Re Exam

Advanced Programming

Exam ID: 25 Feb 3, 2023

Question 1: APREL: A regular-expression library

Question 1.1: Parsing regexes

Design

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

```
RE ::= RESeq| RESeq ‘|’ RE| RESeq ‘&’ RE (need further left method)
RESeq ::= REElt | pRESeq'|ϵ
pRESeq' ::= pREElt many1 pREElt
REElt ::= RERep REElt'
REElt' ::= ‘!’ REElt'
RERep ::= REAtom| REAtom ‘{’ Count ‘}’| REAtom ‘?’| REAtom ‘*’| REAtom ‘+’
REAtom ::= RChar| Class| ‘\’ Number| ‘(’ RE ‘)’| ‘(’ ‘#’ RE ‘)’
Count ::= ::= Number| Number ‘,’| Number ‘,’ Number
Class ::= ‘[’ ClassItemz ‘]’| ‘[’ ‘^’ ClassItemz ‘]’| ‘.’
ClassItemz ::= ϵ | ClassItem ClassItemz
ClassItem ::= CChar | CChar ‘-’ CChar
RChar ::= [any character except the following 13: “!#&()*+.?[\{|”] | EscChar
CChar ::= [any character except the following 4: “-\]^”] | EscChar
EscChar ::= ‘\’ [except“0···9A···Za···z”]| ‘\’‘n’| ‘\’‘t’
Number ::= Digit | Number Digit
```

Implement

I implemented the parsers from bottom to top, follow the grammar and made some necessary changes and write them.

- Basic Design Start with basic part of Number, i implement it in a greedy way, deleting the digit node and parse as much digit as possible to construct an integer. And `CChar` and `RChar` are implemented by `notElem` function. `ClassItemz` are implemented by combining the result from `ClassItem` with `ClassItemz`, when calculating the class we check the sequence of start and end number, if the sequence is wrong, then we return a empty list. For the atom part, i take the ref and re part aside, to parse these two as bias sequence. Parsing repetition and count is the same as the grammar.
- The interesting part of parser is the Define of `RE` and `RSeq`. As conjunction and alternation are left associate, we read them in advance to see how can they get combined. If i read `&`, then we can combine the two first RE with conj. If the sequence is `|` then `&` i read one more RE and combine them(leave the first char to another iteration) the implementation is in `pConjuct` and `pAlt`. In this way i make the expression left associate, also at the same time make conjunction more tighter than alternatives, which i failed initially using simple bias operator.

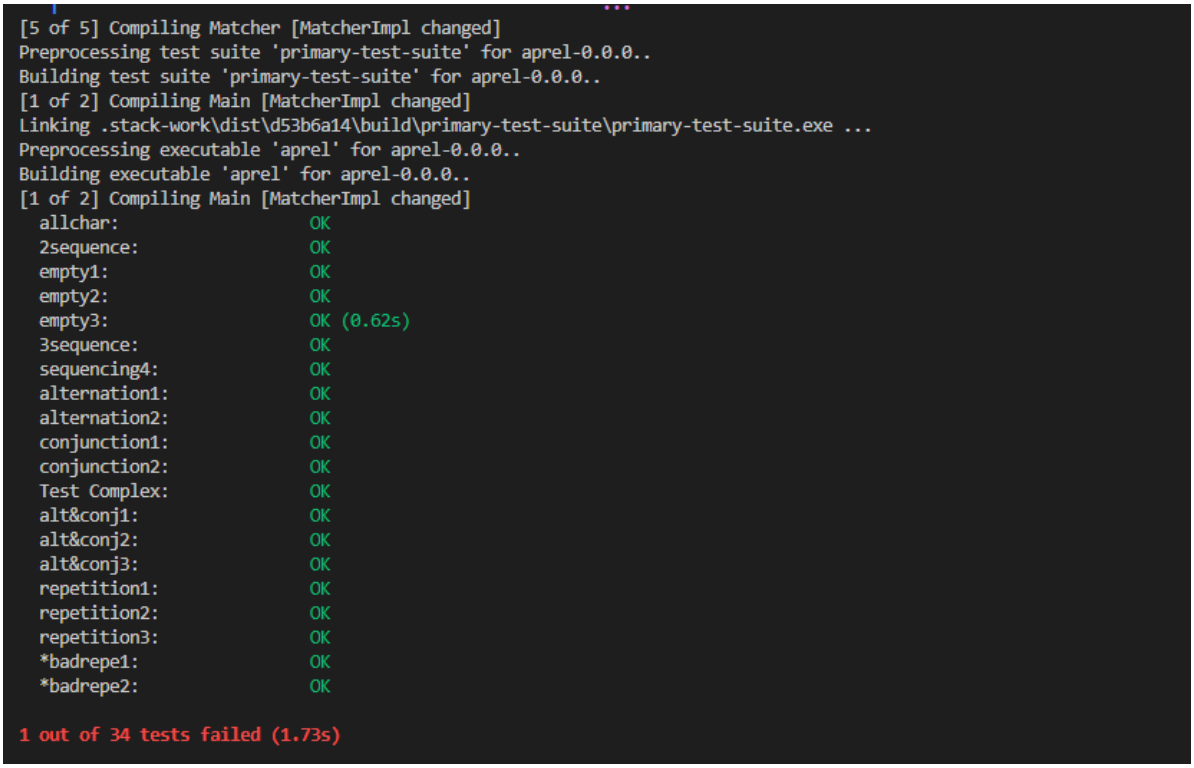
  Then is the solve of empty brackets problem, i use a little hack here, before each `pElt` i consume as much () as possible, so is redundant `(())`, we pick them in a greedy way, so we remove the influence of empty brackets, when reading `backRef` we check if there is an empty brackets, then it is used as a split symbol to fulfill the request of parsing `\1()0`.

Question 1.2: Matching APREL regexes

I managed to implement some of the functions and MatchRE and MatchTop functions like in monad exercise assignment, which can be checked in the appendix, I'm not sure how much of it can work. I used pattern matching to match some regex strings and take strings from the given string, it succeed in matching the second example `"bba"` (could be an accident).

I made 34 unit tests and passing 33 of it mainly passing the parser part. Including simple text and complicated sentences.

The result is as follow:



Assessment

I have complete most part of the parser, some problems including complex empty bracket like `(((((((())))))))` may not be solved.

Question 2: Sneaky Plans

First of all i want to declare the default time out is 1500 thus 1.5 seconds, i know it may be a bit less than normal, however as most simple success method are done in far more less than 1 second, so i think it is a fair timeout value.

Assessment

- **Completeness:** I managed to support all kinds of keikakus, i made the assumption for timeout, as the solve the possible influence of trap, also for simple trap it works as an exception and will cause error.
- **Robustness**: the robustness of this system is good because most of the function are done in processes and will not affect the running server, others function are surrounded by try catch clause, so error will seldom occur. Some extra message are sent by doesn't killed process which may case some error information but won't cause big failure.
- **Maintainability**: the maintainability of the system is good, as most help functions are sub functions so is easy to send, also the name of the function can make people easily conduct the effect of the function. So does the gen_server OPT's API.
- I would split process topic into several operations.
    - **rudiment** this will open a single process as for possible forever loop, i made a after sentence and exit after get the value.
        - ```
          {rudiment, Cmd, Arg} ->
                      Fun = find_func(Cmd,Func),
                      Me = self(),
                      Lid = spawn_link(fun() ->
          listen_to_single(Me,timeout)end),
                      spawn(fun() ->
                          Res = Fun(Arg),
                          Lid ! {success,Res},
                          exit(normal)
                      end),
                      receive
                          {success, R} ->
                              % io:format("Rudiment end: ~p~n",[R]),
                              {success, R};
                          {failure, E} ->
                              % io:format("Rudiment end: ~p~n",[E]),
                              {failure, E}
                      end;
          ```
    - **trap** similar single process, but to simulate it, i made an assumption and this is not achieved by open up process and after timer sleep return failure. The time is calculated by ten times of the input parameter, for the timer in erlang calculate time in 1/thousand second.
    - **progression** create single process one by one and have after clause to catch timeout.
    - **race&side by side** I create a listening process and bunch of concurrent processes for each sub plan, could as follow implementation and for side by side i get the `pid` by `lists:maps` and do further process:
          ```

```
    Pid = spawn(fun() ->
            wait_for_race(--Main process--)%receive sub plan message and
send message
        end),
    Pids = lists:map(fun(El) ->
                    spawn(fun())
                    end,
                    SubKeikakus),
    receive message send by wait for race and process result
```

  - **feint&relay** single process, feint just take the opposite of its plan result, and relay is surrounded by try catch clause, if exception happened, it will return `{failure, not_according_to_keikaku }`.
  - **Report** is just get the operations result so single process.
  - **Ambush** is done with cast so asynchronous, if the progress is ongoing then save the ambush to state, when one operation is done, it checks the state, if there is ambush exists, it will calculate it.

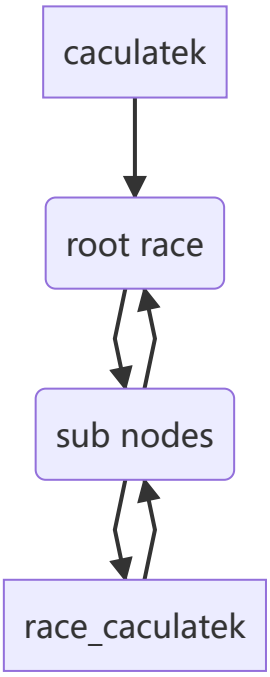
- **Process lifetime** Process is ended if it finished one operation by exit(normal), i did not kill forever loop process, and i didn't test leak process, I'm not sure if i can get the running processes. Further more as i may not have enough time to solve it, as far as i know that `erlang` will kill the process after finite loops, this part will be left for further modification.

- ## Design

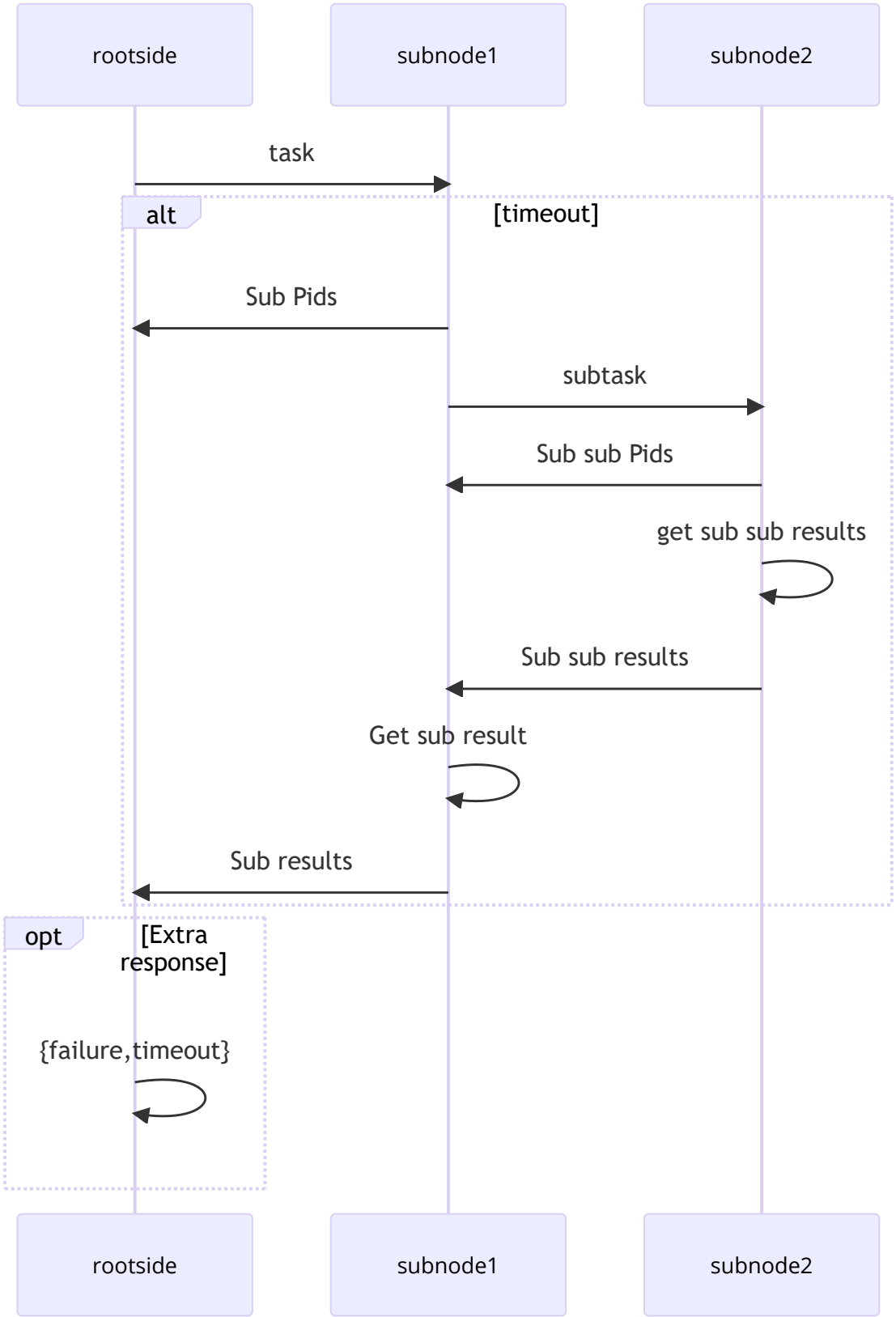
- Yes! All APIs are supported, due to time limit, i do not test the ambush function, so i just use the minimal test to test it, i use `gen_server` to implements these APIs. The state is

```
Cmd functionlist -> [{Cmd,Function}],
Operation list -> [{{Serverid,OperationId},{Status,Value}}]
Ambush list -> [{OperationId,Function}](Ambush)
```

  - Prepare and report function are easy as `init` and get operations' result from the state.
  - **Operation**, the operation will do one call for adding the operation to state, and one cast for calculating the result.
  - **rudiment, trap, feint** are similar to state above.
  - **Race** To achieve race i managed to solve it by two function one for the first race `calculateK` and one for the sub nodes `sub_calculateK`, the second function take one more parameter `pid` which is the listener of the main race node, when there is sub node of sub node, its result will be returned as {Mid,{status,value}} Mid is the id of the sub sub nodes, and the root receive such kind of information by iteratively receive failure info and stop when received success result. The process diagram is similar to below.



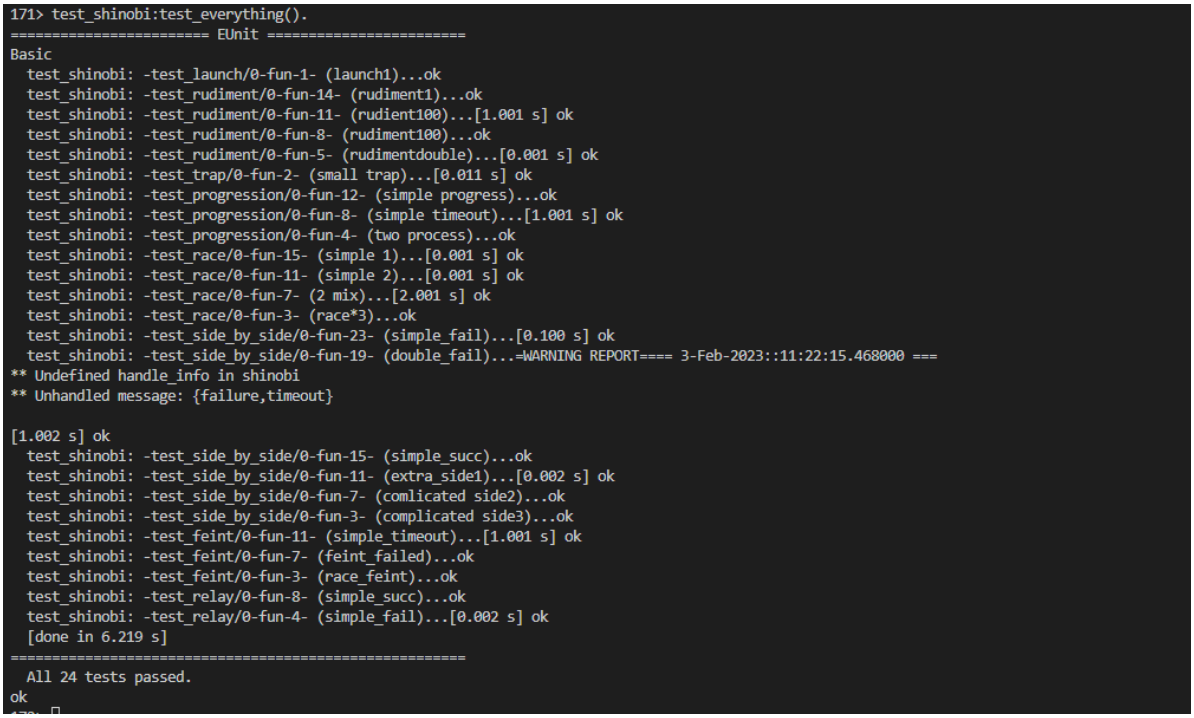
- **Side by side** is more complicated, as it needs all the sub nodes to be succeed. To achieve this, i simulate what i do in race, additionally we will record all the results of sub nodes by its length of `ResultList` needed, when the `PList` returned by `lists:`, when we go to the final step, we need to pick the `pid` and value one by one and add them to the final list. When we get the list or sub lists done, we need to reverse `lists:reverse` it as our combining order it different from the original order. The complex part of it is to figure out the relationship between sub nodes and root nodes as it needs all the result as success or simply one failure, the sequence of operations can be as below, different from race which only need one sub nodes to be in success status:



- **relay** for relay function, i include it in a try catch `_:_` block and for exception just report the `not_according_to_failure`.
- **Ambush** First, ambush check the operation state, if it is not finished, then we save it to the list, when a operation finished, it will scan the state information, if there is a function apply to its `OperationId`, It will run the function and delete the Ambush.

Test

I implemented 24 unit tests and passing all of them, due to the implement way of process, that some message may get unhandled, especially when we need to get all the results in side by side implementation. (Revision, one more timeout test added, total 25 unit tests and passed all).



Appendix

`ParserImpl`

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

import Data.Char
import Text.ParserCombinators.ReadP
import Control.Applicative ((<|>))
import qualified Data.Set as S
import AST
-- import ReadP or Parsec, as relevant

max1 :: Int
max1 = maxBound
allLetterDigit = ['a'..'z']++['A'..'Z']++['0'..'9']
reserved13 = "!#&()*+.?[\\{|"
reserved4 = "-\\]^"
reserved62 = ["0···9A···Za···z"]
-- Do not change the type!
parseRE :: String -> Either String RE
parseRE str = case readP_to_S (do
    result <- pRE
    eof
    return result) str of
    [] -> Left "Parse error"
    x -> case last x of
      (x, "") -> return x
      _ -> Left "Parse error"



-- RE ::= RESeq| conjuct
pRE :: ReadP RE
pRE = pRESeq
    <|> do
    a <- pRESeq
    pConjuct a


pConjuct :: RE -> ReadP RE
pConjuct a = do
    char '&'
    b<- pRESeq
    pConjuct (RConj a b)
    <|> do
    char '|'
    b <- pRESeq
    pAlt a b
    <++ do
        return a

pAlt :: RE -> RE -> ReadP RE
```

```
pAlt a b = do
    char '|'
    c <- pRESeq
    pAlt (RAlt a b) c
    <|> do
    char '&'
    c <- pRESeq
    pAlt a (RConj b c)
    <++ do
        return $ RAlt a b

-- RESeq ::= pREElt RESeq' RESeq''| RESeq' RESeq''|  RESeq' RESeq''
pRESeq :: ReadP RE
pRESeq = do
    a <- pREElt
    return a
    <|> do
    -- a <- pREElt
    b <- pRESeq'
    return (RSeq b)
    <|> return (RSeq [])

-- RESeq' = pREElt | em
pRESeq' :: ReadP [RE]
pRESeq' = do
    a <- pREElt
    b <- many1 pREElt
    return $ a:b

-- REElt ::= RERep Elt'
pREElt :: ReadP RE
pREElt = do
    many skipEmpty
    r <- pRERep
    s <- pElt'
    return $ dealNeg s r

dealNeg :: [Char] -> RE -> RE
dealNeg [] a = a
dealNeg [x] a = RNeg a
dealNeg (x:xs) a = dealNeg xs (RNeg a)

-- Elt' = ϵ | '!' Elt'
pElt' :: ReadP [Char]
pElt' = do
    a <- munch (\x -> x == '!')
    return $ a
    <++ return []

-- RERep ::= REAtom| REAtom '{' Count '}'| REAtom '?'| REAtom '*'| REAtom '+'
pRERep :: ReadP RE
pRERep = do
    a <- pREAtom
    char '{'
    b <- pCount
```

```
    char '}'
    return (RRepeat a b)
    <|> do
    a <- pREAtom
    char '?'
    return (RRepeat a (0,1))
    <|> do
    a <- pREAtom
    char '*'
    return (RRepeat a (0,max1))
    <|> do
    a <- pREAtom
    char '+'
    return (RRepeat a (1,max1))
    <|> pREAtom


-- REAtom ::= RChar| Class| '\' Number| pREAtom'
pREAtom :: ReadP RE
pREAtom = pRChar
    <|> pClass
    <|> pRef
    <|> pREAtom'

pRef = do
    string "\\"
    n <- pNumber
-- help define the back ref and simply number
    string "()"
    return $ RBackref n
    <++ do
    string "\\"
    n <- pNumber
    return $ RBackref n
-- pREAtom'    '(' RE ')'| '(' '#' RE ')'
pREAtom' :: ReadP RE
pREAtom' = do
    string "(#"
    n <- pRE
    char ')'
    return (RCapture n)
    <++ do
    char '('
    n <- pRE
    char ')'
    return n

-- Count ::= ::= Number| Number ','| Number ',' Number
pCount :: ReadP (Int,Int)
pCount = do
    a <- pNumber
    char ','
    b <- pNumber
    case a > b of
        True ->
```

```
            pfail
        False ->
            return ((a,b))
    <++ do
    a <- pNumber
    char ','
    return ((a,max1))
    <++ do
    a <- pNumber
    return ((a,a))


-- Class ::= '[' ClassItemz ']'| '[' '^' ClassItemz ']'| '.'
pClass :: ReadP RE
pClass = do
    char '['
    char '^'
    x <- pClassItemz
    char ']'
    return $ RClass True (S.fromList x)
    <|> do
    char '['
    x <- pClassItemz
    char ']'
    return $ RClass False (S.fromList x)
    <|> do
    char '.'
    return $ RClass True (S.fromList [])

-- ClassItemz ::= ϵ | ClassItem ClassItemz
pClassItemz :: ReadP [Char]
pClassItemz = do
    c <- pClassItem
    ds <- pClassItemz
    return $ c++ds
    <|> return []

-- ClassItem ::= CChar | CChar '-' CChar
pClassItem :: ReadP [Char]
pClassItem = do
    x <- pCChar
    char '-'
    y <- pCChar
    if x < y
        then return [x..y]
        else return []
    <|> do
    x <- pCChar
    return [x]

-- RChar ::= [any character except the following 13: "!#&()*+.?[\{|"] | EscChar
pRChar :: ReadP RE
pRChar = do
    x <- satisfy (`notElem` reserved13)
    return (RClass False (S.singleton x))
```

```
    <|> do
    a <- pEscChar
    return $ (RClass False (S.singleton a))

-- CChar ::= [any character except the following 4: “-\]^”] | EscChar
pCChar :: ReadP Char
pCChar = do
    x <- satisfy (`notElem` reserved4)
    return x
    <|> pEscChar

-- EscChar ::= ‘\’ [any character except the following 62: “0···9A···Za···z”]|
‘\’ ‘n’| ‘\’ ‘t’
pEscChar :: ReadP Char
pEscChar = do
    string "\n"
    return '\n'
    <++ do
    string "\t"
    return '\t'
    <++ do
    char '\\'
    s <- satisfy(`notElem` allLetterDigit)
    return $ s


-- Number ::= Digit Number'
-- | empty
pNumber :: ReadP Int
pNumber = do
    x <- munch1 isDigit
    return $ read x

-- helper function to delete single empty bracket
skipEmpty :: ReadP ()
skipEmpty = do
    a <- many1 (char '(')
    b <- many1 (char ')')
    return ()
    <++ do
    char '('
    char ')'
    return ()
```

`MatcherImpl`

```
-- Put your matcher implementation in this file
module MatcherImpl where

import qualified Data.Set as S
import AST
import Control.Monad

-- Generic string-matching monad. Do not change anything in
-- the following definitions.
```

```haskell
newtype Matcher d a =
  Matcher {runMatcher :: String -> Int -> d -> [(a, Int, d)]}

instance Monad (Matcher d) where
  return a = Matcher (\_s _i d -> return (a, 0, d))
  m >>= f =
    Matcher (\s i d -> do (a, j, d') <- runMatcher m s i d
                         (b, j', d'') <- runMatcher (f a) s (i + j) d'
                         return (b, j + j', d''))

instance Functor (Matcher d) where fmap = liftM
instance Applicative (Matcher d) where pure = return; (<*>) = ap

-- Associated operations to implement. Their definitions may freely use
-- the Matcher term constructor. Do not change the types!

nextChar :: Matcher d Char
nextChar = Matcher(\s _ d -> case s of
                              [] -> []
                              (x:_) -> [(x, 1, d)])

getData :: Matcher d d
getData = Matcher(\_ _ d -> [(d,0,d)])

putData :: d -> Matcher d ()
putData d'= Matcher(\_ _ _ -> [((), 0, d')])

mfail :: Matcher d a
mfail = Matcher(\_ _ _ -> [])


pick :: [Matcher d a] -> Matcher d a
pick [] = mfail
pick xs = Matcher(\s i d -> case xs of
                            (c:cs) -> case runMatcher c s i d of
                                      [] -> runMatcher (pick cs) s i d
                                      (a,j,d'):_ -> [(a,j,d')])

grab :: Matcher d a -> Matcher d (String, a)
grab m = Matcher(\s i d -> case runMatcher m s i d of
                           [] -> []
                           [(a,j,d')] -> [((take j s, a), j, d')]) -- take j s
is the string m consumed

both :: Matcher d a -> (a -> Matcher d b) -> Matcher d b
both m1 f = Matcher(\s i d -> case runMatcher m1 s i d of
                              [] -> []
                              (a,j,d'):_ -> runMatcher (f a) s (i+j) d')

-- neg m succeeds whenever m fails
neg :: Matcher d a -> Matcher d ()
neg m = Matcher(\s i d -> case runMatcher m s i d of
                          [] -> [((),0,d)]
                          _ -> [])
```

```haskell
type Captures = [String]

matchRE :: RE -> Matcher Captures ()
matchRE (RClass b s) = do c <- nextChar
                          if (b && c `elem` s) || (not b && c `notElem` s)
                            then mfail
                            else return ()
matchRE (RSeq []) = return ()
matchRE (RSeq (x:xs)) = do matchRE x
                           matchRE (RSeq xs)
matchRE (RAlt r1 r2) = do pick [matchRE r1, matchRE r2]
matchRE (RConj r1 r2) = do both (matchRE r1) (\_ -> matchRE r2)
matchRE (RCapture r) = do (s, _) <- grab (matchRE r)
                          d <- getData
                          putData (s:d)
matchRE (RNeg r) = do neg (matchRE r)
matchRE (RBackref _) = undefined
-- replicate :: Int -> a -> [a]
matchRE (RRepeat r (a,b)) = do matchRE (RSeq (replicate a r))
                               matchRE (RSeq (replicate b (RAlt r (RSeq []))))
matchTop :: RE -> String -> Maybe Captures
matchTop r s = case runMatcher (matchRE r) s 0 [] of
                 [] -> Nothing
                 ((),_,x):_ -> Just x
```

Haskell test

```haskell
-- This is a suggested skeleton for your main black-box tests. You are not
-- required to use Tasty, but be sure that your test suite can be build
-- and run against any implementation of the APREL APIs.

import AST
import Parser
import Matcher
-- Do not import from the XXXImpl modules here!

import Test.Tasty
import Test.Tasty.HUnit
import qualified Data.Set as S

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

tests :: TestTree
tests = rudimentary -- replace this

testCaseBad :: Show a => String -> Either String a -> TestTree
testCaseBad s t =
  testCase ("*" ++ s) $
    case t of
      Left e -> return ()
      Right a -> assertFailure $ "Unexpected success: " ++ show a
```

```
rudimentary :: TestTree
rudimentary =
  testGroup "Rudimentary tests"
    [testCase "parse1" $
       parseRE "a(#b*)" @?= Right re1,
     testCaseBad "parse2" $
       parseRE "(#*b)a",
     testCase "match1" $
       matchTop re1 "abb" @?= Just ["bb"],
     testCase "*match2" $
       matchTop re1 "bba" @?= Nothing,
     testCase "Test Single Char1" $ parseRE "a" @?= Right re2,
     testCase "Test Single Char1'" $ parseRE "a" @?= parseRE "(a)",
     testCase "Test Single class1" $ parseRE "[^a-z0-2]" @?= Right re3,
     testCase "Test Single class2" $ parseRE "." @?= Right re4,
     testCase "Test Single escape number" $ parseRE str5 @?= Right re5,
     testCase "Test Seq escape number" $ parseRE "\n\t" @?= Right re6,
     testCase "Test Single capture" $ parseRE str7 @?= Right re7,
     testCase "Test Single neg" $ parseRE "a!!!!!" @?= Right re8,
     testCase "lowerlehigher" $ parseRE "[9-0]" @?= Right re9,
     testCase "wrongclass" $ parseRE "[z-a]" @?= Right re27,
     testCase "allchar" $ parseRE "." @?= Right re10,
     testCase "2sequence" $ parseRE "ap" @?= Right re11,
     testCase "empty1" $ parseRE "" @?= Right re12,
     testCase "empty2" $ parseRE "()" @?= Right re12,
     testCase "empty3" $ parseRE "((()))" @?= Right re12,
     testCase "3sequence" $ parseRE "a[0-9]b" @?= Right re13,
     testCase "sequencing4" $ parseRE "[^a-z][a-z]" @?= Right re14,
     testCase "alternation1" $ parseRE "a|bb" @?= Right re16,
     testCase "alternation2" $ parseRE "a|b|c" @?= parseRE "(a|b)|c",
     testCase "conjunction1" $ parseRE "a&[a-z]" @?= Right re18,
     testCase "conjunction2" $ parseRE "[abcd]&[ab]&[bc]" @?= parseRE "([abcd]&
[ab])&[bc]",
     testCase "Test Complex" $ parseRE "(#a)b{1,2}c+[^A-Z]\\1" @?= Right re20,
     testCase "alt&conj1" $ parseRE "a|b|c&de" @?= parseRE "a|b|(c&de)",
     testCase "alt&conj2" $ parseRE "[abc]&[ab]|[a]" @?= parseRE "([abc]&[ab])|
[a]",
     testCase "alt&conj3" $ parseRE "a|b|c&[a-c]" @?= parseRE "a|b|(c&[a-c])",
     testCase "repetition1" $ parseRE "a{2}" @?= Right re24,
     testCase "repetition2" $ parseRE "a{1,4}" @?= Right re25,
     testCase "repetition3" $ parseRE "a{4,}" @?= Right re26,
     testCaseBad "badrepe1" $ parseRE "a?+*" ,
     testCaseBad "badrepe2" $ parseRE "a{10,1}"
     ]
  where
    re1 = RSeq [rChar 'a', RCapture (rStar (rChar 'b'))]
    rChar c = RClass False (S.singleton c)
    rStar r = RRepeat r (0,maxBound)
    --Test Single Char1
    re2 = rChar 'a'
    re3 = RClass True (S.fromList "012abcdefghijklmnopqrstuvwxyz")
    re4 = RClass True (S.fromList "")
    --"Test Single escape number"
    str5 = "\\%"
    re5 = RClass False (S.fromList "%")
```

```
    --"Test Seq escape number"
    re6 = RSeq [RClass False (S.fromList "\n"),RClass False (S.fromList "\t")]
    str7 = "(#a)"
    re7 = RCapture (RClass False (S.fromList "a"))
    re8 = (RNeg (RNeg (RNeg (RNeg (RNeg (RClass False (S.fromList "a")))))))
    re9 = RClass False (S.fromList "")
    re10 = RClass True (S.fromList "")
    re11 = RSeq [RClass False (S.fromList "a"),RClass False (S.fromList "p")]
    re12 = RSeq []
    re13 = RSeq [RClass False (S.fromList "a"),RClass False (S.fromList
"0123456789"),RClass False (S.fromList "b")]
    re14 = RSeq [RClass True (S.fromList "abcdefghijklmnopqrstuvwxyz"),RClass
False (S.fromList "abcdefghijklmnopqrstuvwxyz")]
    re15 = RClass False (S.fromList "()abcde")
    re16 = (RAlt (RClass False (S.fromList "a")) (RSeq [RClass False (S.fromList
"b"),RClass False (S.fromList "b")]))
    re17 = (RAlt (RClass False (S.fromList "a")) (RAlt (RClass False (S.fromList
"b")) (RClass False (S.fromList "c"))))
    re18 = (RConj (RClass False (S.fromList "a")) (RClass False (S.fromList
"abcdefghijklmnopqrstuvwxyz")))
    re19 = RConj (RClass False (S.fromList "abcd")) (RConj (RClass False
(S.fromList "ab")) (RClass False (S.fromList "bc")))
    re20 = RSeq [RCapture (RClass False (S.fromList "a")),RRepeat (RClass False
(S.fromList "b")) (1,2),RRepeat (RClass False (S.fromList "c"))
(1,9223372036854775807),RClass True (S.fromList
"ABCDEFGHIJKLMNOPQRSTUVWXYZ"),RBackref 1]
    re21 = RAlt (RClass False (S.fromList "a")) (RAlt (RClass False (S.fromList
"b")) (RConj (RClass False (S.fromList "c")) (RSeq [RClass False (S.fromList
"d"),RClass False (S.fromList "e")])))
    re23 = RAlt (RClass False (S.fromList "a")) (RAlt (RClass False (S.fromList
"b")) (RConj (RClass False (S.fromList "c")) (RClass False (S.fromList "abc"))))
    re24 = RRepeat (RClass False (S.fromList "a")) (2,2)
    re25 = RRepeat (RClass False (S.fromList "a")) (1,4)
    re26 = RRepeat (RClass False (S.fromList "a")) (4,maxBound)
    re27 = RClass False (S.fromList "")
```

`Shinobi`

```
-module(shinobi).
-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 shinobi.

% Export at least the API:
-export([prepare/0, register_command/3, operation/2, ambush/2, report/1]).

% You may have other exports as well
-export([]).
-export([init/1,terminate/2,handle_call/3,handle_cast/2]).

-type result() :: {success, term()}
                | {failure, term()}
                .

-type keikaku() :: {rudiment, atom(), term()}
                 | {trap, non_neg_integer()}
                 | {progression, nonempty_list(keikaku())}
                 | {race, list(keikaku())}
                 | {side_by_side, list(keikaku())}
                 | {feint, keikaku()}
                 | {relay, keikaku(), fun((result()) -> keikaku())}
                 .

-type operation_status() :: { ongoing, integer() }
                          | { success, term() }
                          | { failure, term() }
                          .

% You may change these types to something more specific, e.g. pid()
-type shinobi() :: term().
-type operation_id() :: term().

%Operation -> {ID,Status,
-spec prepare() -> {ok, Shinobi :: shinobi()} | {error, Error :: any()}.
prepare() ->
    gen_server:start_link(?MODULE, [], []).

-spec register_command(Shinobi :: shinobi(), Cmd :: atom(), Fun :: fun((term())
-> term())) -> ok | {error, already_defined}.
register_command(Shinobi, Cmd, Fun) ->
    gen_server:call(Shinobi, {register, Cmd,Fun}).

-spec operation(shinobi(), keikaku()) -> {ok, operation_id()} | {error, Error ::
any()}.
operation(Shinobi, Keikakus) ->
    Reply = gen_server:call(Shinobi, {operation,Shinobi, Keikakus}),
    case Reply of
        {ok, OperationID} ->
```

```
            gen_server:cast(Shinobi,{oper, OperationID,Keikakus}),
            Res = {ok, OperationID};
        _ ->
            Res = {error, "unknown error with operation"}
        end,
        Res.

-spec ambush(operation_id(), fun((result()) -> any())) -> any().
ambush(OperationId, Fun) ->
    case OperationId of
        {Shinobi, _} ->
            {Shinobi,_} = OperationId,
            gen_server:cast(Shinobi,{ambush, OperationId, Fun});
        _ ->
            {error, bad_operation}
        end.

-spec report(operation_id()) -> operation_status().
report(OperationId) ->
    io:format("report:start"),
    case OperationId of
        {Shinobi, _} ->
            io:format("report request: ~p~n",[[Shinobi,{report, OperationId}]]),
            gen_server:call(Shinobi,{report, OperationId});
        _ ->
            io:format("report error:"),
            {error,badarg}
        end.

%--- server API [{cmd, fun} ...]  [{{Sid,Oid},{status,value}}] [Ambush]
init([]) ->
    State =[[],[],[]],
    {ok, State}.

handle_call({register, Cmd,Fun},_From,[Func,Operation,Am]) ->
    case lists:keyfind(Cmd,1,Func) of
        false ->
            New_Func = [{Cmd,Fun}|Func],
            {reply,{ok},[New_Func,Operation,Am]};
        _ ->
            {reply,{error, already_defined},[Func,Operation,Am]}
    end;

handle_call({operation, Shinobi, Keikakus},_From,State) ->
    [Func,Operation,Ambush] = State,
    case Operation == [] of
        true ->
            OperationID = {Shinobi,1};
        _ ->
            OperationID = {Shinobi,length(Operation)+1}
        end,
    case check_sub(Keikakus) of
        {ok,Ks} ->
            Numsub = get_totalPlan(Ks,0);
        false ->
```

```
            Numsub = 0
        end,
        New_Operation = [{OperationID,{ongoing, Numsub}}|Operation],
        {reply,{ok,OperationID},[Func,New_Operation,Ambush]};

handle_call({report, OperationId},_From,State)->
    [Func,Operations,Ambush] = State,
    {OperationId,{Status,Value}}=lists:keyfind(OperationId,1,Operations),
    {reply,{Status,Value},[Func,Operations,Ambush]}.

handle_cast({ambush, OperationId, Fun},[Func,Operations,Ambush])->
    {_,{Status,Value}} = lists:keyfind(OperationId,1,Operations),
    % io:format("foundvalue: ~p~n",[Value]),
    case Status == ongoing of
        false ->
            New_Ambush=lists:delete({OperationId,Fun},Ambush),
            run_ambush(Status,Value,Fun),
            {noreply,[Func,Operations,New_Ambush]};
        true ->
            NewAmbush = [{OperationId, Fun}|Ambush],
            {noreply,[Func,Operations,NewAmbush]}
        end;

handle_cast({oper, OperationId,Keikaku},State)->
    io:format("cast operation"),
    [Func,Operations,Ambush] = State,
    case calculateK(Keikaku,State) of
        {success,R}->
            New_Operation2 = lists:keyreplace(OperationId,1,Operations,
{OperationId,{success, R}}),
            check_ambush(OperationId,{OperationId,{success, R}},Ambush),
            {noreply,[Func,New_Operation2,Ambush]};
        {failure,Reason} ->
            New_Operation2 = lists:keyreplace(OperationId,1,Operations,
{OperationId,{failure,Reason}}),
            check_ambush(OperationId,{OperationId,{failure,Reason}},Ambush),
            {noreply,[Func,New_Operation2,Ambush]}
    end.

terminate(_Reason, _State) ->
    ok.

%---- helper function
calculateK(Keikaku,State) ->
    [Func,_,_] = State,
    case Keikaku of
        {rudiment, Cmd, Arg} ->
            Fun = find_func(Cmd,Func),
            % io:format("Rudiment start: ~p~n",[Fun]),
            Me = self(),
            Lid = spawn_link(fun() ->
                listen_to_single(Me,timeout)
            end),
            spawn(fun() ->
                Res = Fun(Arg),
```

```
                Lid ! {success,Res},
                exit(normal)
            end),
            receive
                {success, R} ->
                    % io:format("Rudiment end: ~p~n",[R]),
                    {success, R};
                {failure, E} ->
                    % io:format("Rudiment end: ~p~n",[E]),
                    {failure, E}
            end;
        {trap, Limit} ->
            io:format("trap2: ~p~n",[Limit]),
            timer:sleep(Limit*10),
            {failure, timeout};
        {progression, SubKeikakus}->
            case SubKeikakus == [] of
                true ->
                    {failure, "empty plan"};
                _ ->
                    sequencek(SubKeikakus,State,null)
            end;
        {race, SubKeikakus}->
            dealRace(SubKeikakus,State);
        {side_by_side, SubKeikakus} ->
            dealSide(SubKeikakus,State);
        {feint, SubKeikaku}->
            case calculateK(SubKeikaku,State) of
                {success, R} ->
                    {failure, R};
                {failure, E} ->
                    {success, E}
            end;
        {relay, SubKeikaku, FunOperation}->
            case calculateK(SubKeikaku,State) of
                {success, R} ->
                    Res = {succes,R},
                    io:format("relay first operation: ~p~n",[FunOperation]);
                {failure, E} ->
                    Res = {failure, E},
                    io:format("relay first result: ~p~n",[Res])
            end,
            try
                TempResult = FunOperation(Res),
                Result = calculateK(TempResult,State),
                Result
            catch
                _:_ ->
                    {failure, not_according_to_keikaku}
            end
        end.

%helper funtion to find the function {cmd, function}
find_func(Cmd,Func)->
    case lists:keyfind(Cmd,1,Func) of
```

```
        {Cmd,Function} ->
            Function;
        _ ->
            error
        end.

%function for solve progression
sequencek([],_,R)->
    {success, R};
sequencek([Keikakus|Total],State,_)->
    Me = self(),
    Lid = spawn_link(fun() ->
        listen_to_single(Me,timeout)
    end),
    spawn(fun() ->
        Res = calculateK(Keikakus,State),
        Lid ! Res,
        exit(normal)
    end),
    receive
        {success, R1} ->
            sequencek(Total,State,R1);
        {failure, Reason} ->
            {failure, Reason}
    end.

check_sub(Keikaku)->
    case Keikaku of
        {side_by_side,Ks}->
            {ok,Ks};
        {race, Ks}->
            {ok,Ks};
        {progression, Ks}->
            {ok,Ks};
        _ -> false
    end.

%get total plan number
get_totalPlan([],Sum)->Sum;
get_totalPlan([K|Keikakus],Sum)->
    case (K) of
        {rudiment, _,_} -> Sum2 = get_totalPlan(Keikakus,Sum+1);
        {trap,_} -> Sum2 = get_totalPlan(Keikakus,Sum+1);
        {progression, SubKeikakus}->
            Sum2 = get_totalPlan(Keikakus,Sum+1+get_totalPlan(SubKeikakus,0));

        {race, SubKeikakus}  ->
            Sum2 = get_totalPlan(Keikakus,Sum+1+get_totalPlan(SubKeikakus,0));

        {side_by_side, SubKeikakus}  ->
            Sum2 = get_totalPlan(Keikakus,Sum+1+get_totalPlan(SubKeikakus,0));

        {feint, SubKeikaku}  -> Sum2 = get_totalPlan(SubKeikaku,Sum+1);
        {relay, SubKeikaku, _} -> Sum2 = get_totalPlan(SubKeikaku,Sum+1)
        end,
```

```
    Sum2.

%main function to deal race
dealRace(SubKeikakus,State) ->
    Me = self(),
    Pid = spawn(fun() ->
            wait_for_race(Me)
        end),
    lists:map(fun(El) ->
                    spawn(fun() ->
                                race_calculateK(El,State,Pid),
                                exit(normal)
                            end)
                    end,
                    SubKeikakus),
    % wait_for_race(Me),
    receive
        {_,{success, R}} ->
            Res = {success, R};
        {_,{failure, _}} ->
            Res = {failure, []}
    end,
    Res.


wait_for_race(S)->
    receive
        {success, R}->
            Res = {1,{success, R}},
            S ! Res;
        {failure, _} ->
            wait_for_race(S);
        {Pid,{success, R}}->
            S ! {Pid,{success, R}};
        {_,{failure, []}}->
            wait_for_race(S)
    after
        1000 ->
            % 1 just for no meanings
            Res = {1,{failure, []}},
            S ! Res
        end.

dealSide(SubKeikakus,State) ->
    Me = self(),
    NumSub = get_totalPlan(SubKeikakus,0),
    io:format("Side by side NumSub: ~p~n",[NumSub]),
    Pid = spawn(fun() ->
            wait_for_side(Me)
        end),
    Plist = lists:map(fun(Son) ->
                    spawn(fun() ->
                                Result = race_calculateK(Son,State,Pid),
                                Pid ! {Pid,Result},
                                exit(normal)
```

```
                            end)
                    end,
                    SubKeikakus),
    io:format("Plist main: ~p~n",[Plist]),
    solvePlist(Plist,Me,[]),
    receive
        {success, ResultList} ->
            io:format("Receive son's ResultList: ~p~n",[ResultList]),
            {success,get_origin_result(Plist,ResultList)};
        {failure, E} -> {failure, E}
    end.

%wait for side by side answer
wait_for_side(Me)->
    receive
        {_,{failure, E}} ->

            Me ! {failure, E};
        {Pid,{success, R}} ->
            io:format("Pid: ~p~n",[Pid]),
            Me ! {Pid, R},
            wait_for_side(Me)
    end.

listen_to_single(Me,Res) ->
    receive
        {success, R} ->
            % io:format("R1: ~p~n",[R]),
            Me ! {success, R};
        {failure, E} ->
            % io:format("E: ~p~n",[E]),
            Me ! {failure, E}
        after
            1000 ->
                Me ! {failure, Res}
        end.

run_ambush(Status,Value,Fun) ->
     spawn(fun() ->
                Fun({Status,Value}),
                exit(normal)
            end).

check_ambush(OperationId,{OperationId,{Status, Value}},Ambush) ->
    case lists:keyfind(OperationId,1,Ambush) of
        {OperationId, Fun} ->
            run_ambush(Status,Value,Fun),
            New_Ambush = lists:delete({OperationId, Fun},Ambush),
            check_ambush(OperationId,{OperationId,{Status, Value}},New_Ambush);
        false ->
            Ambush
        end.

%helper function to deal with son nodes
race_calculateK(Keikaku,State,Pid) ->
```

```
    Mid = self(),
    [Func,_,_] = State,
    case Keikaku of
        {rudiment, Cmd, Arg} ->
            Fun = find_func(Cmd,Func),
            Lid = spawn_link(fun() ->
                listen_to_single(Mid,timeout)
            end),
            spawn(fun() ->
                Res = Fun(Arg),
                Lid ! {success,Res}
            end),
            receive
                {success, R} ->
                    Result = {success, R};
                {failure, E} ->
                    Result = {failure, E}
            end,
            Pid ! {Mid,Result};
        {trap, Limit} ->
            io:format("trap2: ~p~n",[Limit]),
            timer:sleep(Limit),
            Pid ! {Mid,{failure, timeout}};
        {progression, SubKeikakus}->
            case SubKeikakus == [] of
                true ->
                    Result = {failure, "empty plan"};
                _ ->
                    Null = "null",
                    Result = sequencek(SubKeikakus,State,Null)
            end,
            Pid ! {Mid,Result};
        {race, SubKeikakus}->
            subdealRace(SubKeikakus,State,Pid);
        {side_by_side, SubKeikakus} ->
            subdealSide(SubKeikakus,State,Pid);
        {feint, SubKeikaku}->
            case calculateK(SubKeikaku,State) of
                {success, R} ->
                    Pid ! {Mid,{failure, R}};
                {failure, E} ->
                    Pid ! {Mid,{success, E}}
            end;
        {relay, SubKeikaku, FunOperation}->
            case calculateK(SubKeikaku,State) of
                {success, R} ->
                    Res = {succes,R},
                    io:format("relay first operation: ~p~n",[FunOperation]);
                {failure, E} ->
                    Res = {failure, E},
                    io:format("relay first result: ~p~n",[Res])
            end,
            try
                TempResult = FunOperation(Res),
                Result = calculateK(TempResult,State),
```

```
                Pid ! {Mid,Result}
            catch
                _:_ ->
                    Pid ! {Mid,{failure, not_according_to_keikaku}}
            end
        end.

% son method for race
subdealRace(SubKeikakus,State,Pid) ->
    Me = self(),
    Pid2 = spawn(fun() ->
            wait_for_race(Me)
        end),
    io:format("Race Pid2: ~p~n",[Pid2]),
    lists:map(fun(El) ->
                    spawn(fun() ->
                                race_calculateK(El,State,Pid2)
                            end)
                    end,
                    SubKeikakus),
    receive
        {_,{success, R}} ->
           Pid ! {1,{success, R}};
        {Pid2,{failure, _}} ->
           Pid ! {1,{failure, []}}
    end.

%main function to solve list
solvePlist(Plist,Me,ResultList) ->
        receive
            {failure, ER} ->
                Me ! {failure, ER};
            {Pid,R} ->
                NewResultList = [{Pid, R}|ResultList],
                io:format("NewResultList: ~p~n",[NewResultList]),
                io:format("Plist: ~p~n",[Plist]),
                case is_list(Plist) of
                    true ->
                        case length(NewResultList) == length(Plist) of
                            true ->
                                Me ! {success, NewResultList};
                            false ->
                                solvePlist(Plist,Me,NewResultList)
                            end;
                    false ->
                        case length(NewResultList) == 1 of
                            true ->
                                Me ! {success, NewResultList};
                            false ->
                                solvePlist(Plist,Me,NewResultList)
                            end
                        end
                    after 1000 ->
                        Me ! {failure, timeout}
    end.
```

```
% resultlist = [{Pid, R}]
get_origin_result([],ResultList) ->
    io:format("Final ResultList: ~p~n",[ResultList]),
    Final = lists:reverse(ResultList),
    Final;
get_origin_result([Pid|Plist],ResultList) ->
    Value = lists:keyfind(Pid,1,ResultList),
    % io:format("Original Value found: ~p~n",[Value]),
    case Value == false of
        true ->
            get_origin_result(Plist,[{failure, timeout}|ResultList]);
        false ->
            {Pid, Value1} = Value,
            ResultList2 = lists:delete({Pid, Value1},ResultList),
            get_origin_result(Plist,[Value1|ResultList2])
    end.

% son process of side_by_side
subdealSide(SubKeikakus,State,Pid) ->
    Mid = self(),
    Pid2 = spawn(fun() ->
            wait_for_side(Mid)
        end),
    Plist = lists:map(fun(Son) ->
                    spawn(fun() ->
                                Result = race_calculateK(Son,State,Pid2),
                                % io:format("Son result: ~p~n",[[Result]]),
                                Pid2 ! {Mid ,Result},
                                exit(normal)
                            end)
                    end,
                    SubKeikakus),
    io:format("Plist sub: ~p~n",[Plist]),
    subsolvePlist(Plist,Mid,[]),
    receive
        {success, ResultList} ->
            ResultList2 = get_origin_result(Plist,ResultList),
            io:format("ResultList2: ~p~n",[ResultList2]),
            Pid ! {Mid,{success, ResultList2}};
        {failure, E} -> Pid ! {failure, E}
    end.

%son function to solve list
subsolvePlist(Plist,Me,ResultList) ->
        receive
            {failure, ER} ->
                Me ! {failure, ER};
            {Pid,R} ->
                NewResultList = [{Pid, R}|ResultList],
                io:format("sub_NewResultList: ~p~n",[NewResultList]),
                case is_list(Plist) of
                    true ->
                        case length(NewResultList) == length(Plist) of
                            true ->
```

```
                                Me ! {success, NewResultList};
                            false ->
                                subsolvePlist(Plist,Me,NewResultList)
                            end;
                    false ->
                        io:format("single process: ~n"),
                        case length(NewResultList) == 1 of
                            true ->
                                Me ! {success, NewResultList};
                            false ->
                                subsolvePlist(Plist,Me,NewResultList)
                            end
                        end
                    after 1000 ->
                        Me ! {failure, timeout}
    end.
```

Eunit test

```
-module(test_shinobi).

% You are allowed to split your test 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_shinobi.

-export([test_all/0, test_everything/0]).
-export([test/0]). % You may have other exports as well

-include_lib("eunit/include/eunit.hrl").
test_all() ->
  eunit:test(tests(),[verbose]).

tests() ->
  [{"Basic",spawn,
    [test_launch(),
    test_rudiment(),
    test_trap(),
    test_progression(),
    test_race(),
    test_side_by_side(),
    test_feint(),
    test_relay()
    ]}].

test_everything() ->
  test_all().


test_launch() ->
  [{"launch1", fun() ->
    ?assertMatch({ok, _},shinobi:prepare())
  end}].
```

```
test_rudiment() ->
  [{"rudiment1", fun() ->
    {ok, A} = shinobi:prepare(),
    shinobi:register_command(A, forever, fun Loop(X) -> Loop(X) end),
    Res = shinobi:operation(A, {rudiment, forever, 1}),
    ?assertEqual(Res, {ok, {A,1}})
  end},
  {"rudient100",fun()->
    {ok, A} = shinobi:prepare(),
    shinobi:register_command(A, forever, fun Loop(X) -> Loop(X) end),
    shinobi:operation(A, {rudiment, forever, 1}),
    Result = shinobi:report({A,1}),
    ?assertEqual(Result, {failure, timeout})
    end},
  {"rudiment100",fun()->
    {ok, A} = shinobi:prepare(),
    shinobi:register_command(A, ret, fun return2/1),
    shinobi:operation(A, {rudiment, ret, 1}),
    Result = shinobi:report({A,1}),
    ?assertEqual(Result, {success, 1})
    end},
  {"rudimentdouble",fun()->
    {ok, A} = shinobi:prepare(),
    {ok, B} = shinobi:prepare(),
    shinobi:register_command(A, double, fun return3/1),
    shinobi:register_command(B, double, fun return3/1),
    shinobi:operation(A, {rudiment, double, 4}),
    shinobi:operation(A, {rudiment, double, 8}),
    shinobi:operation(B, {rudiment, double, 32}),
    R1 = shinobi:report({A,1}),
    R2 = shinobi:report({A,2}),
    R3 = shinobi:report({B,1}),
    ?assertEqual(R1, {success, 8}),
    ?assertEqual(R2, {success, 16}),
    ?assertEqual(R3, {success, 64})
    end}].

test_trap() ->
  [{"small trap",fun()->
    {ok, A} = shinobi:prepare(),
    shinobi:register_command(A, forever, fun Loop(X) -> Loop(X) end),
    shinobi:operation(A, {trap, 1}),
    Res = shinobi:report({A,1}),
    ?assertEqual(Res, {failure, timeout})
    end}].

test_progression() ->
  [{"simple progress",
    fun()->
    {ok, A} = shinobi:prepare(),
    shinobi:register_command(A, forever, fun Loop(X) -> Loop(X) end),
    shinobi:register_command(A, ret, fun return2/1),
    Operation = {progression, [{rudiment, ret, 1}, {rudiment, ret, 1}]},
    {ok, OperationID} = shinobi:operation(A, Operation),
    ?assertEqual(shinobi:report(OperationID), {success, 1})
```

```
  end},
  {"simple timeout",fun()->
    {ok, A} = shinobi:prepare(),
    shinobi:register_command(A, forever, fun Loop(X) -> Loop(X) end),
    shinobi:register_command(A, ret, fun return2/1),
    Operation =  {progression, [{rudiment, forever, 1}, {rudiment, ret,
"one"}]},
    {ok, OperationID} = shinobi:operation(A, Operation),
    ?assertEqual(shinobi:report(OperationID),{failure, timeout})
  end},
  {"two process",fun()->
    {ok, A} = shinobi:prepare(),
    {ok, B} = shinobi:prepare(),
    shinobi:register_command(A, ret, fun return2/1),
    shinobi:register_command(B, ret, fun return2/1),
    Operation =  {progression, [{rudiment, ret, 1000}, {rudiment, ret, "one"}]},
    Operation2 =  {progression, [{rudiment, ret, 1000}, {rudiment, ret, 1}]},
    {ok, OperationID} = shinobi:operation(A, Operation),
    {ok, OperationID2} = shinobi:operation(B, Operation2),
    ?assertEqual(shinobi:report(OperationID),{success,"one"}),
    ?assertEqual(shinobi:report(OperationID2),{success, 1})
  end}].


test_race() ->
  [{"simple 1",
    fun() ->
    {ok, A} = shinobi:prepare(),
    shinobi:register_command(A, forever, fun Loop(X) -> Loop(X) end),
    shinobi:register_command(A, ret, fun return2/1),
    Operation = {race, [{trap, 999}, {rudiment, ret, 1}]},
    {ok, OperationID} = shinobi:operation(A, Operation),
    Result = shinobi:report(OperationID),
    ?assertEqual(Result, {success, 1})
  end},
  {"simple 2",
    fun() ->
    {ok, A} = shinobi:prepare(),
    shinobi:register_command(A, forever, fun Loop(X) -> Loop(X) end),
    shinobi:register_command(A, ret, fun return2/1),
    Operation = {race, [{trap, 999}, {race, [{rudiment, ret, 1}, {trap,
999}]}]},
    {ok, OperationID} = shinobi:operation(A, Operation),
    Result = shinobi:report(OperationID),
    ?assertEqual(Result, {success, 1})
  end},
  {"2 mix",
    fun() ->
    {ok, A} = shinobi:prepare(),
    shinobi:register_command(A, forever, fun Loop(X) -> Loop(X) end),
    shinobi:register_command(A, ret, fun return2/1),
    Operation = {race, [{trap, 999}, {side_by_side, [{rudiment, ret, 1}, {trap,
999}]}]},
    {ok, OperationID} = shinobi:operation(A, Operation),
    Result = shinobi:report(OperationID),
```

```
    ?assertEqual(Result, {failure, []})
  end},
  {"race*3",
    fun() ->
    {ok, A} = shinobi:prepare(),
    shinobi:register_command(A, forever, fun Loop(X) -> Loop(X) end),
    shinobi:register_command(A, ret, fun return2/1),
    Operation = {race, [{trap, 999}, {race, [{race, [{rudiment, ret, 100},
{trap, 999}]}, {trap, 999}]}]},
    {ok, OperationID} = shinobi:operation(A, Operation),
    Result = shinobi:report(OperationID),
    ?assertEqual(Result, {success, 100})
  end}].


test_side_by_side() ->
  [{"simple_fail",
    fun() ->
      {ok, Shinobi} = shinobi:prepare(),
      shinobi:register_command(Shinobi, forever, fun Loop(X) -> Loop(X) end),
      shinobi:register_command(Shinobi, ret, fun return2/1),
      {ok, OperationID} = shinobi:operation(Shinobi, mkoperation(6)),
      ?assertEqual(shinobi:report(OperationID),{failure, timeout})
    end},
    {"double_fail",
    fun() ->
      {ok, Shinobi} = shinobi:prepare(),
      shinobi:register_command(Shinobi, forever, fun Loop(X) -> Loop(X) end),
      shinobi:register_command(Shinobi, ret, fun return2/1),
      Operation = {side_by_side, [{side_by_side, [{rudiment, ret,1}, {rudiment,
forever,2}]}, {rudiment, forever,3}]},
      {_, OperationID} = shinobi:operation(Shinobi, Operation),
      ?assertEqual(shinobi:report(OperationID),{failure, timeout})
    end},
    {"simple_succ",
      fun()->
      {ok, Shinobi} = shinobi:prepare(),
      shinobi:register_command(Shinobi, forever, fun Loop(X) -> Loop(X) end),
      shinobi:register_command(Shinobi, ret, fun return2/1),
      Operation = {side_by_side, [{side_by_side, [{rudiment, ret,1}, {rudiment,
ret,2}]}, {rudiment, ret,3}]},
      {ok, OperationID} = shinobi:operation(Shinobi, Operation),
      ?assertEqual(shinobi:report(OperationID) , {success, [[1,2], 3]})
      end},
    {"extra_side1",
      fun()->
      {ok, Shinobi} = shinobi:prepare(),
      shinobi:register_command(Shinobi, forever, fun Loop(X) -> Loop(X) end),
      shinobi:register_command(Shinobi, ret, fun return2/1),
      Operation = {side_by_side, [{side_by_side, [{rudiment, ret,1}, {rudiment,
ret,2}]}, {side_by_side, [{rudiment, ret,3}, {rudiment, ret,4}]}]},
      {ok, OperationID} = shinobi:operation(Shinobi, Operation),
      ?assertEqual(shinobi:report(OperationID) , {success, [[1,2], [3,4]]})
      end},
    {"comlicated side2",
```

```
      fun()->
      {ok, Shinobi} = shinobi:prepare(),
      shinobi:register_command(Shinobi, forever, fun Loop(X) -> Loop(X) end),
      shinobi:register_command(Shinobi, ret, fun return2/1),
      Operation = {side_by_side, [{side_by_side, [{rudiment, ret,1}, {rudiment,
ret,2}]}, {side_by_side, [{rudiment, ret,3}, {rudiment, ret,4},{rudiment,
ret,5}]}]},
      {ok, OperationID} = shinobi:operation(Shinobi, Operation),
      ?assertEqual(shinobi:report(OperationID) , {success, [[1,2], [3,4,5]]})
      end},
    {"complicated side3",
      fun()->
      {ok, Shinobi} = shinobi:prepare(),
      shinobi:register_command(Shinobi, forever, fun Loop(X) -> Loop(X) end),
      shinobi:register_command(Shinobi, ret, fun return2/1),
      Operation = {side_by_side, [{side_by_side, [{rudiment, ret,1}, {rudiment,
ret,2}]},
                       {side_by_side, [{side_by_side, [{rudiment, ret,1},
{rudiment, ret,2}]}, {rudiment, ret,4},{rudiment, ret,5}]}]},
      {ok, OperationID} = shinobi:operation(Shinobi, Operation),
      ?assertEqual(shinobi:report(OperationID) , {success, [[1,2],
[[1,2],4,5]]})
      end},{
  "function with sleep longer than the timeout, the result should be timeout",
  fun() ->
    {ok,Shinobi} = shinobi:prepare(),
    Add_1_sleep = fun(X) -> timer:sleep(2100), X + 1 end,
    Add_5_sleep = fun(X) -> timer:sleep(2100), X + 5 end,
    Add_7 = fun(X) -> X + 7 end,
    Cmd_Add_1_sleep = add_with_sleep1,
    Cmd_Add_6_sleep = add_with_sleep2,
    Cmd_Add_7 = add_seven,
    shinobi:register_command(Shinobi, Cmd_Add_1_sleep, Add_1_sleep),
    shinobi:register_command(Shinobi, Cmd_Add_6_sleep, Add_5_sleep),
    shinobi:register_command(Shinobi, Cmd_Add_7, Add_7),
    Keikaku = {side_by_side, [{rudiment, Cmd_Add_1_sleep, 15}, {rudiment,
Cmd_Add_6_sleep, 20}, {rudiment, Cmd_Add_7, 31}]},
    {ok, OperationId} = shinobi:operation(Shinobi, Keikaku),
    timer:sleep(500),
    ?assertMatch({failure,timeout}, shinobi:report(OperationId))
  end
}].


test_feint() ->
  [{"simple_timeout",
    fun() ->
    {ok, Shinobi} = shinobi:prepare(),
    shinobi:register_command(Shinobi, forever, fun Loop(X) -> Loop(X) end),
    shinobi:register_command(Shinobi, ret, fun return2/1),
    {ok, OperationID} = shinobi:operation(Shinobi, mkoperation(7)),
    ?assertEqual(shinobi:report(OperationID),{success, timeout})
  end},
  {"feint_failed",
  fun()->
```

```
    {ok, Shinobi} = shinobi:prepare(),
    shinobi:register_command(Shinobi, forever, fun Loop(X) -> Loop(X) end),
    shinobi:register_command(Shinobi, ret, fun return2/1),
    Operation = {feint, {rudiment, ret,1}},
    {ok, OperationID} = shinobi:operation(Shinobi, Operation),
    ?assertEqual(shinobi:report(OperationID),{failure, 1})
  end},
  {"race_feint",
  fun()->
    {ok, Shinobi} = shinobi:prepare(),
    shinobi:register_command(Shinobi, forever, fun Loop(X) -> Loop(X) end),
    shinobi:register_command(Shinobi, ret, fun return2/1),
    Operation = {feint, {race, [{trap, 999}, {rudiment, ret, 3}]}},
    {ok, OperationID} = shinobi:operation(Shinobi, Operation),
    ?assertEqual(shinobi:report(OperationID),{failure, 3})
  end}].


test_relay() ->
  [{"simple_succ",
    fun()->
      {ok, Shinobi} = shinobi:prepare(),
    shinobi:register_command(Shinobi, forever, fun Loop(X) -> Loop(X) end),
    shinobi:register_command(Shinobi, ret, fun return2/1),
    {ok, OperationID} = shinobi:operation(Shinobi, mkoperation(4)),
    ?assertEqual(shinobi:report(OperationID),{success, 2})
  end},
  {"simple_fail",
    fun()->
      {ok, Shinobi} = shinobi:prepare(),
    shinobi:register_command(Shinobi, forever, fun Loop(X) -> Loop(X) end),
    shinobi:register_command(Shinobi, ret, fun return2/1),
    Operation = {relay,{rudiment, ret,1},fun (_,_) ->{rudiment, ret,2}end},
    {ok, OperationID} = shinobi:operation(Shinobi, Operation),
    ?assertEqual(shinobi:report(OperationID), {failure,
not_according_to_keikaku})
    end}].


return2(Arg) -> Arg.
return3(Arg) -> Arg+Arg.

%%minimal ret and forever functions
mkoperation(Opr) ->
  case Opr of
    1 ->
      {progression, [{rudiment, forever, 1}, {rudiment, ret, "one"}]};
    2 ->
      {rudiment, forever, 1};
    3 ->
      {trap, 99};
    4 ->
      {relay,{rudiment, ret,1},fun (_) ->{rudiment, ret,2}end};
    5 ->
      {race, [{rudiment, ret, 1}, {trap, 99}]};
```

```
    6 ->
      {side_by_side, [ {trap, 99}, {rudiment, forever, 2}, {rudiment, forever,
3}, {rudiment, forever, 4}]};
    7 ->
      {feint, {rudiment, forever, 1}}
    end.

test() ->
    {ok, A} = shinobi:prepare(),
    shinobi:register_command(A, forever, fun Loop(X) -> Loop(X) end),
    shinobi:register_command(A, ret, fun return2/1),
    Operation = {race, [{trap, 999}, {race, [{rudiment, ret, 1}, {trap,
999}]}]},
    io:format("Operation~p~n",[Operation]),
    {ok, OperationID} = shinobi:operation(A, Operation),
    Result = shinobi:report(OperationID),
    io:format("Result~p~n",[Result]).
```