PrettyRoo.hs

```haskell
-----------------------------------------------------------
-- COMP90045 Programming Language Implementation Project --
--                     Roo Compiler                      --
--  Implemented by Xulin Yang                            --
--  read from the bottom to top                          --
-----------------------------------------------------------
module PrettyRoo (pp)
where
import RooAST
import Data.List

--  Pretty print a whole program:
pp :: Program -> String
pp = strProgram

-----------------------------------------------------------------
--  helper functions
-----------------------------------------------------------------

-- add space indentations based on the input indentation level
addIndentation :: Int -> String
addIndentation 0 = ""
addIndentation n = "    " ++ addIndentation (n - 1)

newline :: String
newline = "\n"

semicolon :: String
semicolon = ";"

comma :: String
comma = ", "  -- There should be a single space after a comma

surroundByParens :: String -> String
surroundByParens s = "(" ++ s ++ ")"

surroundByBrackets :: String -> String
surroundByBrackets s = "[" ++ s ++ "]"

-----------------------------------------------------------------
--  AST toString functions
-----------------------------------------------------------------

strBaseType :: BaseType -> String
strBaseType BooleanType = "boolean"
strBaseType IntegerType = "integer"

strDataType :: DataType -> String
strDataType (AliasDataType t) = t -- t is String already
strDataType (BasyDataType b) = strBaseType b

```

```
strBooleanLiteral :: BooleanLiteral -> String
strBooleanLiteral True = "true"
strBooleanLiteral False = "false"

-- StringLiteral can be directly used as it is String type

-- IntegerLiteral can be turned to string by show as it is Int type

-------------------------------------------------------------------
-- An lvalue (<lvalue>) has four (and only four) possible forms:
--      <id>
--      <id>.<id>
--      <id>[<exp>]
--      <id>[<exp>].<id>
-------------------------------------------------------------------
strLValue :: LValue -> String
strLValue (LId ident) = ident
strLValue (LDot ident1 ident2) = ident1 ++ "." ++ ident2
strLValue (LBrackets ident exp) = ident ++ (surroundByBrackets (strExp exp))
strLValue (LBracketsDot ident1 exp ident2) = ident1 ++ (surroundByBrackets (strExp exp)) ++ "." ++ ident2

-------------------------------------------------------------------
-- Exp related toString & helper functions (with parens elimination)
-------------------------------------------------------------------
-- exp1 has higher precendence than exp2, higher if exp2 has no operator
isHigherPrecendence :: Exp -> Exp -> Bool
isHigherPrecendence exp1@(Op_neg _) exp2 =
  case exp2 of
    (Op_neg _  ) -> False
    _            -> True
isHigherPrecendence exp1@(Op_mul _ _) exp2 =
  case exp2 of
    (Op_neg _  ) -> False
    (Op_mul _ _) -> False
    (Op_div _ _) -> False
    _            -> True
isHigherPrecendence exp1@(Op_div _ _) exp2 =
  case exp2 of
    (Op_neg _  ) -> False
    (Op_mul _ _) -> False
    (Op_div _ _) -> False
    _            -> True
isHigherPrecendence exp1@(Op_add _ _) exp2 =
  case exp2 of
    (Op_neg _  ) -> False
    (Op_mul _ _) -> False
    (Op_div _ _) -> False
    (Op_add _ _) -> False
    (Op_sub _ _) -> False
    _            -> True
isHigherPrecendence exp1@(Op_sub _ _) exp2 =
  case exp2 of
    (Op_neg _  ) -> False
    (Op_mul _ _) -> False
    (Op_div _ _) -> False
    (Op_add _ _) -> False
    (Op_sub _ _) -> False
    _            -> True
isHigherPrecendence exp1@(Op_eq _ _) exp2 =
  case exp2 of
    (Op_not _  )  -> True
    (Op_and _ _)  -> True
    (Op_or  _ _)  -> True
    -- constants has lowest precendence
    (Lval _) -> True
    (BoolConst _) -> True
    (IntConst _) -> True
    (StrConst _) -> True
    _             -> False
isHigherPrecendence exp1@(Op_neq _ _) exp2 =
  case exp2 of
    (Op_not _  )  -> True
    (Op_and _ _)  -> True
    (Op_or  _ _)  -> True
    -- constants has lowest precendence
    (Lval _) -> True
    (BoolConst _) -> True
    (IntConst _) -> True
    (StrConst _) -> True
    _             -> False
isHigherPrecendence exp1@(Op_less _ _) exp2 =
  case exp2 of
    (Op_not _  )  -> True
    (Op_and _ _)  -> True
    (Op_or  _ _)  -> True
    -- constants has lowest precendence
    (Lval _) -> True
    (BoolConst _) -> True
    (IntConst _) -> True
    (StrConst _) -> True
    _             -> False
isHigherPrecendence exp1@(Op_less_eq _ _) exp2 =
  case exp2 of
    (Op_not _  )  -> True
    (Op_and _ _)  -> True
    (Op_or  _ _)  -> True
    -- constants has lowest precendence
    (Lval _) -> True
    (BoolConst _) -> True
    (IntConst _) -> True
    (StrConst _) -> True
    _             -> False
isHigherPrecendence exp1@(Op_large _ _) exp2 =
  case exp2 of
    (Op_not _  )  -> True
    (Op_and _ _)  -> True
    (Op_or  _ _)  -> True
    -- constants has lowest precendence
    (Lval _) -> True
    (BoolConst _) -> True
    (IntConst _) -> True
    (StrConst _) -> True
    _             -> False
isHigherPrecendence exp1@(Op_large_eq _ _) exp2 =
  case exp2 of
    (Op_not _  )  -> True
    (Op_and _ _)  -> True
    (Op_or  _ _)  -> True
    -- constants has lowest precendence
    (Lval _) -> True
    (BoolConst _) -> True
    (IntConst _) -> True
    (StrConst _) -> True
    _             -> False
isHigherPrecendence exp1@(Op_not _) exp2 =
  case exp2 of
    (Op_and _ _)  -> True
    (Op_or  _ _)  -> True
    -- constants has lowest precendence
    (Lval _) -> True
    (BoolConst _) -> True
    (IntConst _) -> True
    (StrConst _) -> True
    _             -> False
isHigherPrecendence exp1@(Op_and _ _) exp2 =
  case exp2 of
    (Op_or  _ _)  -> True
    -- constants has lowest precendence
    (Lval _) -> True
    (BoolConst _) -> True
    (IntConst _) -> True
    (StrConst _) -> True
    _             -> False
isHigherPrecendence exp1@(Op_or _ _) exp2
   -- constants has lowest precendence
  | (not (hasOperatorExp exp2)) = True
  | otherwise = False
isHigherPrecendence _ _ = False

-- exp1 has same precendence as exp2
isSamePrecendence :: Exp -> Exp -> Bool
isSamePrecendence exp1@(Op_neg _) exp2 =
  case exp2 of
    (Op_neg _  ) -> True
    _            -> False
isSamePrecendence exp1@(Op_mul _ _) exp2 =
  case exp2 of
    (Op_mul _ _) -> True
    (Op_div _ _) -> True
    _            -> False
isSamePrecendence exp1@(Op_div _ _) exp2 =
  case exp2 of
    (Op_mul _ _) -> True
    (Op_div _ _) -> True
    _            -> False
isSamePrecendence exp1@(Op_add _ _) exp2 =
  case exp2 of
    (Op_add _ _) -> True
```

```haskell
    (Op_sub _ _) -> True
    _            -> False
isSamePrecendence exp1@(Op_sub _ _) exp2 =
  case exp2 of
    (Op_add _ _) -> True
    (Op_sub _ _) -> True
    _            -> False
isSamePrecendence exp1@(Op_eq _ _) exp2       = False -- relational
isSamePrecendence exp1@(Op_neq _ _) exp2      = False -- relational
isSamePrecendence exp1@(Op_less _ _) exp2     = False -- relational
isSamePrecendence exp1@(Op_less_eq _ _) exp2  = False -- relational
isSamePrecendence exp1@(Op_large _ _) exp2    = False -- relational
isSamePrecendence exp1@(Op_large_eq _ _) exp2 = False -- relational
isSamePrecendence exp1@(Op_not _) exp2 =
  case exp2 of
    (Op_not _)  -> True
    _             -> False
isSamePrecendence exp1@(Op_and _ _) exp2 =
  case exp2 of
    (Op_and _ _)  -> True
    _             -> False
isSamePrecendence exp1@(Op_or _ _) exp2 =
  case exp2 of
    (Op_or  _ _)  -> True
    _             -> False
isSamePrecendence _ _ = False

-- exp1 has smaller precendence than exp2 if it is not higher nor same
isSamllerPrecendence :: Exp -> Exp -> Bool
isSamllerPrecendence exp1 exp2 = (not (isHigherPrecendence exp1 exp2)) &&
                                 (not (isSamePrecendence   exp1 exp2))

-- does expression has operator in it?
hasOperatorExp :: Exp -> Bool
hasOperatorExp (Lval _) = False
hasOperatorExp (BoolConst _) = False
hasOperatorExp (IntConst _) = False
hasOperatorExp (StrConst _) = False
hasOperatorExp _ = True

-- return true if parent is sub/div and child has same precendence as parent
--    pexp: parent expression
--    cexp: child  expression
isDivSubParentSamePrecChild :: Exp -> Exp -> Bool
isDivSubParentSamePrecChild pexp@(Op_div _ _) cexp@(Op_div _ _) = True  -- / with a right child of / need
isDivSubParentSamePrecChild pexp@(Op_div _ _) cexp@(Op_mul _ _) = True  -- / with a right child of * need
isDivSubParentSamePrecChild pexp@(Op_sub _ _) cexp@(Op_sub _ _) = True  -- - (sub) with a right child of -
isDivSubParentSamePrecChild pexp@(Op_sub _ _) cexp@(Op_add _ _) = True  -- - (sub) with a right child of +
isDivSubParentSamePrecChild _ _ = False

-- True if Integer division happens
--    Integer division: 3 * (5 / 3) = 3 * 1 = 3, but (3 * 5) / 3 = 15 / 3 = 5,  so parens needed
isIntegerDision :: Exp -> Exp -> Bool
isIntegerDision pexp@(Op_mul _ _) cexp@(Op_div _ _) = True
isIntegerDision _ _ = False

```

```
-- some notation:
--    pexp: parent      expression (definitely has operator)
--    exp1: left child  expression
--    exp2: right child expression
-- turn binary expression's left child to string
strBinaryExpLChild :: Exp -> Exp -> String
strBinaryExpLChild pexp exp1
  -- left child (with operator) has lower precendence suggests a parens
  | (isSamllerPrecendence exp1 pexp) && (hasOperatorExp exp1) = surroundByParens (strExp exp1)
  -- no parens
  | otherwise = strExp exp1

-- True if expression is "not" <exp>
isNotExp :: Exp -> Bool
isNotExp (Op_not _) = True
isNotExp _ = False

-- some notation:
--    pexp: parent      expression (definitely has operator)
--    exp1: left child  expression
--    exp2: right child expression
-- turn binary expression's right child to string
strBinaryExpRChild :: Exp -> Exp -> String
strBinaryExpRChild pexp exp2
  -- right child (with operator) has lower precendence (except "not" operator) or
  --    same predence as parent (if parent is div(/) or sub(-)) or
  --    integer division happens
  -- suggests a parens
  | (hasOperatorExp exp2) &&
    ((isDivSubParentSamePrecChild pexp exp2) ||
      (isSamllerPrecendence exp2 pexp) ||
      (isIntegerDision pexp exp2)
    ) &&
    (not (isNotExp exp2))
      = surroundByParens (strExp exp2)
  -- no parens
  | otherwise = strExp exp2

-- some notation:
--    pexp: parent      expression
--    exp1: left child  expression
--    exp2: right child expression
strExp :: Exp -> String
-- <lvalue>
strExp (Lval lValue) = strLValue lValue
-- <const>
strExp (BoolConst booleanLiteral) = strBooleanLiteral booleanLiteral
-- <const>
strExp (IntConst integerLiteral) = show integerLiteral
-- <const>
-- White space, and upper/lower case, should be preserved inside strings.
-- stringLiterals
strExp (StrConst stringLiteral) = "\"" ++ stringLiteral ++ "\""
-- <unop: "-"> <exp>
-- no space after unary minus
strExp pexp@(Op_neg exp)
```

```
  | (hasOperatorExp exp) && (not (isSamePrecendence pexp exp)) = "-" ++ surroundByParens (strExp exp) -- n
  | otherwise         = "-" ++ (strExp exp) -- no need to parens constants/Lval
-- <exp> <binop: "*"> <exp>
-- Single space should surround 12 binary operators.
strExp pexp@(Op_mul exp1 exp2) = (strBinaryExpLChild pexp exp1) ++ " * " ++ (strBinaryExpRChild pexp exp2)
-- <exp> <binop: "/"> <exp>
-- Single space should surround 12 binary operators.
strExp pexp@(Op_div exp1 exp2) = (strBinaryExpLChild pexp exp1) ++ " / " ++ (strBinaryExpRChild pexp exp2)
-- <exp> <binop: "+"> <exp>
-- Single space should surround 12 binary operators.
strExp pexp@(Op_add exp1 exp2) = (strBinaryExpLChild pexp exp1) ++ " + " ++ (strBinaryExpRChild pexp exp2)
-- <exp> <binop: "-"> <exp>
-- Single space should surround 12 binary operators.
strExp pexp@(Op_sub exp1 exp2) = (strBinaryExpLChild pexp exp1) ++ " - " ++ (strBinaryExpRChild pexp exp2)
-- <exp> <binop: "="> <exp>
-- Single space should surround 12 binary operators.
strExp pexp@(Op_eq exp1 exp2) = (strBinaryExpLChild pexp exp1) ++ " = " ++ (strBinaryExpRChild pexp exp2)
-- <exp> <binop: "!="> <exp>
-- Single space should surround 12 binary operators.
strExp pexp@(Op_neq exp1 exp2) = (strBinaryExpLChild pexp exp1) ++ " != " ++ (strBinaryExpRChild pexp exp2
-- <exp> <binop: "<"> <exp>
-- Single space should surround 12 binary operators.
strExp pexp@(Op_less exp1 exp2) = (strBinaryExpLChild pexp exp1) ++ " < " ++ (strBinaryExpRChild pexp exp2
-- <exp> <binop: "<="> <exp>
-- Single space should surround 12 binary operators.
strExp pexp@(Op_less_eq exp1 exp2) = (strBinaryExpLChild pexp exp1) ++ " <= " ++ (strBinaryExpRChild pexp
-- <exp> <binop: ">"> <exp>
-- Single space should surround 12 binary operators.
strExp pexp@(Op_large exp1 exp2) = (strBinaryExpLChild pexp exp1) ++ " > " ++ (strBinaryExpRChild pexp exp
-- <exp> <binop: ">="> <exp>
-- Single space should surround 12 binary operators.
strExp pexp@(Op_large_eq exp1 exp2) = (strBinaryExpLChild pexp exp1) ++ " >= " ++ (strBinaryExpRChild pexp
-- <unop: not> <exp>
-- There should be a single space after not.
strExp pexp@(Op_not exp)
  -- need to parens expression with operator and (child's precendence is smaller)
  | (hasOperatorExp exp) && (isSamllerPrecendence exp pexp) = "not " ++ surroundByParens (strExp exp)
  | otherwise         = "not " ++ (strExp exp) -- no need to parens constants/Lval
-- <exp> <binop: and> <exp>
-- Single space should surround 12 binary operators.
strExp pexp@(Op_and exp1 exp2) = (strBinaryExpLChild pexp exp1) ++ " and " ++ (strBinaryExpRChild pexp exp
-- <exp> <binop: or> <exp>
-- Single space should surround 12 binary operators.
strExp pexp@(Op_or exp1 exp2) = (strBinaryExpLChild pexp exp1) ++ " or " ++ (strBinaryExpRChild pexp exp2)


-- Int: indentation level
strStmt :: Int -> Stmt -> String
-- In a procedure body, each statement should start on a new line. So ++ newline in each's end
strStmt indentLevel (Assign lValue exp) =
  -- <lvalue> <- <exp>;
  -- Single spaces should surround the assignment operator <-
  (addIndentation indentLevel) ++ (strLValue lValue) ++ " <- " ++ (strExp exp) ++ semicolon ++ newline
strStmt indentLevel (Read lValue) =
  -- read <lvalue>;
  (addIndentation indentLevel) ++ "read " ++ (strLValue lValue) ++ semicolon ++ newline
```

```
strStmt indentLevel (Write exp) =
  -- write <exp>;
  (addIndentation indentLevel) ++ "write " ++ (strExp exp) ++ semicolon ++ newline
strStmt indentLevel (Writeln exp) =
  -- writeln <exp>;
  (addIndentation indentLevel) ++ "writeln " ++ (strExp exp) ++ semicolon ++ newline
strStmt indentLevel (Call ident exps) =
  -- call <id>(<exp-list>);
  --     where <exp-list> is a (possibly empty according to parser) comma-separated list of expressions.
  (addIndentation indentLevel) ++ "call " ++ ident ++ surroundByParens (intercalate comma (map strExp exps
-- thenStmts is non-empty according to parser, elseStmts is possible empty according to parser
strStmt indentLevel (If exp thenStmts elseStmts) =
  -- IF elseStmts is empty: according to parser: if <exp> then <stmt-list> fi
  if null elseStmts then
    -- "if ... then" should be printed on one line, irrespective of the size of the intervening expression
    (addIndentation indentLevel) ++ "if " ++ (strExp exp) ++ " then" ++ newline ++
    -- more indentation
    (concatMap (strStmt (indentLevel+1)) thenStmts) ++
    -- the terminating fi should be indented exactly as the corresponding if.
    (addIndentation indentLevel) ++ "fi" ++ newline
  -- OTHERWISE            : according to parser: if <expr> then <stmt-list> else <stmt-list> fi
  else
    -- "if ... then" should be printed on one line, irrespective of the size of the intervening expression
    (addIndentation indentLevel) ++ "if " ++ (strExp exp) ++ " then" ++ newline ++
    -- more indentation
    (concatMap (strStmt (indentLevel+1)) thenStmts) ++
    (addIndentation indentLevel) ++ "else" ++ newline ++
    -- more indentation
    (concatMap (strStmt (indentLevel+1)) elseStmts) ++
    -- the terminating fi and else should be indented exactly as the corresponding if.
    (addIndentation indentLevel) ++ "fi" ++ newline
-- stmts is non-empty according to parser
strStmt indentLevel (While exp stmts) =
  -- In a while statement, "while ... do" should be printed on one line,
  --    irrespective of the size of the intervening expression
  (addIndentation indentLevel) ++ "while " ++ (strExp exp) ++ " do" ++ newline ++
  -- more indentation
  (concatMap (strStmt (indentLevel+1)) stmts) ++
  -- The terminating od should be indented exactly as the corresponding while.
  (addIndentation indentLevel) ++ "od" ++ newline

-----------------------------------------------------------------
--  Record toString function
-----------------------------------------------------------------
-- field declaration is of:
--   1. boolean or integer
--   2. followed by an identifier (the field name).
strFieldDecl :: FieldDecl -> String
strFieldDecl (FieldDecl baseType fieldName) = (strBaseType baseType) ++ " " ++ fieldName

-- non-empty input list fieldDecls@(x:xs) according to parser
strFieldDecls :: [FieldDecl] -> String
-- first field decl starts with {
strFieldDecls (x:xs) = (addIndentation 1) ++ "{ " ++ (strFieldDecl x) ++ newline ++
-- rest start with ;
                       (concatMap
                          (\y -> (addIndentation 1) ++ "; " ++ (strFieldDecl y) ++ newline)
                          xs
                       )

-- convert record to string
-- A record type definition involving n fields should be printed on n + 2 lines,
-- as follows:
--     1. The first line contains the word record.
--     2. The remaining lines should be indented, with the first n containing one field declaration each
--        (the first preceded by a left brace and a single space, the rest preceded
--          by a semicolon and a single space),
--          see above strFieldDecls
--     3. and with the last line containing the record name, preceded by a right
--          brace and a single space, and followed by a semicolon;
strRecord :: Record -> String
strRecord (Record fieldDecls recordName) =
  "record" ++ newline ++
  (strFieldDecls fieldDecls) ++
  (addIndentation 1) ++ "} " ++ recordName ++ semicolon ++ newline


-----------------------------------------------------------------
--  Array toString function
-----------------------------------------------------------------
-- An array type definition should be printed on a single line.
-- It contains the word array,
-- followed by a positive integer in square brackets all without intervening
--      white space.
-- That string, the type, and the type alias, should be separated by single
--    spaces, and the whole line terminated by a semicolon.
strArray :: Array -> String
strArray (Array arraySize arrayType arrayName) =
  "array" ++ surroundByBrackets (show arraySize) ++ " " ++ (strDataType arrayType) ++ " " ++ arrayName ++


-----------------------------------------------------------------
--  Procedure toString functions
-----------------------------------------------------------------
strParameter :: Parameter -> String
strParameter (DataParameter dataType paraName) = (strDataType dataType) ++ " " ++ paraName
strParameter (BooleanVal paraName) = "boolean val " ++ paraName
strParameter (IntegerVal paraName) = "integer val " ++ paraName

-- The procedure head (that is, the keyword, procedure name, and list of formal
--     parameters) should be on a single line.
strProcedureHeader :: ProcedureHeader -> String
strProcedureHeader (ProcedureHeader procedureName parameters) =
  procedureName ++ " " ++ (surroundByParens (intercalate comma (map strParameter parameters)))

-- A variable declaration consists of
--   a) a type name (boolean, integer, or a type alias),
--   b) followed by a 1+ comma-separated list of
--        identifiers,
--     i)  the list terminated with a semicolon.
--     ii) There may be any number of variable declarations, in any order.
strVariableDecl :: VariableDecl -> String
```

```haskell
strVariableDecl (VariableDecl dataType varNames) =
  -- Within each procedure, declarations and top-level statements should be indented.
  (addIndentation 1) ++ (strDataType dataType) ++ " " ++ (intercalate comma varNames)  ++ semicolon ++
  -- Each variable declaration should be on a separate line.
  newline

-- variableDecls can be empty according to parser
-- stmts is non-empty according to parser
strProcedureBody :: ProcedureBody -> String
strProcedureBody (ProcedureBody variableDecls stmts) =
  (concatMap strVariableDecl variableDecls) ++
  -- The { and } that surround a procedure body should begin at the start of a
  --    line (no indentation).
  -- Moreover, these delimiters should appear alone, each making up a single line.
  "{" ++ newline ++
  -- Within each procedure, declarations and top-level statements should be indented.
  concatMap (strStmt 1) stmts ++
  "}" ++ newline

-- convert procedure to string
strProcedure :: Procedure -> String
-- The keyword procedure should begin at the start of a line (no indentation)
-- The procedure head (that is, the keyword, procedure name, and list of formal
--     parameters) should be on a single line.
strProcedure (Procedure ph pb) =
  "procedure " ++ (strProcedureHeader ph) ++ newline ++
  (strProcedureBody pb)


-----------------------------------------------------------------
--  Program toString function
-----------------------------------------------------------------
strProgram :: Program -> String
-- If there are no record and array type definitions, the first procedure should
--    start on line 1.
strProgram (Program [] [] procedures) = intercalate newline (map strProcedure procedures)
-- Otherwise there should be a single blank line between the type definitions
--    and the first procedure.
strProgram (Program records arraies procedures) =
  -- Each type definition should start on a new line, and there should be no
  --    blank lines between type definitions. So below two has no newline in between
  concatMap strRecord records ++
  concatMap strArray arraies ++
  newline ++
  -- Consecutive procedure definitions should be separated by a single blank line.
  intercalate newline (map strProcedure procedures)
```

Roo.hs

```haskell
-----------------------------------------------------------
-- COMP90045 Programming Language Implementation Project --
--                      Roo Compiler                     --
--  Implemented by Xulin Yang                            --
-----------------------------------------------------------
module Main (main)
where
import RooParser (ast)
import PrettyRoo (pp)
import System.Environment (getProgName, getArgs)
import System.Exit (exitWith, ExitCode(..))

data Task
  = Parse | Pprint
    deriving (Eq, Show)

main :: IO ()
main
  = do
      progname <- getProgName
      args <- getArgs
      task <- checkArgs progname args
      case task of
        Parse
          -> do
               let [_, filename] = args
               input <- readFile filename
               let output = ast input
               case output of
                 Right tree
                   -> putStrLn (show tree)
                 Left err
                   -> do putStrLn "Parse error at "
                         print err
                         exitWith (ExitFailure 2)
        Pprint
          -> do
               let [_, filename] = args
               input <- readFile filename
               let output = ast input
               case output of
                 Right tree
                   -> putStr (pp tree)
                 Left err
                   -> do putStrLn "Parse error at "
                         print err
                         exitWith (ExitFailure 2)

checkArgs :: String -> [String] -> IO Task
checkArgs _ ['-':_]
  = do
```

```
      putStrLn ("Missing filename")
      exitWith (ExitFailure 1)
checkArgs _ ["-a", filename]
  = return Parse
checkArgs _ ["-p", filename]
  = return Pprint
checkArgs progname _
  = do
      putStrLn ("Usage: " ++ progname ++ " [-p] filename")
      exitWith (ExitFailure 1)

```

RooAST.hs

```haskell
-----------------------------------------------------------
-- COMP90045 Programming Language Implementation Project --
--                     Roo Compiler                      --
--  Implemented by Xulin Yang                            --
--  read from the bottom to top                          --
-----------------------------------------------------------
module RooAST where

-----------------------------------
-- Terminology:
-- 0+: zero or more/possible empty
-- 1+: one or more/ non empty
--     both 0+, 1+ are stored in list [] but 1+ will be implemented in parser not here
-----------------------------------

-----------------------------------
-- Specification of an AST for Roo
-----------------------------------

-- Identifier: String
type Ident = String

-- Base type: boolean, integer type indicator
--     Not necessary to have string as no variable/parameter/declaration has string type
data BaseType
  = BooleanType
  | IntegerType
    deriving (Show, Eq)

-- A boolean literal is false or true.
type BooleanLiteral = Bool
-- An integer literal is a sequence of digits, only stores natural number in our parser implementation
type IntegerLiteral = Int
-- A string literal is a sequence of characters between double quotes.
--   The sequence itself cannot contain double quotes or newline/tab characters.
--   It may, however, contain '" ', '\n', and '\t', respectively, to represent
--   those characters.
type StringLiteral = String

-- User custermized record type, stored as string
type AliasType = String

-- for Array, VariableDecl: they have either boolean, integer, or a type alias data type
--     factored out for reuse purpose
data DataType
  = BasyDataType BaseType
  | AliasDataType AliasType
    deriving (Show, Eq)

-- An lvalue (<lvalue>) has four (and only four) possible forms:
--     An example lvalue is point[0].xCoord
```

```
data LValue
  = LId Ident                     -- <id>
  | LDot Ident Ident              -- <id>.<id>
  | LBrackets Ident Exp           -- <id>[<exp>]
  | LBracketsDot Ident Exp Ident  -- <id>[<exp>].<id>
    deriving (Show, Eq)

-- expression operators:
--     All the operators on the same line have the same precedence,
--         and the ones on later lines have lower precedence;
--     The six relational operators are non-associative
--         so, for example, a = b = c is not a well-formed expression).
--     The six remaining binary operators are left-associative.
-- -              /unary            /
-- * /            /binary and infix /left-associative
-- + -            /binary and infix /left-associative
-- = != < <= > >= /binary and infix /relational, non-associative
-- not            /unary            /
-- and            /binary and infix /left-associative
-- or             /binary and infix /left-associative
data Exp
  = Lval LValue               -- <lvalue>
  | BoolConst BooleanLiteral  -- <const> where <const> is the syntactic category of boolean, integer, and
  | IntConst IntegerLiteral
  | StrConst StringLiteral
                              -- ( <exp> ) is ignored here but handelled in parser
  | Op_or Exp Exp             -- <exp> <binop: or> <exp>
  | Op_and Exp Exp            -- <exp> <binop: and> <exp>
  | Op_eq  Exp Exp            -- <exp> <binop: "="> <exp>
  | Op_neq  Exp Exp           -- <exp> <binop: "!="> <exp>
  | Op_less  Exp Exp          -- <exp> <binop: "<"> <exp>
  | Op_less_eq  Exp Exp       -- <exp> <binop: "<="> <exp>
  | Op_large  Exp Exp         -- <exp> <binop: ">"> <exp>
  | Op_large_eq  Exp Exp      -- <exp> <binop: ">="> <exp>
  | Op_add  Exp Exp           -- <exp> <binop: "+"> <exp>
  | Op_sub  Exp Exp           -- <exp> <binop: "-"> <exp>
  | Op_mul  Exp Exp           -- <exp> <binop: "*"> <exp>
  | Op_div  Exp Exp           -- <exp> <binop: "/"> <exp>
  | Op_not Exp                -- <unop: not> <exp>
  | Op_neg Exp                -- <unop: "-"> <exp>
    deriving (Show, Eq)

-- Stmt has following two category:
--  1) atom statement:
--      <lvalue> <- <exp> ;
--      read <lvalue> ;
--      write <exp> ;
--      writeln <exp> ;
--      call <id>(<exp-list>) ;
--          where <exp-list> is a 0+ comma-separated list of expressions.
--  2) composite statement:
--      if <exp> then <stmt-list> else <stmt-list> fi
--      if <exp> then <stmt-list> fi # just make above second [Stmt] empty
--      while <exp> do <stmt-list> od
--          where <stmt-list> is a 1+ sequence of statements, atomic or composite
--
```

```haskell
-- the data structure for above grammer are given accordingly below
data Stmt
  -- 1) atom statement:
  = Assign LValue Exp
  | Read LValue
  | Write Exp
  | Writeln Exp
  | Call Ident [Exp]
  -- 2) composite statement:
  | If Exp [Stmt] [Stmt]
  | While Exp [Stmt]
    deriving (Show, Eq)

-- Each formal parameter has two components (in the given order):
--  1. a parameter type/mode indicator, which is one of these five:
--    a) a type alias,
--    b) boolean,
--    c) integer,
--    d) boolean val
--    e) integer val
--  2. an identifier
data Parameter
  = DataParameter DataType Ident -- a) b) c) above
  | BooleanVal Ident             -- d)       above
  | IntegerVal Ident             -- e)       above
    deriving (Show, Eq)

-- The header has two components (in this order):
--   1. an identifier (the procedure's name), and
--   2. a comma-separated list of 0+ formal parameters within a pair
--        of parentheses (so the parentheses are always present).
data ProcedureHeader
  = ProcedureHeader Ident [Parameter]
    deriving (Show, Eq)

-- A variable declaration consists of
--   a) a type name (boolean, integer, or a type alias),
--   b) followed by a 1+ comma-separated list of
--        identifiers,
--     i)  the list terminated with a semicolon.
--     ii) There may be any number of variable declarations, in any order.
data VariableDecl
  = VariableDecl DataType [Ident]
    deriving (Show, Eq)

-- procedure body consists of 0+ local variable declarations,
--   1. A variable declaration consists of
--     a) a type name (boolean, integer, or a type alias),
--     b) followed by a 1+ comma-separated list of identifiers,
--       i)  the list terminated with a semicolon.
--       ii) There may be any number of variable declarations, in any order.
--   2. followed by a 1+ sequence of statements,
data ProcedureBody
  = ProcedureBody [VariableDecl] [Stmt]
    deriving (Show, Eq)
```

```
-- Each procedure consists of (in the given order):
--   1. the keyword procedure,
--   2. a procedure header, and
--   3. a procedure body.
data Procedure
  = Procedure ProcedureHeader ProcedureBody
    deriving (Show, Eq)

-- array type definition consists of (in the given order):
--   1. the keyword array,
--   2. a (positive) integer literal enclosed in square brackets,
--   3. a type name which is either an identifier (a type alias) or one of
--      boolean and integer,
--   4. an identifier (giving a name to the array type), and
--   5. a semicolon.
data Array
  = Array IntegerLiteral DataType Ident
    deriving (Show, Eq)

-- field declaration is of:
--   1. boolean or integer
--   2. followed by an identifier (the field name).
data FieldDecl
  = FieldDecl BaseType Ident
    deriving (Show, Eq)

-- record consists of:
--   1. the keyword record,
--   2. a 1+ list of field declarations, separated by semicolons,
--        the whole list enclosed in braces,
--   3. an identifier, and
--   4. a semicolon.
data Record
  = Record [FieldDecl] Ident
    deriving (Show, Eq)

-- A Roo program consists of
--   1. 0+ record type definitions, followed by
--   2. 0+ array type definitions, followed by
--   3. 1+ procedure definitions.
data Program
  = Program [Record] [Array] [Procedure]
    deriving (Show, Eq)
```

===

RooParser.hs

===

```haskell
-----------------------------------------------------------
-- COMP90045 Programming Language Implementation Project --
--                     Roo Compiler                      --
--  Implemented by Xulin Yang                            --
--  read from the bottom to top                          --
-----------------------------------------------------------
module RooParser (ast)
where
import RooAST
import Text.Parsec
import Text.Parsec.Language (emptyDef)
import Text.Parsec.Expr
import qualified Text.Parsec.Token as Q
import System.Environment
import System.Exit
import Debug.Trace (trace)

type Parser a
   = Parsec String Int a

scanner :: Q.TokenParser Int
scanner
   = Q.makeTokenParser
     (emptyDef
     { Q.commentLine     = "#"
     , Q.nestedComments  = True
     , Q.identStart      = letter
     -- An identifir is a non-empty sequence of alphanumeric characters,
    --  underscore and apostrophe ('), and it must start with a (lower or upper case) letter.
     , Q.identLetter     = alphaNum <|> char '_' <|> char '\''
     , Q.opStart         = oneOf "+-*<"
     , Q.opLetter        = oneOf "="
     , Q.reservedNames   = joeyReserved
     , Q.reservedOpNames = joeyOpnames
     })

whiteSpace    = Q.whiteSpace scanner
lexeme        = Q.lexeme scanner
natural       = Q.natural scanner
identifier    = Q.identifier scanner
semi          = Q.semi scanner
comma         = Q.comma scanner
dot           = Q.dot scanner
parens        = Q.parens scanner
braces        = Q.braces scanner
brackets      = Q.brackets scanner
squares       = Q.squares scanner
reserved      = Q.reserved scanner
reservedOp    = Q.reservedOp scanner
stringLiteral = Q.stringLiteral scanner

```

```
joeyReserved, joeyOpnames :: [String]

-- reserved words according to the specification
joeyReserved
  = ["and", "array", "boolean", "call", "do", "else", "false", "fi", "if",
    "integer", "not", "od", "or", "procedure", "read", "record", "then",
    "true", "val", "while", "write", "writeln"]

-- reserved operators from specification
-- 12 binary oprator (and, or; above reserved string);
-- 2 unary: not (above reserved string), -;
-- assignment operator <-
joeyOpnames
  = [ "+", "-", "*", "/", "=", "!=", "<", "<=", ">", ">=", "<-"]

------------------------------------------------------------------
--  Note: 0+ is ensured using many/sepBy and 1+ using many1/sepBy1
------------------------------------------------------------------

------------------------------------------------------------------
--  parse reused base type integer/boolean; no string here as mentioned in AST
--  parse reused data type integer/boolean/type alias
------------------------------------------------------------------
pBaseType :: Parser BaseType
pBaseType
  = do
      reserved "boolean"
      return BooleanType
    <|>
    do
      reserved "integer"
      return IntegerType
    <?>
      "base type"

pDataType :: Parser DataType
pDataType
  =
    do
      baseType <- pBaseType
      return (BasyDataType baseType)
    <|>
    do
      alias <- identifier
      return (AliasDataType alias)
    <?>
      "data type"


------------------------------------------------------------------
--  parse literals
------------------------------------------------------------------
pBooleanLiteral :: Parser BooleanLiteral
pBooleanLiteral
 =
   do { reserved "true"; return (True) }
   <|>
   do { reserved "false"; return (False) }
   <?>
      "boolean literal"

pIntegerLiteral :: Parser IntegerLiteral
pIntegerLiteral
  =
    do
      n <- natural <?> "number"
      return (fromInteger n :: Int)
    <?>
      "Integer Literal"

-- don't accept newline, tab, quote but "\n", "\t", "\"" <- two character string should still be accepted
pcharacter :: Parser String
pcharacter
  =
    try(
      do
        string ('\\':['n'])
        return (['\\', 'n'])
    )
    <|>
    try(
      do
        string ('\\':['t'])
        return (['\\', 't'])
    )
    <|>
    try(
      do
        string ('\\':['"'])
        return (['\\', '"'])
    )
    <|>
    do
      c <- noneOf ['\n', '\t', '"']
      return ([c])
    <?>
      "any character except newline, tab, quote"

-- Parser for string
pString :: Parser String
pString
  =
    do
      -- String is surrounded by two quotes
      char '"'
      -- Parse characters except newline / tab characters and quotes
      str <- many pcharacter
      char '"' <?> "\'\"\' to wrap the string"
      spaces -- consumes following spaces
      return (concat str)
    <?>
      "string cannot has newline, quote, tab"
```

```

pStringLiteral :: Parser StringLiteral
pStringLiteral
  =
    do
      s <- pString
      return (s)
    <?>
      "string literal"


-----------------------------------------------------------------
-- An lvalue (<lvalue>) has four (and only four) possible forms:
--     <id>
--     <id>.<id>
--     <id>[<exp>]
--     <id>[<exp>].<id>
-- An example lvalue is point[0].xCoord
-----------------------------------------------------------------
pLValue :: Parser LValue
pLValue
  = try (
      do
        ident1 <- identifier
        exp <- brackets pExp
        dot
        ident2 <- identifier
        return (LBracketsDot ident1 exp ident2)
   )
    <|>
    try (
      do
        ident <- identifier
        exp <- brackets pExp
        return (LBrackets ident exp)
    )
    <|>
    try (
      do
        ident1 <- identifier
        dot
        ident2 <- identifier
        return (LDot ident1 ident2)
    )
    <|>
    do
      ident <- identifier
      return (LId ident)
    <?>
      "LValue"

-----------------------------------------------------------------
--  pExp is the main parser for expression.
--
--  It is built using Parces's powerful
--  buildExpressionParser and takes into account the operator
--  precedences and associativity specified in 'opTable' below.
-------------------------------------------------------------------
prefix name fun
  = Prefix (do { reservedOp name
               ; return fun
               }
           )

binary name op
  = Infix (do { reservedOp name
              ; return op
              }
          ) AssocLeft

relation name rel
  = Infix (do { reservedOp name
              ; return rel
              }
          ) AssocNone

-- expression operators:
--     All the operators on the same line have the same precedence,
--         and the ones on later lines have lower precedence;
--     The six relational operators are non-associative
--         so, for example, a = b = c is not a well-formed expression).
--     The six remaining binary operators are left-associative.
-- -              /unary            /
-- * /            /binary and infix /left-associative
-- + -            /binary and infix /left-associative
-- = != < <= > >= /binary and infix /relational, non-associative
-- not            /unary            /
-- and            /binary and infix /left-associative
-- or             /binary and infix /left-associative
opTable
  = [ [ prefix   "-"   Op_neg    ]
    , [ binary   "*"   Op_mul    , binary   "/"  Op_div  ]
    , [ binary   "+"   Op_add    , binary   "-"  Op_sub  ]
    , [ relation "="   Op_eq     , relation "!=" Op_neq  , relation "<"  Op_less
      , relation "<="  Op_less_eq, relation ">"  Op_large, relation ">=" Op_large_eq ]
    , [ prefix   "not" Op_not    ]
    , [ binary   "and" Op_and    ]
    , [ binary   "or"  Op_or     ]
    ]

pExp :: Parser Exp
pExp
  = buildExpressionParser opTable pFac
    <?>
      "expression"

pFac :: Parser Exp
pFac
  = choice [parens pExp, -- ( <exp> )
            pLval,
            pBoolConst,
            pIntConst,
            pStrConst,
            pNeg         -- used to parse expression like ------1 (arbitrary unary minus before expression
           ]
    <?>
      "simple expression"

pLval, pBoolConst, pIntConst, pStrConst :: Parser Exp
pLval
  =
    do
      lval <- pLValue
      return (Lval lval)
    <?>
      "lval expression"

pBoolConst
  =
    do
      b <- pBooleanLiteral
      return (BoolConst b)
    <?>
      "bool const/literal expression"

pIntConst
  =
    do
      i <- pIntegerLiteral
      return (IntConst i)
    <?>
      "int const/literal expression"

pStrConst
  =
    do
      s <- pStringLiteral
      return (StrConst s)
    <?>
      "string const/literal expression"

-- parse arbitrary unary minus in pFac
pNeg :: Parser Exp
pNeg
  =
  do
    reservedOp "-"
    exp <- pExp
    return (Op_neg exp)
  <?>
    "unary minus"


-----------------------------------------------------------------
--  pStmt is the main parser for statements.
--  Statement related parsers
-----------------------------------------------------------------
pStmt, pStmtAtom, pStmtComp :: Parser Stmt
```

```
-- atom statement:
--     <lvalue> <- <exp> ;
--     read <lvalue> ;
--     write <exp> ;
--     writeln <exp> ;
--     call <id> ( <exp-list> ) ;
--         where <exp-list> is a (possibly empty) comma-separated list of expressions.
-- composite statement:
--     if <expr> then <stmt-list> else <stmt-list> fi
--     if <exp> then <stmt-list> fi # just make second [Stmt] empty
--     while <expr> do <stmt-list> od
--         where <stmt-list> is a non-empty sequence of statements, atomic or composite
pStmt = choice [pStmtAtom, pStmtComp]

pStmtAtom
  =
    do
      r <- choice [pAsg, pRead, pWrite, pWriteln, pCall]
      -- all atomic stmt's semicolon is comsumed here
      semi
      return r
    <?>
      "atomic statement"

pAsg, pRead, pWrite, pWriteln, pCall :: Parser Stmt

-- parse: <lvalue> <- <exp> ;
pAsg
  =
    do
      lvalue <- pLValue
      reservedOp "<-"
      rvalue <- pExp
      return (Assign lvalue rvalue)
    <?>
      "assign"

-- parse: read <lvalue> ;
pRead
  = do
      reserved "read"
      lvalue <- pLValue
      return (Read lvalue)
    <?>
      "read"

-- parse: write <exp> ;
pWrite
  = do
      reserved "write"
      expr <- pExp
      return (Write expr)
    <?>
      "write"

-- parse: writeln <exp> ;
```

```
pWriteln
  = do
      reserved "writeln"
      expr <- pExp
      return (Writeln expr)
    <?>
      "writeln"

-- parse: call <id>(<exp-list>) ;
pCall
  = do
      reserved "call"
      ident <- identifier
      exprs <- parens (pExp `sepBy` comma) -- 0+ comma-separated list of expressions
      return (Call ident exprs)
    <?>
      "call"

pStmtComp = (choice [pIf, pWhile]) <?> "composite statement"

pIf, pWhile :: Parser Stmt

-- parse:
--    if <exp> then <stmt-list> else <stmt-list> fi
--    if <exp> then <stmt-list> fi # just make above second [Stmt] empty
pIf
  = do

      reserved "if"
      exp <- pExp
      reserved "then"
      stmts <- many1 pStmt
      -- check if there is an else statment
      -- if not, return empty
      estmts <- (
        do
          reserved "fi"
          return []
        <|>
        do
          reserved "else"
          -- else body can not be empty
          s <- many1 pStmt
          reserved "fi"
          return s
        )
      return (If exp stmts estmts)
    <?>
      "if"

-- parse: while <exp> do <stmt-list> od
pWhile
  = do
      reserved "while"
      exp <- pExp
      reserved "do"
      stmts <- many1 pStmt -- a 1+ sequence of statements, atomic or composite
      reserved "od"
      return (While exp stmts)
    <?>
      "while"

------------------------------------------------------------------
--  Procedure related parser
------------------------------------------------------------------
-- Each formal parameter has two components (in the given order):
--  1. a parameter type/mode indicator, which is one of these five:
--    a) a type alias,
--    b) boolean,
--    c) integer,
--    d) boolean val
--    e) integer val
--  2. an identifier
pParameter :: Parser Parameter
pParameter
  =
    try(
      do
        -- parse boolean val variable
        reserved "boolean"
        reserved "val"
        name <- identifier
        return (BooleanVal name)
    )
    <|>
    try(
      do
        -- parse integer val variable
        reserved "integer"
        reserved "val"
        name <- identifier
        return (IntegerVal name)
    )
    <|>
    do
      -- parse boolean/integer/type_alias variable
      paraType <- pDataType
      name <- identifier
      return (DataParameter paraType name)
    <?>
      "parameter"

-- The header has two components (in this order):
--   1. an identifier (the procedure's name), and
--   2. a comma-separated list of 0+ formal parameters within a pair
--        of parentheses (so the parentheses are always present).
pProcedureHeader :: Parser ProcedureHeader
pProcedureHeader
  =
    do
      procedureName <- identifier
      parameters <- parens (pParameter `sepBy` comma)
      return (ProcedureHeader procedureName parameters)
    <?>
      "procedure header"

-- A variable declaration consists of
--   a) a type name (boolean, integer, or a type alias),
--   b) followed by a 1+ comma-separated list of
--        identifiers,
--     i)  the list terminated with a semicolon.
--     ii) There may be any number of variable declarations, in any order.
pVariable :: Parser VariableDecl
pVariable
  =
    do
      varType <- pDataType
      varNames <- (identifier `sepBy1` comma)
      semi
      return (VariableDecl varType varNames)
    <?>
      "variable"


-- procedure body consists of 0+ local variable declarations,
--   1. A variable declaration consists of
--     a) a type name (boolean, integer, or a type alias),
--     b) followed by a 1+ comma-separated list of identifiers,
--       i)  the list terminated with a semicolon.
--       ii) There may be any number of variable declarations, in any order.
--   2. followed by a 1+ sequence of statements,
pProcedureBody :: Parser ProcedureBody
pProcedureBody
  =
    do
      vars <- many pVariable
      stmts <- braces (many1 pStmt)
      return (ProcedureBody vars stmts)
    <?>
      "procedure body"

-- Each procedure consists of (in the given order):
--   1. the keyword procedure,
--   2. a procedure header, and
--   3. a procedure body.
pProcedure :: Parser Procedure
pProcedure
  =
    do
      reserved "procedure"
      procedureHeader <- pProcedureHeader
      procedureBody <- pProcedureBody
      return (Procedure procedureHeader procedureBody)
    <?>
      "procedure"

-----------------------------------------------------------------
--  Array related parser
```

```haskell
-------------------------------------------------------------------
-- array type definition consists of (in the given order):
--   1. the keyword array,
--   2. a (positive) integer literal enclosed in square brackets,
--   3. a type name which is either an identifier (a type alias) or one of
--      boolean and integer,
--   4. an identifier (giving a name to the array type), and
--   5. a semicolon.
pArray :: Parser Array
pArray
  =
    do
      reserved "array"
      pos <- getPosition
      arraySize <- brackets pIntegerLiteral
      -- need to check arraySize > 0 (positive integer)
      if arraySize == 0
      then
        error ("array size sould not be 0 at line: " ++ (show (sourceLine pos)) ++ ", column: " ++ (show (
      else do
        arrayType <- pDataType
        arrayName <- identifier
        semi
        return (Array arraySize arrayType arrayName)
      <?>
        "array"

-------------------------------------------------------------------
--  Record related parser
-------------------------------------------------------------------
-- field declaration is of:
--   1. boolean or integer
--   2. followed by an identifier (the field name).
pFieldDecl :: Parser FieldDecl
pFieldDecl
  =
    do
      fieldType <- pBaseType
      fieldName <- identifier
      return (FieldDecl fieldType fieldName)
    <?>
      "field declaration"

-- record consists of:
--   1. the keyword record,
--   2. a 1+ list of field declarations, separated by semicolons,
--        the whole list enclosed in braces,
--   3. an identifier, and
--   4. a semicolon.
pRecord :: Parser Record
pRecord
  =
    do
      reserved "record"
      recordFieldDecls <- braces (pFieldDecl `sepBy1` semi)
      recordName <- identifier
      semi
      return (Record recordFieldDecls recordName)
    <?>
      "record"

-----------------------------------------------------------------
--  Program related parser
-----------------------------------------------------------------
-- A Roo program consists of
--   1. 0+ record type definitions, followed by
--   2. 0+ array type definitions, followed by
--   3. 1+ procedure definitions.
pProgram :: Parser Program
pProgram
  = do
      records <- many pRecord
      arraies <- many pArray
      procedures <- many1 pProcedure
      return (Program records arraies procedures)
    <?>
      "program"


-----------------------------------------------------------------
-- main (given skeleton code)
-----------------------------------------------------------------

joeyParse :: Parser Program
joeyParse
  = do
      whiteSpace
      p <- pProgram
      eof
      return p

ast :: String -> Either ParseError Program
ast input
  =  runParser joeyParse 0 "" input

pMain :: Parser Program
pMain
  = do
      whiteSpace
      p <- pProgram
      eof
      return p

main :: IO ()
main
  = do { progname <- getProgName
       ; args <- getArgs
       ; checkArgs progname args
       ; input <- readFile (head args)
       ; let output = runParser pMain 0 "" input
       ; case output of
           Right ast -> print ast
           Left  err -> do { putStr "Parse error at "
                           ; print err
                           }
       }

checkArgs :: String -> [String] -> IO ()
checkArgs _ [filename]
   = return ()
checkArgs progname _
   = do { putStrLn ("Usage: " ++ progname ++ " filename\n\n")
       ; exitWith (ExitFailure 1)
       }
```