:account_balance(A2),
            Money2 = calc_Total_Money([A1Holding1, A2Holding1]),
            Stock2 = calc_Total_Stock([A1Holding1, A2Holding1]),
            eqc:equals(Money1, Money2),
            eqc:equals(Stock1, Stock2)
        end
    ).

prop_total_trades() ->
    ?FORALL(
        {Strategy, Holdings1,Holdings2, Offer},
        {mkstrategy(buy_everything), holdings(), holdings(),
offer()},
        begin
            {ok,Pid} = erlst:launch(),
            A1 = erlst:open_account(Pid, Holdings1),
            A2 = erlst:open_account(Pid, Holdings2),
            erlst:make_offer(A1, Offer),
            erlst:make_offer(A1, Offer),
            erlst:make_offer(A1, Offer),
            erlst:make_offer(A1, Offer),
            erlst:make_offer(A2, Offer),
            erlst:make_offer(A2, Offer),
            erlst:make_offer(A2, Offer),
            erlst:make_offer(A2, Offer),
            erlst:add_trader(A2, Strategy),
            erlst:add_trader(A2, Strategy),
            erlst:add_trader(A2, Strategy),
            erlst:add_trader(A2, Strategy),
            erlst:add_trader(A1, Strategy),
            erlst:add_trader(A1, Strategy),
            erlst:add_trader(A1, Strategy),
            erlst:add_trader(A1, Strategy),
            Num = erlst:shutdown(Pid),
            Num =< 8
        end
    ).
```