Advanced Programming Re-exam Report

gsf346 (Jiashuo Li) Exam number: 2

February 4, 2022

1 Question 1: The Slush template language

1.1 Question 1.1: Parsing Slush

We have to deal with the slush language, a template language that is domain-specific to how structured data values should be rendered in a human-readable form. So first we have to parse those language into an "internal representation form" based on the grammar and given a certain type of AST(abstract syntax tree). I use readP library to implement the parser. In this part, I will show you the details of how to concretely build this parser.

1.1.1 overall:parseString

This is the pre-defined top-level parsing function, which is implemented as:

```
parseString :: String -> Either ParseErr Template
parseString s= case readP_to_S (do; template<-pTemplate;eof;return template) s of
    [(s1,"")] -> Right s1
    _ -> Left "ParseError"
```

We use readP library function readP_to_S, to convert our parser into a Haskell ReadS-style function. In this way we can run our parser, from getting a string as a template to the parser to return a list of possible parses as (a,String) pairs. And other cases, namely if some errors happen when parsing, this function returns a parsing error. And according to the AST:

```
type Template = [Frag]

data Frag =
    TLit String
  | TOutput Exp
  | TAssign Var Exp
  | TIf Exp Template Template
  | TFor Var Exp Template
  | TCapture Var Template
  deriving (Eq, Show, Read)

data Exp =
    EVar Var
  | ENum Int
  | EField Exp Field
  | EPlus Exp Exp
  | ELeq Exp Exp
  deriving (Eq, Show, Read)

```

```haskell
type Var = Ident
type Field = Ident
type Ident = String
```

We deal with the pTemplate in details, which is shown below.

1.1.2 Decompose text into pTemplate and pFrag

Accordibng to the definition of the AST, we could the language into pTemplate, which is lists of pFrags, and further, pFrags also contian several different types of functinoal part like: TLit String, TOutput Exp,TCapture Var Template and so on:

```haskell
pTemplate :: ReadP Template
pTemplate = many pFrag <++ return []
pFrag:: ReadP Frag
pFrag= pConditional <++ pCapture <++ pOutput <++ pIteration  <++ pAssignment <++ pLiteral
```

Later in the report I will show you how these pFrags are implemented.

1.1.3 Handle the whitespace

For white space part, it may contain spaces, tab, next line character, thus we have to skip this content. By using whitespace as well as white spaces, we skip those content, based on the function skipMany and skipMany1. And I use a lexeme to handle the irrelevant part before we meet the real content. So, the implementation code is shown as below:

```haskell
lexeme :: ReadP a -> ReadP a
lexeme p = do a <- p; whitespace; return a
whitespace :: ReadP ()
whitespace =
  skipMany $
    do satisfy (`elem` " \n\t"); return ()
whitespaces :: ReadP ()
whitespaces =
  skipMany1 $
    do satisfy (`elem` " \n\t"); return ()
```

1.1.4 dealing with literal

A literal is any non-empty sequence of characters other than "{". Additionally, so that literals can also contain "{" with a white space. To start with, we have to encapsulate the literal into TLit form to match with the abstract syntax tree:

```haskell
pLiteral :: ReadP Frag
pLiteral = TLit <$> literal
```

Then, we define literal and literal' to deal with multi-cases:

```haskell
literal :: ReadP String
literal = do
  char '{'
  char ' '
  s<-literal'
  return $ "{"++s
  <++
```

```
  (do
  s<-munch1 (/= '{')
  (do
    string "{"
    char ' '
    s'<- literal'
    return  $ s++"{"++s'
   )
   <++ return s)
```

First let's see the literal part. To start with, if the string we meet starts with a left braces and a white space, then we could bound s to the further results of function literal', with the prefix "{"; if the string we parse has the first one or more characters that is not "{"(we use munch1 to design the restrict and bind the left part of the string to s), then we bind the literal' to the latter part to s', and return the string in the form of $s++"{"++s'$. Else, if the above two cases are neither satisfied, we just stop and return the string. Then we have a look at literal':

```
literal' :: ReadP String
literal' = do
  s<-munch (/= '{')
  (do
    string "{"
    char ' '
    s'<- literal'
    return  $ s++"{"++s'
   )
   <++ return s
```

We go on with further parsing process pf literal': I use munch to see if there are zero or more characters that is not "{", and bind this front part to s, followed with "{ " and recursively binding the latter part s' to literal', return the form of $s++"{"++s'$; otherwise, s is just returned and stop parsing.

1.1.5 ident

An ident is any non-empty sequence of ASCII letters, digits, and underscores, starting with a letter. There are no reserved identifiers. And in the AST it is defined as String type. To match with the AST, we bind the ident result to EVar form(Evar Var):

```
pEVar :: ReadP Exp
pEVar = EVar <$> ident
```

So we design the code like this:

```
ident:: ReadP Ident
ident =
  do
    first <- satisfy isLetter
    rest <- munch (\c -> (isAscii  c && isLetter c)|| isNumber c || c == '_')
    return (first:rest)
```

First we judge whether it is a letter, and bind first to it, then for the rest, we use munch to see if there is zero or more letters or numbers or '_', and bind them to rest part. Finally, it return Ident in the form of (first:rest).

1.1.6 numeral

A numeral is any non-empty sequence of decimal digits, optionally preceded by a negative sign. To match with the AST, we bind the numeral result to ENum form(ENum Int):

```
pENum :: ReadP Exp
pENum = ENum <$> numeral
```

We deal with it as follows:

```
numeral =
  do
    first <- satisfy (\c -> isNumber c|| c=='-')
    rest <- munch isNumber
    return (read (first:rest) ::Int)
```

Samilarly, first we judge the first element is a number or a '-'symbol, if is true then we bind the first to it, and bind the rest numerical part to rest, using munch operation, and finally return the result in "Int" form by calling read function.

1.1.7 implement Frag type:TOutput Exp

We design the function pOutput to implement output expression:

```
pOutput :: ReadP Frag
pOutput= do
  string "{{"
  whitespace
  exp<-pExp
  whitespace
  string "}}"
  return $ TOutput exp
```

The idea is: when parsing content satisfied with the start"{{" and the ending "}}", then after skiping the white sapce we bind the inner content to exp, and wrap it into the form (Toutput exp), to get further processing.

1.1.8 implement Frag type:TAssign Var Exp

We design the function pAssignment to implement assignment operation:

```
pAssignment :: ReadP Frag
pAssignment = do
  string "{%"
  whitespace
  string "assign"
  whitespaces
  var<-ident
  whitespaces
  string "="
  whitespaces
  exp<-pExp
  whitespace
  string "%}"
  return $ TAssign var exp
```

The idea is: when parsing content satisfied with the start"{%" and the ending "%}", then after skiping the white sapce we bind the ident to var, bind the expression content to exp, and wrap them into the form (TAssign var exp), to get further processing.

1.1.9 implement Frag type:TIf Exp Template Template

We design the function pConditional to implement if conditions :

```
pConditional :: ReadP Frag
pConditional = do
  string "{%"
  whitespace
  string "if"
  whitespaces
  exp<-pExp
  whitespace
  string "%}"
  template<-pTemplate
  condRest<-pCondRest
  string "{%"
  whitespace
  string "endif"
  whitespace
  string "%}"
  case condRest of
    (Just t)-> return $ TIf exp template t
    Nothing -> return $ TIf exp template []
```

The idea is: After finding the "if" and "endif" sign in the start"{%" and the ending "%}", and skipping the white spaces, we locate the condition statement and bind it to exp for further processing, and also analyse the condRest part, which represents as the "else" part. If condRest is bind with Nothing, then we could just return in the form of (TIf exp template []); otherwise, it should return as: (TIf exp template t). And as for implementing the condRest part, we design as folllows:

```
pCondRest :: ReadP (Maybe Template)
pCondRest =(do
  string "{%"
  whitespace
  string "else"
  whitespace
  string "%}"
  Just <$> pTemplate

  )
  <++
    (do
      string "{%"
      whitespace
      string "elsif"
      whitespaces
      exp<-pExp
      whitespace
      string "%}"
      template<-pTemplate
      condRest<-pCondRest
      case condRest of
        (Just t)->return $ Just [TIf exp template t]
        Nothing->return $ Just [TIf exp template []]
```

```
    )
  <++
      return Nothing
```

In that we divide the problem into three cases: 1."{%"+"else" +"%}"; 2. "{%"+"elsif"+expression +"%}"; 3.Nothing. The corresponding value is then returned for each different case.

1.1.10 implement Frag type:TFor Var Exp Template

We design the function pIteration to implement for-loops :

```
pIteration :: ReadP Frag
pIteration = do
  string "{%"
  whitespace
  string "for"
  whitespaces
  var<-ident
  whitespaces
  string "in"
  whitespaces
  exp<-pExp
  whitespace
  string "%}"
  template<-pTemplate
  string "{%"
  whitespace
  string "endfor"
  whitespace
  string "%}"
  return $ TFor var exp template
```

The idea is that we parse content satisfied with the start"{%" + "for" and the ending "end for" + "%}", then after skiping the white sapce we bind the ident to var, bind the expression content to exp, bind the operation in iteration to template, and wrap them into the form (TFor var exp template), to get further processing.

1.1.11 implement expression with Disambiguation

First we analyse the grammar:

```
Exp ::= Var
| Numeral
| Exp '.' Field
| Exp '+' Exp
| Exp '<=' Exp
| '(' Exp ')
```

From the grammar we could see we have to eliminate left recursion in order to achieve disambiguation, so first we rewrite the grammar into the form without left recursion(also at the same time, take the priority, and left-associative into account):

```
Exp -> Exp' "<=" Exp'| Exp'
Exp' -> E2 E1
E1 -> '+' E2 E1| Nothing
```

```
E2 -> E3 E4
E3 = (E)|Numeral|Var
E4 -> '.' ident E4| Nothing
```

In this way we divide the expression into different part with different priority, with respect to operators' associativity. And based on this new grammar, we could implement the expression with disambiguation.

1.2 Question 1.2: Rendering Slush

In the Rendering module, given the pre-defined function:

```
type EvalM a = Ctx -> Either RenderErr a
type ExecM a = (Ctx,Either RenderErr String) -> (Ctx,Either RenderErr String)
```

we divide into two parts to introduce my design: evaluation and execution.

1.2.1 evaluation

An expression evaluates to a value, or signals an error, so we define our function as:

```
eval :: Exp -> EvalM Value
```

And in this function's implement, depending on the syntax, it can evaluate various types of input and return results or prompt for errors. For example, when dealing with EField $e\ x$, which means e evaluates to a record, and that record includes a field named x, then we have such designs:

```
eval (EField exp field) ctx= do
  v <-eval exp ctx
  case v of
    (R ctx)-> case lookup field ctx of
      Nothing-> Left "Not found in ctx"
      Just v-> return v
    _-> Left "No corresponding field value"
```

The idea is that it: First we evaluate the expression, and bind the result to V. Then we discuss different cases: If v match with the pattern (R ctx), which means we get a corresponding field value, we lookup into the context to find whether the value exist, namely successful find will return the value v, and failure find will return a RendErr "Not found in ctx"; Otherwise, if in the former step we got a v that doesn't match with (R ctx), we will just return a RendErr "No corresponding field value".

1.2.2 execution

An execution of a template may produce some output text, and/or modify the context; alternatively, it may signal an error. So we define our function as:

```
exec :: Template -> ExecM () -- do not change type!
exec [] (ctx,s)=  (ctx,s)
exec template (ctx,s)=  foldl updateResult (ctx,s) template
```

Because template keeps track of the current values of all variables, either pre-existing or introduced in the template, so we have designed the updateResult as follow:

```
updateResult :: (Ctx,Either RenderErr String)->Frag->(Ctx,Either RenderErr String)
```

Which will take the input (Cts, string) and a template Fragment into execution, and the result will be used to influence globally by foldl function.

1.3 Test

I also write about 30 black box test, which result is shown as below:

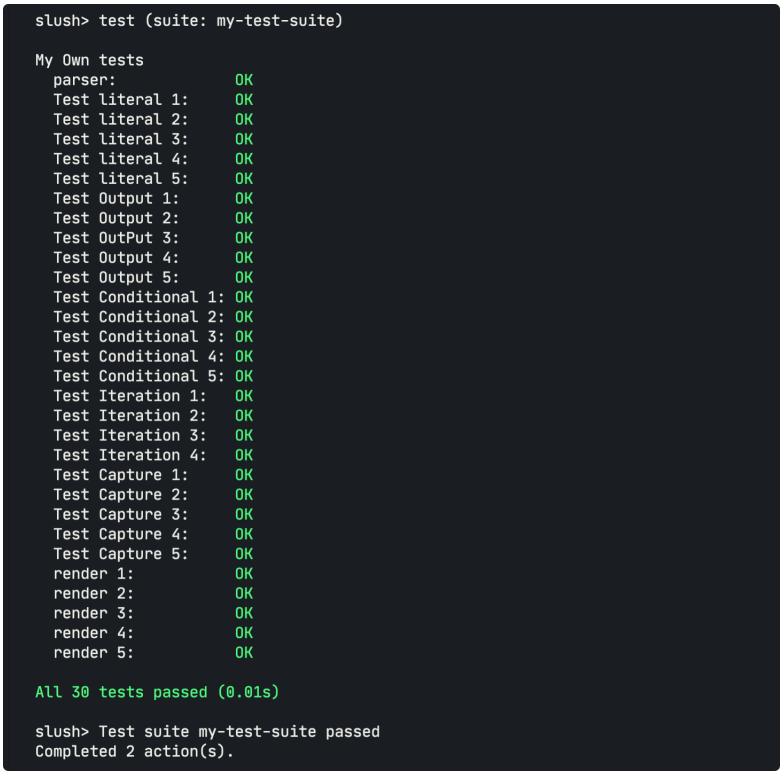

Figure 1: blackbox test

2 Question 2: Observables

2.1 Question 2.1: The observable module

2.1.1 observable implement

We have implemented all the APIs, here is a quick review of those functions:

1. new(): We define the -spec annotation as: new()-¿reply_msg(). This function will call gen_server:start(observable, [], []) with corresponding callback function init(), which does the initialization work:

```
-spec init(_)->reply_msg().
init(_) ->
    Subscribers = [],
    Events = [],
    {ok,{Subscribers, Events}}.
```

   If the process is successfully started it will return ok with its pid(), and return error message if failed.

2. add_subscriber(P, S, Filt, Lim): We define the function as:

```
-spec add_subscriber(pid(),pid(),filter(),limit())-> reply_msg().
add_subscriber(P, S, Filt, Lim) ->
    case Lim of
        infinity-> gen_server:call(P, {add, S, Filt, Lim});
        _-> case is_integer(Lim) of
```

```
                    true -> case Lim > 0 of
                                true -> gen_server:call(P, {add, S, Filt, Lim});
                                false -> {error,"Wrong limit"}
                            end;
                    false -> {error,"Wrong limit"}
                end
        end.
```

The idea is that it first check whether the form of the limit is correctness, and then if the limit is acceptable it use gen_server:call(P, add, S, Filt, L), which directs callback function handle_call to work. Successful execution will return ok with newly updated subscriber lists NewList, Events, failed execution will return an error message, for example multiple add a same subscriber will return error, "Already subscribed".

3. subscribers(P): It will call the function gen_server:call(P, subscribers), which corresponds to the callback function handle_call(subscribers, _From, List, Events):

```
handle_call({subscribers}, _From, {List, Events}) ->
    {Subs, _, _} = lists:unzip3(List),
    {reply, {ok, Subs}, {List, Events}};
```

Successful execution will return ok with a list showing all the subscribers of the publisher, failed execution will return an error message.

4. publish(P, E): To achieve that the function can return before the event has reached all subscribes, we use gen_server:cast to implement asynchronous notifications. And the cast function corresponds with handle_cast function:

```
handle_cast({publish, E, Ref}, {List, Events}) ->
case lists:keyfind(Ref, 2, Events) of
    false -> NewList = publishEvent(E, Ref, List),
             NewEvents = Events ++ [{E,Ref}],
             {noreply, {NewList, NewEvents}};
    _ -> {noreply, {List, Events}}
end.
```

The idea is that it first check the event list that to see if this event is already exist by its unique reference number. If not, it will update the event list and call the publishEvent function, to broadcast the events to all the subscribers. Of course the subscribers could publish the new event to their own subscribers, which repeatedly call the former publish function with its call back function handle_cast(request(),state()). We make sure that an observable server should only (re-)publish an event to its subscribers once, by using the make_ref() to check.

5. events(P) : the events() function will receive the publisher, and return its events list by calling handle_call function with corresponding call_back function handle_call(events, _From, List, Events).

2.1.2 topics that I should mention

1. I support the complete API: new(),add_subscriber(P, S, Filt, Lim),subscribers(P),publish(P, E) and events(P)
2. I made a assumption that the filter function will always return either true or false.
3. processes: Every time we use new() to start a gen_server, it has start a new process, and process can have two roles: publisher and subscriber, which the first is not exclusive to the second. All the process communicate with each other via APIs, thouhg gen_server:start, gen server:call ,gen server:cast, and their corresponding callback functions: init(), handle_call(),handle_cast().

4. data that processes maintain: Process using mainly two type of lists that maintain data: subscriber list and event lists. subscriber list is designed as the form [pid()], and event lists are in the form [integer(),reference()].

5. implementation robust filter: Because I have the assumption that the filter function will always return either true or false, so I do not add extra organism to ensure the robust property.

2.1.3 using observable

According to the pre-requirement of the question, we implement the setup function to initialize the subscribers(and publishers), also adding those subscriber based on the relation graph, with respect to the requirement, for example:

```
observable:add_subscriber(Robin, John, fun(X) ->
                                        case is_integer(X) of
                                            true -> case X rem 2 of
                                                        0 -> true;
                                                        1 -> false
                                                    end;
                                            false -> false
                                        end
                                       end, 1),
```

And by sending events the events are published between different levels of subscribers. Finally in the test_both function, we use:

```
(lists:sort(EventLeslie) == lists:sort([liskov,{small,talk},"Hello"])) and
↪  (lists:sort(EventPeter) == lists:sort(["Hello",4])).
```

to check that if the events of Leslie and Peter are same. And the running result is true, as the picture below, proving the test is successful.

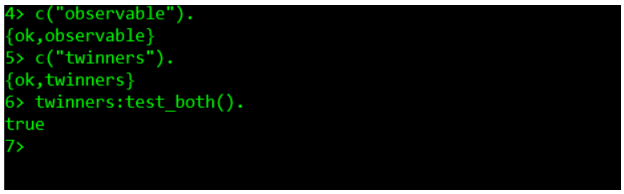

Figure 2: erlang quickchek test result

2.2 Question 2.2: Testing observable

Using Eunit library, I construct a set of unit tests:

```
test_all() ->
  eunit:test(
     [
      test_start_new_process(),
      test_repeated_new_process(),
      test_add1sub_right(),
      test_add1sub_infinite(),
      test_add1sub_wrong(),
      test_add_sub_to_sub(),
```

```
10          test_subscriber_1(),
11          test_subscriber_2(),
12          test_publish_1(),
13          test_event_1(),
14          test_event_2(),
15          test_event_3()
16        ], [verbose]).
```

My strategy is first test possible legal and illegal cases then focus on edge cases and cases that could reflect the property. In this set of tests, I test these cases:

1. Start a new server that does not exist
2. Start a new server that is already exist
3. add one sub to another, with right form
4. add one sub to another, with wrong form
5. add one subscriber to another
6. show subscribers
7. show subscribers, with one failed add operation
8. publish event, normal case
9. show event
10. show different events with same value after multiple publish
11. show same event after multiple publish

These test cases are very representative, and could test a wide parts of my code, but it still has some limitations due to the variety of different edge cases.

There are some tests that are qualitative different from the minimal testing required by the twinners module. For example, test_event_2 shows a case in which different events with same value are published to subscribers, and show the property that subscriber could receive the different events with same value. And all the test is tested, as the picture shows:

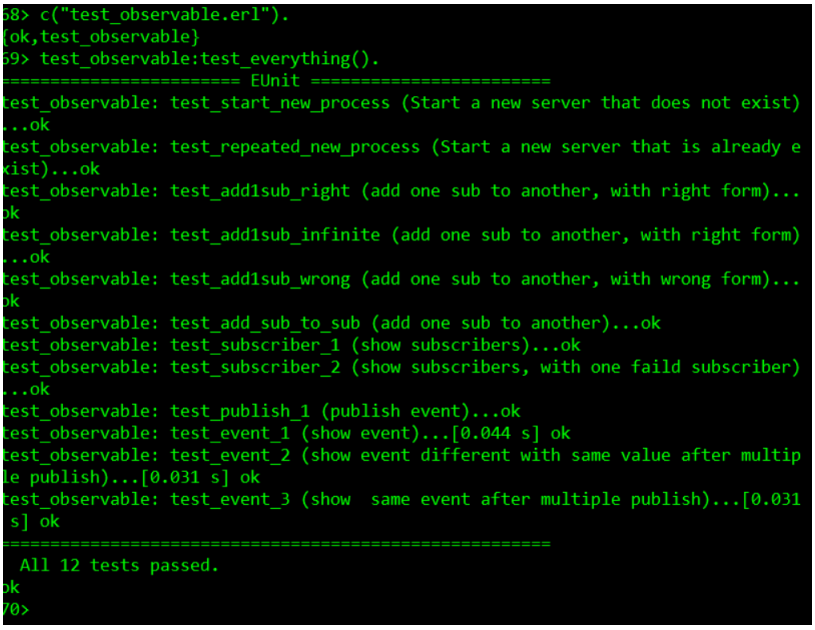

Figure 3: erlang test

3 overall evaluation

3.1 haskell

- **Completeness**:I have finished all the question in the haskell part.
- **Correctness**: I have passed the default tests, and passed the black box tests that I write.
- **Efficiency**: The executed time and space(according to the data structure I use) is reasonable.
- **Maintainability**: I have divided the big parser and render into several small parts, which I think is Maintainable.

3.2 erlang

- **Completeness**:I have finished all the APIs.
- **Correctness**: I have passed the twinner tests, and passed tests that I write.
- **Efficiency**: The executed time and space(according to the data structure I use) is reasonable.

4 code

4.1 haskell

4.1.1 ParserImpl.hs

```
1 module ParserImpl where
2
3 import Ast
4 import Debug.Trace
```

```haskell
import Data.Char ( isLetter, isNumber, isAscii, isSpace )
import Text.ParserCombinators.ReadP
import Text.Parsec.Char (letter)
import Control.Applicative((<|>))
type ParseErr = String
  -- (or something else, as long as it's an instance of Eq and Show

parseString :: String -> Either ParseErr Template -- do not change type!
parseString s= case readP_to_S (do; template<-pTemplate;eof;return template) s of
    [(s1,"")] -> Right s1
    _ -> Left "ParseError"

lexeme :: ReadP a -> ReadP a
lexeme p = do a <- p; whitespace; return a
whitespace :: ReadP ()
whitespace =
  skipMany $
    do satisfy (`elem` " \n\t"); return ()
whitespaces :: ReadP ()
whitespaces =
  skipMany1 $
    do satisfy (`elem` " \n\t"); return ()
pTemplate :: ReadP Template
pTemplate = many pFrag <++ return []
pFrag:: ReadP Frag
pFrag= pConditional <++ pCapture <++ pOutput <++ pIteration  <++ pAssignment <++ pLiteral
pLiteral :: ReadP Frag
pLiteral = TLit <$> literal
pOutput :: ReadP Frag
pOutput= do
  string "{{"
  whitespace
  exp<-pExp
  whitespace
  string "}}"
  return $ TOutput exp
pAssignment :: ReadP Frag
pAssignment = do
  string "{%"
  whitespace
  string "assign"
  whitespaces
  var<-ident
  whitespaces
  string "="
  -- traceM var
  -- traceM "debug2"
  whitespaces
  exp<-pExp
  -- traceM £ show £ TAssign var exp
  whitespace
  string "%}"
```

```
57    return $ TAssign var exp
58  pConditional :: ReadP Frag
59  pConditional = do
60    string "{%"
61    whitespace
62    string "if"
63    whitespaces
64    exp<-pExp
65    whitespace
66    string "%}"
67    template<-pTemplate
68    condRest<-pCondRest
69    string "{%"
70    whitespace
71    string "endif"
72    whitespace
73    string "%}"
74    case condRest of
75      (Just t)-> return $ TIf exp template t
76      Nothing -> return $ TIf exp template []
77  pCondRest :: ReadP (Maybe Template)
78  pCondRest =(do
79    string "{%"
80    whitespace
81    string "else"
82    whitespace
83    string "%}"
84    -- traceM "debug1"
85    -- frag<- pFrag
86    Just <$> pTemplate
87    -- return £ Just [frag]
88    )
89    <++
90      (do
91        string "{%"
92        whitespace
93        string "elsif"
94        whitespaces
95        exp<-pExp
96        whitespace
97        string "%}"
98        template<-pTemplate
99        condRest<-pCondRest
100       case condRest of
101         (Just t)->return $ Just [TIf exp template t]
102         Nothing->return $ Just [TIf exp template []]
103     )
104   <++
105       return Nothing
106
107 pIteration :: ReadP Frag
108 pIteration = do
```

```
    string "{%"
    whitespace
    string "for"
    whitespaces
    var<-ident
    whitespaces
    string "in"
    whitespaces
    exp<-pExp
    whitespace
    string "%}"
    template<-pTemplate
    string "{%"
    whitespace
    string "endfor"
    whitespace
    string "%}"
    return $ TFor var exp template
pCapture :: ReadP Frag
pCapture = do
    string "{%"
    whitespace
    string "capture"
    whitespaces
    var<-ident
    whitespace
    string "%}"
    template<-pTemplate
    string "{%"
    whitespace
    string "endcapture"
    whitespace
    string "%}"
    return $ TCapture var template
pEVar :: ReadP Exp
pEVar = EVar <$> ident
pENum :: ReadP Exp
pENum = ENum <$> numeral
pEClause :: ReadP Exp
pEClause = do
    string "("
    exp <-pExp
    string ")"
    return exp
pExp:: ReadP Exp
pExp = (do
    exp'<-pExp'
    whitespace
    string "<="
    whitespace
    ELeq exp' <$> pExp')
    <++ pExp'
```

```
161
162 pExp' :: ReadP Exp
163 pExp' = do
164   exp2<-pExp2
165   exp1<-pExp1
166   case exp1 of
167     Nothing->return exp2
168     Just e->return $ e exp2
169
170 pExp1 :: ReadP (Maybe (Exp -> Exp))
171 pExp1= (do
172   whitespace
173   string "+"
174   whitespace
175   exp2<-pExp2
176   exp1<-pExp1
177   case exp1 of
178     Nothing->return $ Just $ \e->EPlus e exp2
179     Just e->return $ Just $ \e'->e (EPlus e' exp2)) <++ return Nothing
180
181 pExp2:: ReadP Exp
182 pExp2 = do
183   exp3<-pExp3
184   exp4<-pExp4
185   case exp4 of
186     Nothing -> return exp3
187     Just exp4' -> return $  exp4' exp3
188
189 pExp3:: ReadP Exp
190 pExp3=pEClause <++ pEVar <++ pENum
191
192 pExp4:: ReadP (Maybe (Exp -> Exp))
193 pExp4 = (do
194   string "."
195   i<-ident
196   exp4<-pExp4
197   case exp4 of
198     Nothing -> return $ Just $ \exp3 -> EField exp3 i
199     Just e-> return $ Just $ \e'-> e (EField e' i) ) <++ return Nothing
200
201 literal :: ReadP String
202 literal = do
203   char '{'
204   char ' '
205   s<-literal'
206   return $ "{"++s
207   <++
208   (do
209   s<-munch1 (/= '{')
210   (do
211     string "{"
212     char ' '
```

```
    s'<- literal'
    return  $ s++"{"++s'
   )
   <++ return s)
literal' :: ReadP String
literal' = do
  s<-munch (/= '{')
  (do
    string "{"
    char ' '
    s'<- literal'
    return  $ s++"{"++s'
   )
   <++ return s

ident:: ReadP Ident
ident =
  do
    first <- satisfy isLetter
    rest <- munch (\c -> (isAscii  c && isLetter c)|| isNumber c || c == '_')
    return (first:rest)
numeral:: ReadP Int
numeral =
  do
    first <- satisfy (\c -> isNumber c|| c=='-')
    rest <- munch isNumber
    return (read (first:rest) ::Int)
```

4.1.2 RendererImpl.hs

```
module RendererImpl (RenderErr, render) where

import Ast
import Data
import Control.Monad.Reader

type RenderErr = String
  -- (or something else, as long as it's an instance of Eq and Show

type EvalM a = Ctx -> Either RenderErr a
type ExecM a = (Ctx,Either RenderErr String) -> (Ctx,Either RenderErr String)

eval :: Exp -> EvalM Value -- do not change type!
eval (ENum num) ctx = Right (N num)
eval (EPlus exp1 exp2) ctx= do
  v1<-eval exp1 ctx
  v2<-eval exp2 ctx
  case v1 of
    (N num1)-> case v2 of
      (N num2)-> Right (N (num1+num2))
      _->Left "Error happen in eval EPlus exp2"
    _->Left "Error happen in eval EPlus exp1"
```

```
eval (ELeq exp1 exp2) ctx= do
  v1<-eval exp1 ctx
  v2<-eval exp2 ctx
  case v1 of
    (N num1)-> case v2 of
      (N num2)-> if num1>num2 then Right (N 0) else Right  (N 1)
      _->Left "Error happen in eval ELeq exp2"
    _->Left "Error happen in eval ELeq exp1"
eval (EVar x) ctx= case lookup x ctx of
    Nothing -> Left "Not found"
    Just v->return v
eval (EField exp field) ctx= do
  v <-eval exp ctx
  case v of
    (R ctx)-> case lookup field ctx of
      Nothing-> Left "Not found in ctx"
      Just v-> return v
    _-> Left "No corresponding field value"

exec :: Template -> ExecM () -- do not change type!
exec [] (ctx,s)=  (ctx,s)
exec template (ctx,s)=  foldl updateResult (ctx,s) template

delFromAL :: Eq key => [(key, a)] -> key -> [(key, a)]
delFromAL l key = filter (\a -> fst a /= key) l
updateResult :: (Ctx,Either RenderErr String)->Frag->(Ctx,Either RenderErr String)
updateResult (ctx, Left err) frag=(ctx,Left err)
updateResult (ctx,Right s) (TLit s')= (ctx,Right $ s++s')
updateResult (ctx,Right s) (TOutput e)=case eval e ctx of
  Left err->(ctx,Left err)
  Right v->case v of
    (N num)->(ctx, Right $ s++show num)
    (S s1)->(ctx,Right $ s++s1)
    _-> (ctx,Left "TOutput handle error")
updateResult (ctx, Right s) (TAssign x e)=case eval e ctx of
  Left err-> (ctx,Left err)
  Right v->case lookup x ctx of
    Just a->((x,v):delFromAL ctx x,Right s)
    Nothing ->((x,v):ctx,Right s)
updateResult (ctx,Right s) (TIf e t1 t2)=case eval e ctx of
  Left err->(ctx,Left err)
  Right v-> case v of
    (N 0)->exec t2 (ctx,Right s)
    (N _)->exec t1 (ctx,Right s)
    (S "")->exec t2 (ctx,Right s)
    (S _)->exec t1 (ctx,Right s)
    (L [])->exec t2 (ctx,Right s)
    (L l)->exec t1 (ctx, Right s)
    (R _)->(ctx, Left "TIf handle error")
updateResult (ctx,Right s) (TFor x e t)=case eval e ctx of
  Left err->(ctx,Left err)
  Right v-> case v of
```

```
75      (L [])->(ctx,Right s)
76      -- (L l)-> case culErrOrLastResult £ foldl (\v ctx->exec t ((x,v):delFromAL ctx
        ↪  x,Right "")) (ctx,Right "") l of
77      --   (Right s1)->case culErrOrResult £ map (\v->exec t ((x,v):delFromAL ctx x,Right
        ↪  "")) l of
78      --     (Right s2)->((x,S s1 ):delFromAL ctx x,Right £  s++s2)
79      --     (Left err)->(ctx,Left err)
80      --   (Left err)->(ctx,Left err)
81      (L l)-> case evalTFor (ctx,Right "") x t l of
82        (Right s1)->((x,last l):delFromAL ctx x,Right $ s++s1)
83        (Left err)->(ctx,Left err)
84      _->(ctx,Left "Handle the TFor error")
85  updateResult (ctx,Right s)(TCapture x t)= case snd $ exec t (ctx,Right "") of
86    (Right s')->((x,S s'):delFromAL ctx x,Right $ s++s')
87    (Left err)->(ctx,Left err)
88
89  evalTFor::(Ctx,Either RenderErr String)->Ident->Template->[Value]->Either RenderErr
    ↪  String
90  evalTFor (ctx,Left err) x template l=Left err
91  evalTFor (ctx,Right s) x template []=Right s
92  evalTFor (ctx,Right s) x template (v:l)= case exec template ((x,v):delFromAL ctx x,Right
    ↪  s) of
93    (ctx,Left err)->Left err
94    (ctx,Right s')->evalTFor (ctx,Right s') x template l
95
96  render :: Ctx -> Template -> Either RenderErr String -- do not change type!
97  render ctx t =snd $ exec  t (ctx,Right "")
```

4.1.3 BlackBox.hs

```
1   -- Rudimentary test suite. Feel free to replace anything.
2
3   import Ast
4   import Data
5
6   -- Do not import directly from ParserImpl or RendererImpl here; put
7   -- any white-box tests of internal functions in suite1/WhiteBox.hs
8   import Parser
9   import Renderer
10
11  import Test.Tasty
12  import Test.Tasty.HUnit
13  import Text.Parsec (parse)
14
15  main :: IO ()
16  main = defaultMain $ localOption (mkTimeout 1000000) tests
17
18  tests = testGroup "My Own tests" [
19    testCase "parser" $
20      parseString tms @?= Right tmp,
21    testCase "Test literal 1" $
22      parseString "{ hello" @?= Right [TLit "{hello"],
```

```
23    testCase "Test literal 2" $
24      parseString "hello }" @?= Right [TLit "hello }"],
25    testCase "Test literal 3" $
26      parseString "hello} {  .}  { } } world" @?= Right [TLit "hello} { .}  {} } world"],
27    testCase "Test literal 4" $
28      parseString "{ hello} {  .}  { } } world" @?=Right [TLit "{hello} { .}  {} } world"],
29    testCase "Test literal 5" $
30      parseString "You have ordered the following items:\n" @?= Right [TLit "You have
        ↪  ordered the following items:\n"],
31    testCase "Test Output 1" $
32      parseString "{{x+3+y}}" @?= Right [TOutput (EPlus (EPlus (EVar "x") (ENum 3)) (EVar
        ↪  "y"))],
33    testCase "Test Output 2" $
34      parseString "{{x+(3+y)<=x+(3+y)}}" @?= Right [TOutput (ELeq (EPlus (EVar "x") (EPlus
        ↪  (ENum 3) (EVar "y"))) (EPlus (EVar "x") (EPlus (ENum 3) (EVar "y"))))],
35    testCase "Test OutPut 3" $
36      parseString "{{x+3+y}}" @?=  Right [TOutput (EPlus (EPlus (EVar "x") (ENum 3)) (EVar
        ↪  "y"))],
37    testCase "Test Output 4" $
38      parseString "{{x<=y+z.u}}" @?= Right [TOutput (ELeq (EVar "x") (EPlus (EVar "y")
        ↪  (EField (EVar "z") "u")))],
39    testCase "Test Output 5" $
40      parseString "{{x<=y+z.u+w+(2<=3.c+1.v.u)}}" @?= Right [TOutput (ELeq (EVar "x")
        ↪  (EPlus (EPlus (EPlus (EVar "y") (EField (EVar "z") "u")) (EVar "w")) (ELeq (ENum
        ↪  2) (EPlus (EField (ENum 3) "c") (EField (EField (ENum 1) "v") "u")))))],
41    testCase "Test Conditional 1" $
42      parseString "{% if x+0 %} hello {% else %}world{% endif %}" @?= Right [TIf (EPlus
        ↪  (EVar "x") (ENum 0)) [TLit " hello "] [TLit "world"]],
43    testCase "Test Conditional 2" $
44      parseString "{% if x<=1 %} love {% elsif x+1 %} haske11 {% endif %}" @?= Right [TIf
        ↪  (ELeq (EVar "x") (ENum 1)) [TLit " love "] [TIf (EPlus (EVar "x") (ENum 1)) [TLit
        ↪  " haske11 "] []]],
45    testCase "Test Conditional 3" $
46      parseString "{% if (x) %} I_want_to {% else %}sleep{% endif %}" @?= Right [TIf (EVar
        ↪  "x") [TLit " I_want_to "] [TLit "sleep"]],
47    testCase "Test Conditional 4" $
48      parseString "{% if x+69 %} pls_let_me {% elsif x<=56 %}pass_exam{% endif %}" @?=
        ↪  Right [TIf (EPlus (EVar "x") (ENum 69)) [TLit " pls_let_me "] [TIf (ELeq (EVar
        ↪  "x") (ENum 56)) [TLit "pass_exam"] []]],
49    testCase "Test Conditional 5" $
50      parseString "{% if x.niubijiashuo %} J1aShu0 {% else %} niubility {% endif %}" @?=
        ↪  Right [TIf (EField (EVar "x") "niubijiashuo") [TLit " J1aShu0 "] [TLit "
        ↪  niubility "]],
51    testCase "Test Iteration 1" $
52      parseString "{% for x in y+z %} hello {% endfor %}" @?=Right [TFor "x" (EPlus (EVar
        ↪  "y") (EVar "z")) [TLit " hello "]],
53    testCase "Test Iteration 2" $
54      parseString "{% for love in y<=2 %} hasikou {% endfor %}" @?= Right [TFor "love"
        ↪  (ELeq (EVar "y") (ENum 2)) [TLit " hasikou "]],
55    testCase "Test Iteration 3" $
56      parseString "{% for jiashuo in y.niubi %} jiashuo_niubi {% endfor %}" @?= Right [TFor
        ↪  "jiashuo" (EField (EVar "y") "niubi") [TLit " jiashuo_niubi "]],
```

```
57    testCase "Test Iteration 4" $
58      parseString "{% for doknow in x+1 %} what_to_spell {% endfor %}" @?= Right [TFor
        ↪  "doknow" (EPlus (EVar "x") (ENum 1)) [TLit " what_to_spell "]],
59    testCase "Test Capture 1" $
60      parseString "{% capture x %} hello {% endcapture %}" @?= Right [TCapture "x" [TLit "
        ↪  hello "]],
61    testCase "Test Capture 2" $
62      parseString "{% capture y %} {% for jiashuo in y.niubi %} jiashuo_niubi {% endfor %}
        ↪  {% endcapture %}" @?= Right [TCapture "y" [TLit " ",TFor "jiashuo" (EField (EVar
        ↪  "y") "niubi") [TLit " jiashuo_niubi "],TLit " "]],
63    testCase "Test Capture 3" $
64      parseString "{% capture z %} {% if x.niubijiashuo %} J1aShu0 {% else %} niubility {%
        ↪  endif %} {% endcapture %}" @?= Right [TCapture "z" [TLit " ",TIf (EField (EVar
        ↪  "x") "niubijiashuo") [TLit " J1aShu0 "] [TLit " niubility "],TLit " "]],
65    testCase "Test Capture 4" $
66      parseString "{% capture a %} {% capture b %} {% capture c %} hell0 {% endcapture %}
        ↪  {% endcapture %} {% endcapture %}" @?= Right [TCapture "a" [TLit " ",TCapture "b"
        ↪  [TLit " ",TCapture "c" [TLit " hell0 "],TLit " "],TLit " "]],
67    testCase "Test Capture 5" $
68      parseString "{% capture b %} {% if x<=1 %} {% capture z %} fxck {% endcapture %} {%
        ↪  elsif x+1 %} haske11 {% endif %} {% endcapture %}" @?= Right [TCapture "b" [TLit
        ↪  " ",TIf (ELeq (EVar "x") (ENum 1)) [TLit " ",TCapture "z" [TLit " fxck "],TLit "
        ↪  "] [TIf (EPlus (EVar "x") (ENum 1)) [TLit " haske11 "] []],TLit " "]],
69    testCase "Test render 1" $
70      render ctx tmp @?= Right out,
71    testCase "Test Render 2" $
72      render [("x", R [("niubijiashuo",N 2)] )] [TIf (EField (EVar "x") "niubijiashuo")
        ↪  [TLit " J1aShu0 "] [TLit " niubility "]] @?=Right " J1aShu0 ",
73    testCase "Test Render 3" $
74      render ctx2 tmp2 @?= Right out2,
75    testCase "Test Render 4" $
76      render [] [TCapture "x" [TLit " hello "]] @?=  Right " hello ",
77    testCase "Test Render 5" $
78      render ctx3 tmp3 @?= Right "  J1aShu0  "
79    ]
80
81    where
82      tms = "Hello, {{user.first_name}}!\n"
83      tmp = [TLit "Hello, ",
84             TOutput (EField (EVar "user") "first_name"),
85             TLit "!\n"]
86      ctx = [("user", R [("first_name", S "John"), ("last_name", S "Doe")])]
87      out = "Hello, John!\n"
88      ctx2=  [("order", R [("client", S "John Smith"),
89                  ("items", L [R [("name", S "Universal widget"),
90                                  ("count", N 1)],
91                               R [("name", S "Small gadget"),
92                                  ("count", N 10)]])])]
93      tmp2=  [TIf (EField (EVar "order") "items")
94         [TLit "You have ordered the following items:\n",
95          TAssign "i" (ENum 0),
96          TFor "item" (EField (EVar "order") "items")
```

```
              [TAssign "i" (EPlus (EVar "i") (ENum 1)),
               TLit "\n  ", TOutput (EVar "i"), TLit ". ",
               TOutput (EField (EVar "item") "name"),
               TIf (ELeq (ENum 2) (EField (EVar "item") "count"))
                   [TLit " (quantity ",
                    TOutput (EField (EVar "item") "count"),
                    TLit ")"]
                   [ ],
               TLit "\n"],
        TLit "\nThank you for shopping with us, ",
        TOutput (EField (EVar "order") "client"),
        TLit "!\n"]
       [TLit "You haven't ordered anything yet!\n"],
          TLit "\n"]
    out2="You have ordered the following items:\n\n  1. Universal widget\n\n  2. Small
    ↪  gadget (quantity 10)\n\nThank you for shopping with us, John Smith!\n\n"
    ctx3= [("x", R [("niubijiashuo", N 2)] )]
    tmp3=[TCapture "z" [TLit " ",TIf (EField (EVar "x") "niubijiashuo") [TLit " J1aShu0
    ↪  "] [TLit " niubility "],TLit " "]]
```

4.2 erlang

4.2.1 observable.erl

```
-module(observable).
-behaviour(gen_server).
-export([new/0, add_subscriber/4, subscribers/1, publish/2, events/1]).
-export([init/1, handle_call/3, handle_cast/2]).

%-type event()  ::{integer(),reference()}.
%-type events() ::[event()].
%-type subs_id() :: [pid()].

-type event()  ::term().
-type limit()  :: pos_integer() | infinity.
-type filter() :: fun((event()) -> boolean()).
-type sub()::{pid(),filter(),limit()} .
-type list_subs() ::[sub()].

-type reply_msg():: {error,any()}|{ok,any()}.
-type noreply_msg()::  {error,any()}|{ok,any()}.


-type request()::term().
-type from():: pid().
-type state()::term().
-type result()::term.

-spec new()->reply_msg().  %{ok,pid()}|{error, term()}  {ok,{Subscribers, Events}}
new() ->
  case gen_server:start(observable, [], []) of
        {ok, P}-> {ok, P};
        {error,Reason}  -> {error, Reason}
```

```
30      end.
31
32  -spec add_subscriber(pid(),pid(),filter(),limit())-> reply_msg().
33  add_subscriber(P, S, Filt, Lim) ->
34      case Lim of
35          infinity-> gen_server:call(P, {add, S, Filt, Lim});
36          _-> case is_integer(Lim) of
37                  true -> case Lim > 0 of
38                              true -> gen_server:call(P, {add, S, Filt, Lim});
39                              false -> {error,"Wrong limit"}
40                          end;
41                  false -> {error,"Wrong limit"}
42              end
43      end.
44  -spec subscribers(pid()) -> reply_msg(). % {ok,[<0.92.0>]}
45
46
47  subscribers(P) ->
48      gen_server:call(P, {subscribers}).
49
50  -spec publish(pid(),event())->noreply_msg().  % ok
51  publish(P, E) ->
52      gen_server:cast(P, {publish, E, make_ref()}).
53
54  -spec events(pid()) ->reply_msg().  %{ok,[5,a,a]}
55  events(P) ->
56       gen_server:call(P, {events}).
57
58  -spec init(_)->reply_msg().
59  init(_) ->
60      Subscribers = [],
61      Events = [],
62      {ok,{Subscribers, Events}}.
63
64  -spec handle_call(request(),from(),state()) ->reply_msg().
65  handle_call({add, S, Filt, Lim}, _From, {List, Events}) ->
66      case lists:keyfind(S, 1, List) of
67          false -> NewList = List ++ [{S, Filt, Lim}],
68                   {reply, ok, {NewList, Events}};
69          _-> {reply, {error, "Already subscribed"}, {List, Events}}
70      end;
71
72  handle_call({subscribers}, _From, {List, Events}) ->
73      {Subs, _, _} = lists:unzip3(List),
74      {reply, {ok, Subs}, {List, Events}};
75
76  handle_call({events}, _From, {List, Events}) ->
77      {Evs, _} = lists:unzip(Events),
78      {reply, {ok, Evs}, {List, Events}}.
79
80  -spec handle_cast(request(),state()) ->noreply_msg().
81  handle_cast({publish, E, Ref}, {List, Events}) ->
```

```
    case lists:keyfind(Ref, 2, Events) of
        false -> NewList = publishEvent(E, Ref, List),
                 NewEvents = Events ++ [{E,Ref}],
                 {noreply, {NewList, NewEvents}};
        _ -> {noreply, {List, Events}}
    end.

-spec publishEvent(event(),reference(),list_subs())->result().
publishEvent(_, _, []) -> [];
publishEvent(E, Ref, [{S, Filt, Lim}|Rest]) ->
    case Filt(E) of
        true -> gen_server:cast(S, {publish, E, Ref}),
                case Lim of
                    infinity -> [{S, Filt, infinity}] ++ publishEvent(E, Ref, Rest);
                    _ -> case Lim - 1  of
                            0 -> publishEvent(E, Ref, Rest);
                            A -> [{S, Filt, A}] ++ publishEvent(E, Ref, Rest)
                        end
                end;
        _ -> [{S, Filt, Lim}] ++ publishEvent(E, Ref, Rest)
    end.
```

4.2.2 twinners.erl

```
-module(twinners).
-export([setup/0, send_events/4, test_both/0]).

setup() ->
    {ok, Tony} = observable:new(),
    {ok, Robin} = observable:new(),
    {ok, John} = observable:new(),
    {ok, Alan} = observable:new(),
    {ok, Peter} = observable:new(),
    {ok, Barbara} = observable:new(),
    {ok, Leslie} = observable:new(),
    observable:add_subscriber(Robin, John, fun(X) ->
                                            case is_integer(X) of
                                                true -> case X rem 2 of
                                                            0 -> true;
                                                            1 -> false
                                                        end;
                                                false -> false
                                            end
                                           end, 1),
    observable:add_subscriber(Tony, John, fun(_) -> true end, infinity),
    observable:add_subscriber(John, Peter, fun(_) -> true end, 2),
    observable:add_subscriber(John, Barbara, fun(List) ->
                                               case is_list(List) of
                                                true -> case lists:nth(2, List) of
                                                            101 -> true;
                                                            _ -> false
                                                        end;
```

```
                                                    false -> false
                                                    end
                                                end , infinity),
    observable:add_subscriber(Alan, Barbara, fun(_) -> true end, infinity),
    observable:add_subscriber(Barbara, Leslie, fun(_) -> true end, infinity),
    [Tony, Robin, John, Alan, Peter, Barbara, Leslie].

send_events(X, Y, W, Z) ->
    observable:publish(X, 5),
    observable:publish(X, 4),
    observable:publish(X, point),
    observable:publish(Y, "Hello"),
    observable:publish(W, {small, talk}),
    observable:publish(Z, liskov).

test_both() ->
    [_Tony, Robin, John, Alan, Peter, Barbara, Leslie] = setup(),
    send_events(Robin, John, Alan, Barbara),
    timer:sleep(30),
    {ok, EventLeslie} = observable:events(Leslie),
    {ok, EventPeter} = observable:events(Peter),
    (lists:sort(EventLeslie) == lists:sort([liskov,{small,talk},"Hello"])) and
    ↪  (lists:sort(EventPeter) == lists:sort(["Hello",4])).
```

4.2.3 test_observable.erl

```
-module(test_observable).
-include_lib("eunit/include/eunit.hrl").
-export([test_all/0, test_everything/0]).


% 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_observable.


test_all() ->
   eunit:test(
      [
       test_start_new_process(),
       test_repeated_new_process(),
       test_add1sub_right(),
       test_add1sub_infinite(),
       test_add1sub_wrong(),
       test_add_sub_to_sub(),
       test_subscriber_1(),
       test_subscriber_2(),
       test_publish_1(),
       test_event_1(),
       test_event_2(),
       test_event_3()
```

```
27        ], [verbose]).
28
29  test_start_new_process()->
30  { "Start a new server that does not exist",
31    fun() ->
32      ?assertMatch({ok, _}, observable:new())
33
34
35    end
36  }.
37
38  test_repeated_new_process()->
39  { "Start a new server that is already exist",
40    fun() ->
41      {ok, B} = observable:new(),
42      ?assertNotMatch({ok, B}, observable:new())
43    end
44  }.
45
46
47  test_add1sub_right()->
48  { "add one sub to another, with right form",
49    fun() ->
50      {ok, C} = observable:new(),
51      {ok, D} = observable:new(),
52       ?assertMatch(ok, observable:add_subscriber(C,D, fun(X) -> X>0 end,100))
53
54
55    end
56  }.
57  test_add1sub_infinite()->
58  { "add one sub to another, with right form",
59    fun() ->
60      {ok, E} = observable:new(),
61      {ok, F} = observable:new(),
62       ?assertMatch(ok, observable:add_subscriber(E,F, fun(X) -> X>0 end,infinity))
63
64
65    end
66  }.
67  test_add1sub_wrong()->
68  { "add one sub to another, with wrong form",
69    fun() ->
70      {ok, G} = observable:new(),
71      {ok, H} = observable:new(),
72      ?assertMatch({error,_}, observable:add_subscriber(G,H, fun(X) -> X>0 end,-666))
73
74
75    end
76  }.
77
78  test_add_sub_to_sub()->
```

```
79  { "add one sub to another",
80    fun() ->
81      {ok, A1} = observable:new(),
82      {ok, B1} = observable:new(),
83      {ok, C1} = observable:new(),
84      observable:add_subscriber(A1,B1, fun(X) -> X>0 end,100),
85      ?assertMatch(ok, observable:add_subscriber(B1,C1, fun(X) -> X*(X-1)<0 end,100))
86
87
88    end
89  }.
90
91  test_subscriber_1()->
92  { "show subscribers",
93    fun() ->
94      {ok, A2} = observable:new(),
95      {ok, B2} = observable:new(),
96      {ok, C2} = observable:new(),
97      observable:add_subscriber(A2,B2, fun(X) -> X>0 end,100),
98      observable:add_subscriber(A2,C2, fun(X) -> X*(X-1)<0 end,infinity),
99      ?assertMatch({ok,[B2,C2]}, observable:subscribers(A2))
100
101
102   end
103 }.
104
105 test_subscriber_2()->
106 { "show subscribers, with one faild subscriber",
107   fun() ->
108     {ok, A3} = observable:new(),
109     {ok, B3} = observable:new(),
110     {ok, C3} = observable:new(),
111     observable:add_subscriber(A3,B3, fun(X) -> X>0 end,100),
112     observable:add_subscriber(A3,C3, fun(X) -> X*(X-1)<0 end,-731),
113     ?assertEqual({ok,[B3]}, observable:subscribers(A3))
114
115   end
116 }.
117
118 test_publish_1()->
119 { "publish event",
120   fun() ->
121     {ok, A4} = observable:new(),
122     {ok, B4} = observable:new(),
123     observable:add_subscriber(A4,B4, fun(X) -> X>0 end,11),
124     ?assertEqual(ok,observable:publish(A4,5))
125
126   end
127 }.
128 test_event_1()->
129 { "show event",
130   fun() ->
```

```
131       {ok, A5} = observable:new(),
132       {ok, B5} = observable:new(),
133       {ok, C5} = observable:new(),
134       observable:add_subscriber(A5,B5, fun(X) -> X>0 end,100),
135       observable:add_subscriber(A5,C5, fun(X) -> X*(X-1)>0 end,-731),
136       observable:publish(A5,5),
137       timer:sleep(30),
138       ?assertEqual({ok,[5]}, observable:events(B5))
139
140     end
141   }.
142   test_event_2()->
143   { "show event different with same value after multiple publish",
144     fun() ->
145       {ok, A6} = observable:new(),
146       {ok, B6} = observable:new(),
147       {ok, C6} = observable:new(),
148       observable:add_subscriber(A6,B6, fun(X) -> X>0 end,100),
149       observable:add_subscriber(B6,C6, fun(X) -> X*(X-1)>0 end,100),
150       observable:publish(A6,5),
151       observable:publish(B6,5),
152       timer:sleep(30),
153       ?assertEqual({ok,[5,5]}, observable:events(C6))
154
155     end
156   }.
157
158   test_event_3()->
159   { "show  same event after multiple publish",
160     fun() ->
161       {ok, A7} = observable:new(),
162       {ok, B7} = observable:new(),
163       {ok, C7} = observable:new(),
164       {ok, D7} = observable:new(),
165       observable:add_subscriber(A7,B7, fun(X) -> X>0 end,100),
166       observable:add_subscriber(A7,C7, fun(X) -> X*(X-1)>0 end,100),
167       observable:add_subscriber(B7,D7, fun(X) -> X>0 end,100),
168       observable:add_subscriber(C7,D7, fun(X) -> X>0 end,100),
169       observable:publish(A7,1000),
170       timer:sleep(30),
171       ?assertEqual({ok,[1000]}, observable:events(D7))
172
173     end
174   }.
175
176
177
178
179   test_everything() ->
180     test_all().
```