==

Codegen.hs

==

```haskell
1   ----------------------------------------------------------
2   -- COMP90045 Programming Language Implementation Project --
3   --                  Roo Compiler                         --
4   --   Implemented by Xulin Yang, Wenrui Zhang            --
5   --   Implemented by Team: GNU_project                   --
6   ----------------------------------------------------------
7   module Codegen(ozCode, Consequence(..)) where
8
9   import OzCode
10  import Control.Monad
11  import Control.Monad.State
12  import Control.Monad.Except
13  import RooAST
14  import SymbolTable
15  import Data.Map (Map, (!))
16  import qualified Data.Map as Map
17  import Data.Either
18
19  type Consequence = Either String [OzInstruction]
20
21  data Btype = Int
22             | Bool
23             | String
24             | BRecord String
25
26  ozCode :: SymTable -> Program -> Consequence
27  ozCode st prog = evalStateT (codeGeneration prog) st
28
29  -- transfer the Roo program into oz instructions
30  codeGeneration :: Program -> SymTableState [OzInstruction]
31  codeGeneration (Program _ _ procedures)
32    =
33      do
34        let generatedCode = [ProcedureInstruction $ ICall "proc_main",
35                             ProcedureInstruction $ IHalt]
36
37        mapM_ appendInstruction generatedCode
38        mapM_ generateProcedure procedures
39
40        st <- get
41        return $ instructions st
42
43  -- generate oz instructions for the given procedure
44  generateProcedure :: Procedure -> SymTableState ()
45  generateProcedure p@(Procedure (ProcedureHeader procID params)
46                                 (ProcedureBody _ stmts))
47    =
48      do
49        appendInstruction (Label $ "proc_" ++ procID)
50        pushLocalVariableTable
51        insertProcedureVariable p
```

```haskell
52
53          -- generate code of procedure's statements
54          appendInstruction (Comment "prologue")
55          slotNum <- getSlotCounter
56          if slotNum /= 0 then
57            do
58              appendInstruction (StackInstruction $ PushStackFrame slotNum)
59          else
60            return ()
61
62          -- load parameters
63          let paraNum = length params
64          appendInstruction (Comment "load parameters")
65          mapM_ (\i ->
66                do
67                    appendInstruction(StackInstruction $ Store i i)
68                ) [0..(paraNum -1)]
69
70          -- init variables
71          reg_init <- getRegisterCounter
72          appendInstruction (Comment "initialise variables")
73          appendInstruction (ConstantInstruction $ OzIntConst reg_init 0)
74          mapM_ (\i ->
75                do
76                    appendInstruction (StackInstruction $ Store i reg_init)
77                ) [paraNum..(slotNum -1)]
78          setRegisterCounter reg_init
79
80          mapM_ generateStatement stmts
81
82          -- end of the procedure
83          appendInstruction (Comment "epilogue")
84          if slotNum /= 0 then
85            do
86              appendInstruction (StackInstruction $ PopStackFrame slotNum)
87          else
88            return ()
89
90          appendInstruction (ProcedureInstruction IReturn)
91
92          popLocalVariableTable
93
94  -- ------------------------------------------------------------------------------
95  -- Generate Statement
96  -- ------------------------------------------------------------------------------
97
98  generateStatement :: Stmt -> SymTableState ()
99  generateStatement (Assign (LId lId) (Lval rValue))
100    =
101      do
102        (_, _, lVarType, lTotalSlot) <- getVariableType lId
103        appendInstruction (Comment $ show lId ++ " <- " ++ show rValue)
104        case lVarType of
105          -- assign the array by array reference
106          (ArrayVar _) ->
107              do
```

2

```
108            assignEle lTotalSlot (LId lId) rValue
109        -- assign the record by record reference
110        (RecordVar _)->
111          do
112            assignEle lTotalSlot (LId lId) rValue
113        -- other case
114        _ ->
115          do
116            reg <- getRegisterCounter
117            loadExp reg (Lval rValue)
118            storeVal reg (LId lId)
119            setRegisterCounter reg
120 generateStatement (Assign (LBrackets lId lexp) (Lval rValue))
121    =
122      do
123        (_, _, lVarType, lTotalSlot) <- getVariableType lId
124        appendInstruction (Comment $ show (LBrackets lId lexp)
125          ++ " <- " ++ show rValue)
126        case lVarType of
127          (ArrayVar alias) ->
128            do
129              (size, _) <- getArrayType alias
130              assignEle (div lTotalSlot size) (LBrackets lId lexp) rValue
131          _ -> liftEither $ throwError $ "Expect Array as type"
132 generateStatement (Assign lValue exp)
133    =
134      do
135        appendInstruction (Comment $ show lValue ++ " <- " ++ show exp)
136        reg <- getRegisterCounter
137        loadExp reg exp
138        storeVal reg lValue
139        setRegisterCounter reg
140 generateStatement (Read lValue)
141    =
142      do
143        appendInstruction (Comment $ "Read " ++ show lValue)
144        bType <- getType $ Lval lValue
145        reg <- getRegisterCounter
146        let cmd = case bType of Int -> "read_int"
147                               Bool -> "read_bool"
148                               String -> "read_string"
149        appendInstruction (ProcedureInstruction $ ICallBuiltIn cmd)
150        storeVal reg lValue
151        setRegisterCounter reg
152 generateStatement (Write exp)
153    =
154      do
155        appendInstruction (Comment $ "Write " ++ show exp)
156        bType <- getType exp
157        reg <- getRegisterCounter
158        loadExp reg exp
159        let cmd = case bType of Int -> "print_int"
160                               Bool -> "print_bool"
161                               String -> "print_string"
162        appendInstruction (ProcedureInstruction $ ICallBuiltIn cmd)
163        setRegisterCounter reg
```

```
164  generateStatement (Writeln exp)
165      =
166        do
167          appendInstruction (Comment $ "Writeln " ++ show exp)
168          bType <- getType exp
169          reg <- getRegisterCounter
170          loadExp reg exp
171          let cmd = case bType of Int -> "print_int"
172                                  Bool -> "print_bool"
173                                  String -> "print_string"
174          appendInstruction (ProcedureInstruction $ ICallBuiltIn cmd)
175          appendInstruction (ProcedureInstruction $ ICallBuiltIn "print_newline")
176          setRegisterCounter reg
177  generateStatement (Call procID params)
178      =
179        do
180          appendInstruction (Comment $ "Call " ++ show procID)
181          let paraNum = length params
182          (formalParams, _) <- getProcedure procID
183          mapM_ (\i ->
184                  do
185                      reg <- getRegisterCounter
186                      let (byValue, _) = formalParams !! i
187                      case byValue of
188                          True -> loadExp reg $ params !! i
189                          False -> case (params !! i) of
190                                        Lval lValue -> loadVarAddress reg lValue
191                                        _ -> return ()
192                ) [0..(paraNum - 1)]
193          appendInstruction (ProcedureInstruction $ ICall $ "proc_" ++ procID)
194          setRegisterCounter 0
195  generateStatement (IfThen exp stmts)
196      =
197        do
198          trueLabel <- getlabelCounter
199          falseLabel <- getlabelCounter
200          appendInstruction (Comment $ "if " ++ show(exp))
201
202          -- guard
203          reg <- getRegisterCounter
204          loadExp reg exp
205          appendInstruction (BranchInstruction $ Cond False reg falseLabel)
206          setRegisterCounter reg
207
208          -- then
209          appendInstruction (Label trueLabel)
210          appendInstruction (Comment $ "then")
211          mapM_ generateStatement stmts
212          appendInstruction (Comment $ "fi")
213
214          -- after if then
215          appendInstruction (Label falseLabel)
216  generateStatement (IfThenElse exp stmts1 stmts2)
217      =
218        do
219          trueLabel <- getlabelCounter
```

```
220        falseLabel <- getlabelCounter
221        endLabel <- getlabelCounter
222        appendInstruction (Comment $ "if " ++ show(exp))
223
224        -- guard
225        reg <- getRegisterCounter
226        loadExp reg exp
227        appendInstruction (BranchInstruction $ Cond False reg falseLabel)
228        setRegisterCounter reg
229
230        -- then
231        appendInstruction (Label trueLabel)
232        appendInstruction (Comment $ "then")
233        mapM_ generateStatement stmts1
234        appendInstruction (BranchInstruction $ Uncond endLabel)
235
236        -- else
237        appendInstruction (Label falseLabel)
238        appendInstruction (Comment $ "else")
239        mapM_ generateStatement stmts2
240        appendInstruction (Comment $ "fi")
241
242        -- after if then else
243        appendInstruction (Label endLabel)
244 generateStatement (While exp stmts)
245    =
246      do
247        trueLabel <- getlabelCounter
248        falseLabel <- getlabelCounter
249        appendInstruction (Comment $ "While " ++ show(exp))
250        appendInstruction (Label trueLabel)
251
252        -- guard
253        reg <- getRegisterCounter
254        loadExp reg exp
255        appendInstruction (BranchInstruction $ Cond False reg falseLabel)
256        setRegisterCounter reg
257
258        -- whileloop body
259        appendInstruction (Comment $ "do")
260        mapM_ generateStatement stmts
261        appendInstruction (BranchInstruction $ Uncond trueLabel)
262        appendInstruction (Comment $ "od")
263
264        -- after loop
265        appendInstruction (Label falseLabel)
266
267 -- load an expression to the given register
268 loadExp :: Int -> Exp -> SymTableState ()
269 loadExp reg (Lval lValue) = loadVal reg lValue
270 loadExp reg (BoolConst vl)
271   = appendInstruction (ConstantInstruction $ OzIntConst reg $ boolToInt vl)
272 loadExp reg (IntConst vl)
273   = appendInstruction (ConstantInstruction $ OzIntConst reg vl)
274 loadExp reg (StrConst vl)
275   = appendInstruction (ConstantInstruction
```

5

```
276           $ OzStringConst reg (strReplace vl))
277 loadExp reg (Op_or lExp rExp)
278    =
279      do
280        loadExp reg lExp
281        reg_1 <- getRegisterCounter
282        loadExp reg_1 rExp
283        appendInstruction (LogicInstruction $ LogicOr reg reg reg_1)
284        setRegisterCounter reg_1
285 loadExp reg (Op_and lExp rExp)
286    =
287      do
288        loadExp reg lExp
289        reg_1 <- getRegisterCounter
290        loadExp reg_1 rExp
291        appendInstruction (LogicInstruction $ LogicAnd reg reg reg_1)
292        setRegisterCounter reg_1
293 loadExp reg (Op_eq lExp rExp)
294    =
295      do
296        loadExp reg lExp
297        reg_1 <- getRegisterCounter
298        loadExp reg_1 rExp
299        appendInstruction (ComparisonInstruction
300            $ CmpInstruction Eq OpInt reg reg reg_1)
301        setRegisterCounter reg_1
302 loadExp reg (Op_neq lExp rExp)
303    =
304      do
305        loadExp reg lExp
306        reg_1 <- getRegisterCounter
307        loadExp reg_1 rExp
308        appendInstruction (ComparisonInstruction
309            $ CmpInstruction Ne OpInt reg reg reg_1)
310        setRegisterCounter reg_1
311 loadExp reg (Op_less lExp rExp)
312    =
313      do
314        loadExp reg lExp
315        reg_1 <- getRegisterCounter
316        loadExp reg_1 rExp
317        appendInstruction (ComparisonInstruction
318            $ CmpInstruction Lt OpInt reg reg reg_1)
319        setRegisterCounter reg_1
320 loadExp reg (Op_less_eq lExp rExp)
321    =
322      do
323        loadExp reg lExp
324        reg_1 <- getRegisterCounter
325        loadExp reg_1 rExp
326        appendInstruction (ComparisonInstruction
327            $ CmpInstruction Le OpInt reg reg reg_1)
328        setRegisterCounter reg_1
329 loadExp reg (Op_large lExp rExp)
330    =
331      do
```

```
332        loadExp reg lExp
333        reg_1 <- getRegisterCounter
334        loadExp reg_1 rExp
335        appendInstruction (ComparisonInstruction
336            $ CmpInstruction Gt OpInt reg reg reg_1)
337        setRegisterCounter reg_1
338    loadExp reg (Op_large_eq lExp rExp)
339      =
340        do
341        loadExp reg lExp
342        reg_1 <- getRegisterCounter
343        loadExp reg_1 rExp
344        appendInstruction (ComparisonInstruction
345            $ CmpInstruction Ge OpInt reg reg reg_1)
346        setRegisterCounter reg_1
347    loadExp reg (Op_add lExp rExp)
348      =
349        do
350        loadExp reg lExp
351        reg_1 <- getRegisterCounter
352        loadExp reg_1 rExp
353        appendInstruction (ArithmeticInstruction
354            $ Add OpInt reg reg reg_1)
355        setRegisterCounter reg_1
356    loadExp reg (Op_sub lExp rExp)
357      =
358        do
359        loadExp reg lExp
360        reg_1 <- getRegisterCounter
361        loadExp reg_1 rExp
362        appendInstruction (ArithmeticInstruction
363            $ Sub OpInt reg reg reg_1)
364        setRegisterCounter reg_1
365    loadExp reg (Op_mul lExp rExp)
366      =
367        do
368        loadExp reg lExp
369        reg_1 <- getRegisterCounter
370        loadExp reg_1 rExp
371        appendInstruction (ArithmeticInstruction
372            $ Mul OpInt reg reg reg_1)
373        setRegisterCounter reg_1
374    loadExp reg (Op_div lExp rExp)
375      =
376        do
377        loadExp reg lExp
378        reg_1 <- getRegisterCounter
379        loadExp reg_1 rExp
380        appendInstruction (ArithmeticInstruction
381            $ Div OpInt reg reg reg_1)
382        setRegisterCounter reg_1
383    loadExp reg (Op_not exp)
384      =
385        do
386        loadExp reg exp
387        appendInstruction (LogicInstruction
```

```haskell
388              $ LogicNot reg reg)
389 loadExp reg (Op_neg exp)
390    =
391      do
392        loadExp reg exp
393        appendInstruction (ArithmeticInstruction
394            $ Neg OpInt reg reg)
395
396 loadVal :: Int -> LValue -> SymTableState ()
397 loadVal reg lValue
398    =
399      do
400        loadVarAddress reg lValue
401        appendInstruction (StackInstruction $ LoadIndirect reg reg)
402
403 -- store a left value to the given register
404 storeVal :: Int -> LValue -> SymTableState ()
405 storeVal reg lValue
406    =
407      do
408        reg_1 <- getRegisterCounter
409        loadVarAddress reg_1 lValue
410        appendInstruction (StackInstruction $ StoreIndirect reg_1 reg)
411        setRegisterCounter reg_1
412
413 -- load a left value from the given register
414 loadVarAddress :: Int -> LValue -> SymTableState ()
415 loadVarAddress reg (LId ident)
416    =
417      do
418        (byValue, slotNum, _, _) <- getVariableType ident
419        if byValue then
420        -- load the address of the slot use loadAddress
421          appendInstruction (StackInstruction $ LoadAddress reg slotNum)
422        -- load the address directly as the slot store reference(address)
423        else
424          appendInstruction (StackInstruction $ Load reg slotNum)
425
426 loadVarAddress reg (LBrackets arrayID exp)
427    =
428      do
429        loadExp reg exp
430        reg_1 <- getRegisterCounter
431        loadVarAddress reg_1 (LId arrayID)
432        (_, _, varType, totalSlot) <- getVariableType arrayID
433        case varType of
434          (ArrayVar alias) ->
435            do
436              (size, _) <- getArrayType alias
437              reg_2 <- getRegisterCounter
438              appendInstruction (ConstantInstruction
439                  $ OzIntConst reg_2 $ div totalSlot size)
440              appendInstruction (ArithmeticInstruction
441                  $ Mul OpInt reg reg reg_2)
442              appendInstruction (ArithmeticInstruction
443                  $ SubOff reg reg_1 reg)
```

```
444        _ -> liftEither $ throwError $ "Expect Array as type"
445      setRegisterCounter reg_1
446
447  loadVarAddress reg (LDot recordID fieldID)
448    =
449      do
450        loadVarAddress reg (LId recordID)
451        (_, _, varType, _) <- getVariableType recordID
452        case varType of
453          (RecordVar alias) ->
454            do
455              (_, offset) <- getRecordField alias fieldID
456              reg_1 <- getRegisterCounter
457              appendInstruction (ConstantInstruction
458                  $ OzIntConst reg_1 $ offset)
459              appendInstruction (ArithmeticInstruction
460                  $ SubOff reg reg reg_1)
461              setRegisterCounter reg_1
462          _ -> liftEither $ throwError $ "Expect Record as type"
463
464  loadVarAddress reg (LBracketsDot arrayID exp fieldID)
465    =
466      do
467        loadVarAddress reg (LBrackets arrayID exp)
468        (_, _, varType, _) <- getVariableType arrayID
469        case varType of
470          (ArrayVar alias) ->
471            do
472              (size, dataType) <- getArrayType alias
473              case dataType of
474                (AliasDataType recordName) ->
475                  do
476                    (_, offset) <- getRecordField recordName fieldID
477                    reg_1 <- getRegisterCounter
478                    appendInstruction (ConstantInstruction
479                        $ OzIntConst reg_1 $ offset)
480                    appendInstruction (ArithmeticInstruction
481                        $ SubOff reg reg reg_1)
482                    setRegisterCounter reg_1
483                _ -> liftEither $ throwError
484                      $ "Expect record as type"
485          _ -> liftEither $ throwError $ "Expect Array as type"
486
487  -- ---------------------------------------------------------------------------
488  -- Help functions
489  -- ---------------------------------------------------------------------------
490  getType :: Exp -> SymTableState (Btype)
491  getType (BoolConst _) = return Bool
492  getType (IntConst _) = return Int
493  getType (StrConst _) = return String
494  getType (Op_or _ _) = return Bool
495  getType (Op_and _ _) = return Bool
496  getType (Op_eq _ _) = return Bool
497  getType (Op_neq _ _) = return Bool
498  getType (Op_less _ _) = return Bool
499  getType (Op_less_eq _ _) = return Bool
```

```
500  getType (Op_large _ _) = return Bool
501  getType (Op_large_eq _ _) = return Bool
502  getType (Op_add _ _) = return Int
503  getType (Op_sub _ _) = return Int
504  getType (Op_mul _ _) = return Int
505  getType (Op_div _ _) = return Int
506  getType (Op_not _) = return Bool
507  getType (Op_neg _) = return Int
508
509  getType (Lval (LId ident))
510      =
511        do
512          (_, _, varType, _) <- getVariableType ident
513          let result = case varType of BooleanVar -> Bool
514                                       IntegerVar -> Int
515          return result
516
517  getType (Lval (LBrackets arrayID _))
518      =
519        do
520          (_, _, varType, _) <- getVariableType arrayID
521          case varType of
522            (ArrayVar alias) ->
523              do
524                (_, dataType) <- getArrayType alias
525                case dataType of
526                  (BaseDataType BooleanType) -> return Bool
527                  (BaseDataType IntegerType) -> return Int
528                  (AliasDataType alias) -> return (BRecord alias)
529                  _ -> liftEither $ throwError $ "Expect Int/Bool"
530            _ -> liftEither $ throwError $ "Expect Array as type"
531
532  getType (Lval (LDot recordID fieldID))
533      =
534        do
535          (_, _, varType, _) <- getVariableType recordID
536          case varType of
537            (RecordVar alias) ->
538              do
539                (baseType, _) <- getRecordField alias fieldID
540                case baseType of
541                  BooleanType -> return Bool
542                  IntegerType -> return Int
543                  _ -> liftEither $ throwError $ "Expect Int/Bool"
544            _ -> liftEither $ throwError $ "Expect Record as type"
545
546  getType (Lval (LBracketsDot arrayID exp fieldID))
547      =
548        do
549          varType <- getType (Lval (LBrackets arrayID exp))
550          case varType of
551            (BRecord alias) ->
552              do
553                (baseType, _) <- getRecordField alias fieldID
554                case baseType of
555                  BooleanType -> return Bool
```

```haskell
              IntegerType -> return Int
                _ -> liftEither $ throwError $ "Expect Int/Bool"
          _ -> liftEither $ throwError $ "Expect Record as type"


boolToInt :: Bool -> Int
boolToInt True = 1
boolToInt False = 0


-- assign the element in the array or record one by one
assignEle :: Int -> LValue -> LValue -> SymTableState ()
assignEle rTotalSlot lValue rValue
  =
    do
      mapM_ (\n ->
        do
          reg <- getRegisterCounter
          loadVarAddress reg rValue
          reg_1 <- getRegisterCounter
          loadVarAddress reg_1 lValue
          reg_2 <- getRegisterCounter
          appendInstruction (ConstantInstruction
              $ OzIntConst reg_2 n)
          appendInstruction (ArithmeticInstruction
              $ SubOff reg reg reg_2)
          appendInstruction (StackInstruction
              $ LoadIndirect reg reg)
          appendInstruction (ArithmeticInstruction
              $ SubOff reg_1 reg_1 reg_2)
          appendInstruction (StackInstruction
              $ StoreIndirect reg_1 reg)
          setRegisterCounter reg
        )[0..(rTotalSlot -1)]
```

```
1   ---------------------------------------------------------------
2   -- COMP90045 Programming Language Implementation Project --
3   --                   Roo Compiler                        --
4   --   Implemented by Xulin Yang, Wenrui Zhang            --
5   --   Implemented by Team: GNU_project                   --
6   ---------------------------------------------------------------
7
8   module OzCode where
9
10  import Data.Char
11
12  type Register = Int
13
14  -- ----------------------------------------------------------------------------
15  -- define the oz instruction
16  -- ----------------------------------------------------------------------------
17
18  data StackInstruction
19    = PushStackFrame Int
20    | PopStackFrame Int
21    | Store Int Register
22    | Load Register Int
23    | LoadAddress Register Int
24    | LoadIndirect Register Register
25    | StoreIndirect Register Register
26    deriving (Eq)
27
28  data ConstantInstruction
29    = OzIntConst Register Int
30    | OzRealConst Register Float
31    | OzStringConst Register String
32    deriving (Eq)
33
34  data OpType
35    = OpInt
36    | OpReal
37    deriving (Eq)
38
39  data ArithmeticInstruction
40    = Add OpType Register Register Register
41    | AddOff Register Register Register
42    | Sub OpType Register Register Register
43    | SubOff Register Register Register
44    | Mul OpType Register Register Register
45    | Div OpType Register Register Register
46    | Neg OpType Register Register
47    deriving (Eq)
48
49  data ComparisonOperator
50    = Eq | Ne | Gt | Ge | Lt | Le
51    deriving (Eq)
```

```haskell
52
53   data ComparisonInstruction
54     = CmpInstruction ComparisonOperator OpType Register Register Register
55     deriving (Eq)
56
57   data LogicInstruction
58     = LogicAnd Register Register Register
59     | LogicOr Register Register Register
60     | LogicNot Register Register
61     deriving (Eq)
62
63   data OperationInstruction
64     = IntToReal Register Register
65     | Move Register Register
66     deriving (Eq)
67
68   data BranchInstruction
69     = Cond Bool Register String
70     | Uncond String
71     deriving (Eq)
72
73   data ProcedureInstruction
74     = ICall String
75     | ICallBuiltIn String
76     | IReturn
77     | IHalt
78     deriving (Eq)
79
80   data DebugInstruction
81     = DebugReg Register
82     | DebugSlot Int
83     | DebugStack
84     deriving (Eq)
85
86   data OzInstruction
87     = StackInstruction StackInstruction
88     | ArithmeticInstruction ArithmeticInstruction
89     | ComparisonInstruction ComparisonInstruction
90     | LogicInstruction LogicInstruction
91     | ConstantInstruction ConstantInstruction
92     | OperationInstruction OperationInstruction
93     | BranchInstruction BranchInstruction
94     | ProcedureInstruction ProcedureInstruction
95     | DebugInstruction DebugInstruction
96     | Comment String
97     | Label String
98     deriving (Eq)
99
100  -- ------------------------------------------------------------------------------
101  -- define the oz code format
102  -- ------------------------------------------------------------------------------
103
104  instance Show StackInstruction where
105    show (PushStackFrame size) = "push_stack_frame " ++ show (size)
106    show (PopStackFrame size) = "pop_stack_frame " ++ show (size)
107    show (Store slotnum register) = "store " ++ show (slotnum)
```

13

```
108        ++ ", r" ++ show (register)
109    show (Load register slotnum) = "load r" ++ show (register) ++ ", "
110        ++ show (slotnum)
111    show (LoadAddress register slotnum) = "load_address r"
112        ++ show (register) ++ ", "++ show (slotnum)
113    show (LoadIndirect register1 register2) = "load_indirect r"
114        ++ show (register1) ++ ", r" ++ show (register2)
115    show (StoreIndirect register1 register2) = "store_indirect r"
116        ++ show (register1) ++ ", r" ++ show (register2)
117
118 instance Show ConstantInstruction where
119    show (OzIntConst register int) = "int_const r" ++ show (register)
120        ++ ", " ++ show (int)
121    show (OzRealConst register float) = "real_const r" ++ show (register)
122        ++ ", " ++ show (float)
123    show (OzStringConst register string) = "string_const r" ++ show (register)
124        ++ ", " ++ show (string)
125
126 instance Show OpType where
127    show OpInt = "int"
128    show OpReal = "real"
129
130 instance Show ArithmeticInstruction where
131    show (Add opType r1 r2 r3) = "add_" ++ show (opType) ++ " r"
132        ++ show (r1) ++ ", r" ++ show (r2) ++ ", r" ++ show (r3)
133    show (AddOff r1 r2 r3) = "add_offset r"
134        ++ show (r1) ++ ", r" ++ show (r2) ++ ", r" ++ show (r3)
135    show (Sub opType r1 r2 r3) = "sub_" ++ show (opType) ++ " r"
136        ++ show (r1) ++ ", r" ++ show (r2) ++ ", r" ++ show (r3)
137    show (SubOff r1 r2 r3) = "sub_offset r"
138        ++ show (r1) ++ ", r" ++ show (r2) ++ ", r" ++ show (r3)
139    show (Mul opType r1 r2 r3) = "mul_" ++ show (opType) ++ " r"
140        ++ show (r1) ++ ", r" ++ show (r2) ++ ", r" ++ show (r3)
141    show (Div opType r1 r2 r3) = "div_" ++ show (opType) ++ " r"
142        ++ show (r1) ++ ", r" ++ show (r2) ++ ", r" ++ show (r3)
143    show (Neg opType r1 r2) = "neg_" ++ show (opType) ++ " r"
144        ++ show (r1) ++ ", r" ++ show (r2)
145
146 instance Show ComparisonOperator where
147    show Eq = "cmp_eq_"
148    show Ne = "cmp_ne_"
149    show Gt = "cmp_gt_"
150    show Ge = "cmp_ge_"
151    show Lt = "cmp_lt_"
152    show Le = "cmp_le_"
153
154 instance Show ComparisonInstruction where
155    show (CmpInstruction comparisonOp opType r1 r2 r3) =
156        show (comparisonOp) ++ show (opType) ++ " r"
157        ++ show (r1) ++ ", r" ++ show (r2) ++ ", r" ++ show (r3)
158
159 instance Show LogicInstruction where
160    show (LogicAnd r1 r2 r3) = "and r" ++ show (r1)
161        ++ ", r" ++ show (r2) ++ ", r" ++ show (r3)
162    show (LogicOr r1 r2 r3) = "or r" ++ show (r1)
163        ++ ", r" ++ show (r2) ++ ", r" ++ show (r3)
```

```haskell
164    show (LogicNot r1 r2) = "not r" ++ show (r1) ++ ", r" ++ show (r2)

166  instance Show OperationInstruction where
167    show (IntToReal r1 r2) = "int_to_real r" ++ show (r1) ++ ", r" ++ show (r2)
168    show (Move r1 r2) = "move r" ++ show (r1) ++ ", r" ++ show (r2)

170  instance Show BranchInstruction where
171    show (Cond bool register label) = "branch_on_" ++ map toLower (show (bool))
172      ++ " r" ++ show (register) ++ ", " ++ id (label)
173    show (Uncond label) = "branch_uncond " ++ id (label)

175  instance Show ProcedureInstruction where
176    show (ICall label) = "call " ++ id (label)
177    show (ICallBuiltIn func) = "call_builtin " ++ id (func)
178    show (IReturn) = "return"
179    show (IHalt) = "halt"

181  instance Show DebugInstruction where
182    show (DebugReg register) = "debug_reg r" ++ show (register)
183    show (DebugSlot slotNum) = "debug_slot " ++ show (slotNum)
184    show (DebugStack) = "debug_stack"

186  instance Show OzInstruction where
187    show (StackInstruction instruction) = "    " ++ show (instruction)
188    show (ArithmeticInstruction instruction) = "    " ++ show (instruction)
189    show (ComparisonInstruction instruction) = "    " ++ show (instruction)
190    show (LogicInstruction instruction) = "    " ++ show (instruction)
191    show (ConstantInstruction instruction) = "    " ++ show (instruction)
192    show (OperationInstruction instruction) = "    " ++ show (instruction)
193    show (BranchInstruction instruction) = "    " ++ show (instruction)
194    show (ProcedureInstruction instruction) = "    " ++ show (instruction)
195    show (DebugInstruction instruction) = "    " ++ show (instruction)
196    show (Comment comment) = "# " ++ id (comment)
197    show (Label label) = id (label) ++ ":"

199  -- display the given list of oz instruction line by line
200  writeCode :: [OzInstruction] -> String
201  writeCode instructions = concat $ map (\x -> (show x) ++ "\n") instructions

203  -- format the string in the oz code
204  strReplace :: String -> String
205  strReplace "" = ""
206  strReplace ('\\':'n':xs) = '\n' : strReplace xs
207  strReplace ('\\':'t':xs) = '\t' : strReplace xs
208  strReplace ('\\':'\"':xs) = '\"' : strReplace xs
209  strReplace (x:xs) = x : strReplace xs
```

PrettyRoo.hs

```
1   ----------------------------------------------------------
2   -- COMP90045 Programming Language Implementation Project --
3   --                  Roo Compiler                         --
4   --   Implemented by Xulin Yang                           --
5   --   Implemented by Team: GNU_project                    --
6   ----------------------------------------------------------
7   module PrettyRoo (pp)
8   where
9   import RooAST
10  import Data.List
11
12  -- Pretty print a whole program:
13  pp :: Program -> String
14  pp = strProgram
15
16  ----------------------------------------------------------------------
17  -- helper functions
18  ----------------------------------------------------------------------
19
20  -- add space indentations based on the input indentation level
21  addIndentation :: Int -> String
22  addIndentation 0 = ""
23  addIndentation n = "    " ++ addIndentation (n - 1)
24
25  newline :: String
26  newline = "\n"
27
28  semicolon :: String
29  semicolon = ";"
30
31  comma :: String
32  comma = ", "   -- There should be a single space after a comma
33
34  surroundByParens :: String -> String
35  surroundByParens s = "(" ++ s ++ ")"
36
37  surroundByBrackets :: String -> String
38  surroundByBrackets s = "[" ++ s ++ "]"
39
40  ----------------------------------------------------------------------
41  -- AST toString functions
42  ----------------------------------------------------------------------
43
44  strBaseType :: BaseType -> String
45  strBaseType BooleanType = "boolean"
46  strBaseType IntegerType = "integer"
47
48  strDataType :: DataType -> String
49  strDataType (AliasDataType t) = t -- t is String already
50  strDataType (BaseDataType b) = strBaseType b
51
```

```
52  strBool :: Bool -> String
53  strBool True = "true"
54  strBool False = "false"
55
56  -- String can be directly used as it is String type
57
58  -- Int can be turned to string by show
59
60  -----------------------------------------------------------------
61  -- An lvalue (<lvalue>) has four (and only four) possible forms:
62  --      <id>
63  --      <id>.<id>
64  --      <id>[<exp>]
65  --      <id>[<exp>].<id>
66  -----------------------------------------------------------------
67  strLValue :: LValue -> String
68  strLValue (LId ident) = ident
69  strLValue (LDot ident1 ident2) = ident1 ++ "." ++ ident2
70  strLValue (LBrackets ident exp) = ident ++ (surroundByBrackets (strExp exp))
71  strLValue (LBracketsDot ident1 exp ident2) = ident1 ++ (surroundByBrackets (strExp exp)) ++ "." ++ ident2
72
73  -----------------------------------------------------------------
74  -- Exp related toString & helper functions (with parens elimination)
75  -----------------------------------------------------------------
76  -- exp1 has higher precendence than exp2, higher if exp2 has no operator
77  isHigherPrecendence :: Exp -> Exp -> Bool
78  isHigherPrecendence exp1@(Op_neg _) exp2 =
79    case exp2 of
80      (Op_neg _  ) -> False
81      _            -> True
82  isHigherPrecendence exp1@(Op_mul _ _) exp2 =
83    case exp2 of
84      (Op_neg _  ) -> False
85      (Op_mul _ _) -> False
86      (Op_div _ _) -> False
87      _            -> True
88  isHigherPrecendence exp1@(Op_div _ _) exp2 =
89    case exp2 of
90      (Op_neg _  ) -> False
91      (Op_mul _ _) -> False
92      (Op_div _ _) -> False
93      _            -> True
94  isHigherPrecendence exp1@(Op_add _ _) exp2 =
95    case exp2 of
96      (Op_neg _  ) -> False
97      (Op_mul _ _) -> False
98      (Op_div _ _) -> False
99      (Op_add _ _) -> False
100     (Op_sub _ _) -> False
101     _            -> True
102 isHigherPrecendence exp1@(Op_sub _ _) exp2 =
103   case exp2 of
104     (Op_neg _  ) -> False
105     (Op_mul _ _) -> False
106     (Op_div _ _) -> False
107     (Op_add _ _) -> False
```

17

```
108      (Op_sub _ _) -> False
109      _                    -> True
110  isHigherPrecendence exp1@(Op_eq _ _) exp2 =
111    case exp2 of
112      (Op_not _  )  -> True
113      (Op_and _ _)  -> True
114      (Op_or  _ _)  -> True
115      -- constants has lowest precendence
116      (Lval _) -> True
117      (BoolConst _) -> True
118      (IntConst _) -> True
119      (StrConst _) -> True
120      _                  -> False
121  isHigherPrecendence exp1@(Op_neq _ _) exp2 =
122    case exp2 of
123      (Op_not _  )  -> True
124      (Op_and _ _)  -> True
125      (Op_or  _ _)  -> True
126      -- constants has lowest precendence
127      (Lval _) -> True
128      (BoolConst _) -> True
129      (IntConst _) -> True
130      (StrConst _) -> True
131      _                  -> False
132  isHigherPrecendence exp1@(Op_less _ _) exp2 =
133    case exp2 of
134      (Op_not _  )  -> True
135      (Op_and _ _)  -> True
136      (Op_or  _ _)  -> True
137      -- constants has lowest precendence
138      (Lval _) -> True
139      (BoolConst _) -> True
140      (IntConst _) -> True
141      (StrConst _) -> True
142      _                  -> False
143  isHigherPrecendence exp1@(Op_less_eq _ _) exp2 =
144    case exp2 of
145      (Op_not _  )  -> True
146      (Op_and _ _)  -> True
147      (Op_or  _ _)  -> True
148      -- constants has lowest precendence
149      (Lval _) -> True
150      (BoolConst _) -> True
151      (IntConst _) -> True
152      (StrConst _) -> True
153      _                  -> False
154  isHigherPrecendence exp1@(Op_large _ _) exp2 =
155    case exp2 of
156      (Op_not _  )  -> True
157      (Op_and _ _)  -> True
158      (Op_or  _ _)  -> True
159      -- constants has lowest precendence
160      (Lval _) -> True
161      (BoolConst _) -> True
162      (IntConst _) -> True
163      (StrConst _) -> True
```

```haskell
164      _                   -> False
165  isHigherPrecendence exp1@(Op_large_eq _ _) exp2 =
166    case exp2 of
167      (Op_not _  )  -> True
168      (Op_and _ _)  -> True
169      (Op_or  _ _)  -> True
170      -- constants has lowest precendence
171      (Lval _) -> True
172      (BoolConst _) -> True
173      (IntConst _) -> True
174      (StrConst _) -> True
175      _                   -> False
176  isHigherPrecendence exp1@(Op_not _) exp2 =
177    case exp2 of
178      (Op_and _ _)  -> True
179      (Op_or  _ _)  -> True
180      -- constants has lowest precendence
181      (Lval _) -> True
182      (BoolConst _) -> True
183      (IntConst _) -> True
184      (StrConst _) -> True
185      _                   -> False
186  isHigherPrecendence exp1@(Op_and _ _) exp2 =
187    case exp2 of
188      (Op_or  _ _)  -> True
189      -- constants has lowest precendence
190      (Lval _) -> True
191      (BoolConst _) -> True
192      (IntConst _) -> True
193      (StrConst _) -> True
194      _                   -> False
195  isHigherPrecendence exp1@(Op_or _ _) exp2
196      -- constants has lowest precendence
197    | (not (hasOperatorExp exp2)) = True
198    | otherwise = False
199  isHigherPrecendence _ _ = False
200
201  -- exp1 has same precendence as exp2
202  isSamePrecendence :: Exp -> Exp -> Bool
203  isSamePrecendence exp1@(Op_neg _) exp2 =
204    case exp2 of
205      (Op_neg _  ) -> True
206      _                 -> False
207  isSamePrecendence exp1@(Op_mul _ _) exp2 =
208    case exp2 of
209      (Op_mul _ _) -> True
210      (Op_div _ _) -> True
211      _                 -> False
212  isSamePrecendence exp1@(Op_div _ _) exp2 =
213    case exp2 of
214      (Op_mul _ _) -> True
215      (Op_div _ _) -> True
216      _                 -> False
217  isSamePrecendence exp1@(Op_add _ _) exp2 =
218    case exp2 of
219      (Op_add _ _) -> True
```

```
220       (Op_sub _ _) -> True
221       _                 -> False
222  isSamePrecendence exp1@(Op_sub _ _) exp2 =
223    case exp2 of
224      (Op_add _ _) -> True
225      (Op_sub _ _) -> True
226      _                 -> False
227  isSamePrecendence exp1@(Op_eq _ _) exp2        = False -- relational
228  isSamePrecendence exp1@(Op_neq _ _) exp2       = False -- relational
229  isSamePrecendence exp1@(Op_less _ _) exp2      = False -- relational
230  isSamePrecendence exp1@(Op_less_eq _ _) exp2   = False -- relational
231  isSamePrecendence exp1@(Op_large _ _) exp2     = False -- relational
232  isSamePrecendence exp1@(Op_large_eq _ _) exp2  = False -- relational
233  isSamePrecendence exp1@(Op_not _) exp2 =
234    case exp2 of
235      (Op_not _)  -> True
236      _                 -> False
237  isSamePrecendence exp1@(Op_and _ _) exp2 =
238    case exp2 of
239      (Op_and _ _)  -> True
240      _                 -> False
241  isSamePrecendence exp1@(Op_or _ _) exp2 =
242    case exp2 of
243      (Op_or  _ _)  -> True
244      _                 -> False
245  isSamePrecendence _ _ = False
246
247  -- exp1 has smaller precendence than exp2 if it is not higher nor same
248  isSamllerPrecendence :: Exp -> Exp -> Bool
249  isSamllerPrecendence exp1 exp2 = (not (isHigherPrecendence exp1 exp2)) &&
250                                   (not (isSamePrecendence   exp1 exp2))
251
252  -- does expression has operator in it?
253  hasOperatorExp :: Exp -> Bool
254  hasOperatorExp (Lval _) = False
255  hasOperatorExp (BoolConst _) = False
256  hasOperatorExp (IntConst _) = False
257  hasOperatorExp (StrConst _) = False
258  hasOperatorExp _ = True
259
260  -- return true if parent is sub/div and child has same precendence as parent
261  --    pexp: parent expression
262  --    cexp: child  expression
263  isDivSubParentSamePrecChild :: Exp -> Exp -> Bool
264  isDivSubParentSamePrecChild pexp@(Op_div _ _) cexp@(Op_div _ _) = True  -- / with a right child of / need
265  isDivSubParentSamePrecChild pexp@(Op_div _ _) cexp@(Op_mul _ _) = True  -- / with a right child of * need
266  isDivSubParentSamePrecChild pexp@(Op_sub _ _) cexp@(Op_sub _ _) = True  -- - (sub) with a right child of -
267  isDivSubParentSamePrecChild pexp@(Op_sub _ _) cexp@(Op_add _ _) = True  -- - (sub) with a right child of +
268  isDivSubParentSamePrecChild _ _ = False
269
270  -- True if Integer division happens
271  --    Integer division: 3 * (5 / 3) = 3 * 1 = 3, but (3 * 5) / 3 = 15 / 3 = 5,  so parens needed
272  isIntegerDision :: Exp -> Exp -> Bool
273  isIntegerDision pexp@(Op_mul _ _) cexp@(Op_div _ _) = True
274  isIntegerDision _ _ = False
275
```

```
276  -- some notation:
277  --    pexp: parent      expression (definitely has operator)
278  --    exp1: left child  expression
279  --    exp2: right child expression
280  -- turn binary expression's left child to string
281  strBinaryExpLChild :: Exp -> Exp -> String
282  strBinaryExpLChild pexp exp1
283    -- left child (with operator) has lower precendence suggests a parens
284    | (isSamllerPrecendence exp1 pexp) && (hasOperatorExp exp1) = surroundByParens (strExp exp1)
285    -- no parens
286    | otherwise = strExp exp1
287
288  -- True if expression is "not" <exp>
289  isNotExp :: Exp -> Bool
290  isNotExp (Op_not _) = True
291  isNotExp _ = False
292
293  -- some notation:
294  --    pexp: parent      expression (definitely has operator)
295  --    exp1: left child  expression
296  --    exp2: right child expression
297  -- turn binary expression's right child to string
298  strBinaryExpRChild :: Exp -> Exp -> String
299  strBinaryExpRChild pexp exp2
300    -- right child (with operator) has lower precendence (except "not" operator) or
301    --    same predence as parent (if parent is div(/) or sub(-)) or
302    --    integer division happens
303    -- suggests a parens
304    | (hasOperatorExp exp2) &&
305      ((isDivSubParentSamePrecChild pexp exp2) ||
306       (isSamllerPrecendence exp2 pexp) ||
307       (isIntegerDision pexp exp2)
308      ) &&
309      (not (isNotExp exp2))
310        = surroundByParens (strExp exp2)
311    -- no parens
312    | otherwise = strExp exp2
313
314  -- some notation:
315  --    pexp: parent      expression
316  --    exp1: left child  expression
317  --    exp2: right child expression
318  strExp :: Exp -> String
319  -- <lvalue>
320  strExp (Lval lValue) = strLValue lValue
321  -- <const>
322  strExp (BoolConst booleanLiteral) = strBool booleanLiteral
323  -- <const>
324  strExp (IntConst integerLiteral) = show integerLiteral
325  -- <const>
326  -- White space, and upper/lower case, should be preserved inside strings.
327  -- stringLiterals
328  strExp (StrConst stringLiteral) = "\"" ++ stringLiteral ++ "\""
329  -- <unop: "-"> <exp>
330  -- no space after unary minus
331  strExp pexp@(Op_neg exp)
```

```
332    | (hasOperatorExp exp) && (not (isSamePrecendence pexp exp)) = "-" ++ surroundByParens (strExp exp) -- n
333    | otherwise          = "-" ++ (strExp exp) -- no need to parens constants/Lval
334  -- <exp> <binop: "*"> <exp>
335  -- Single space should surround 12 binary operators.
336  strExp pexp@(Op_mul exp1 exp2) = (strBinaryExpLChild pexp exp1) ++ " * " ++ (strBinaryExpRChild pexp exp2)
337  -- <exp> <binop: "/"> <exp>
338  -- Single space should surround 12 binary operators.
339  strExp pexp@(Op_div exp1 exp2) = (strBinaryExpLChild pexp exp1) ++ " / " ++ (strBinaryExpRChild pexp exp2)
340  -- <exp> <binop: "+"> <exp>
341  -- Single space should surround 12 binary operators.
342  strExp pexp@(Op_add exp1 exp2) = (strBinaryExpLChild pexp exp1) ++ " + " ++ (strBinaryExpRChild pexp exp2)
343  -- <exp> <binop: "-"> <exp>
344  -- Single space should surround 12 binary operators.
345  strExp pexp@(Op_sub exp1 exp2) = (strBinaryExpLChild pexp exp1) ++ " - " ++ (strBinaryExpRChild pexp exp2)
346  -- <exp> <binop: "="> <exp>
347  -- Single space should surround 12 binary operators.
348  strExp pexp@(Op_eq exp1 exp2) = (strBinaryExpLChild pexp exp1) ++ " = " ++ (strBinaryExpRChild pexp exp2)
349  -- <exp> <binop: "!="> <exp>
350  -- Single space should surround 12 binary operators.
351  strExp pexp@(Op_neq exp1 exp2) = (strBinaryExpLChild pexp exp1) ++ " != " ++ (strBinaryExpRChild pexp exp2
352  -- <exp> <binop: "<"> <exp>
353  -- Single space should surround 12 binary operators.
354  strExp pexp@(Op_less exp1 exp2) = (strBinaryExpLChild pexp exp1) ++ " < " ++ (strBinaryExpRChild pexp exp2
355  -- <exp> <binop: "<="> <exp>
356  -- Single space should surround 12 binary operators.
357  strExp pexp@(Op_less_eq exp1 exp2) = (strBinaryExpLChild pexp exp1) ++ " <= " ++ (strBinaryExpRChild pexp (
358  -- <exp> <binop: ">"> <exp>
359  -- Single space should surround 12 binary operators.
360  strExp pexp@(Op_large exp1 exp2) = (strBinaryExpLChild pexp exp1) ++ " > " ++ (strBinaryExpRChild pexp exp2
361  -- <exp> <binop: ">="> <exp>
362  -- Single space should surround 12 binary operators.
363  strExp pexp@(Op_large_eq exp1 exp2) = (strBinaryExpLChild pexp exp1) ++ " >= " ++ (strBinaryExpRChild pexp
364  -- <unop: not> <exp>
365  -- There should be a single space after not.
366  strExp pexp@(Op_not exp)
367    -- need to parens expression with operator and (child's precendence is smaller)
368    | (hasOperatorExp exp) && (isSamllerPrecendence exp pexp) = "not " ++ surroundByParens (strExp exp)
369    | otherwise          = "not " ++ (strExp exp) -- no need to parens constants/Lval
370  -- <exp> <binop: and> <exp>
371  -- Single space should surround 12 binary operators.
372  strExp pexp@(Op_and exp1 exp2) = (strBinaryExpLChild pexp exp1) ++ " and " ++ (strBinaryExpRChild pexp exp
373  -- <exp> <binop: or> <exp>
374  -- Single space should surround 12 binary operators.
375  strExp pexp@(Op_or exp1 exp2) = (strBinaryExpLChild pexp exp1) ++ " or " ++ (strBinaryExpRChild pexp exp2)
376
377
378  -- Int: indentation level
379  strStmt :: Int -> Stmt -> String
380  -- In a procedure body, each statement should start on a new line. So ++ newline in each's end
381  strStmt indentLevel (Assign lValue exp) =
382    -- <lvalue> <- <exp>;
383    -- Single spaces should surround the assignment operator <-
384    (addIndentation indentLevel) ++ (strLValue lValue) ++ " <- " ++ (strExp exp) ++ semicolon ++ newline
385  strStmt indentLevel (Read lValue) =
386    -- read <lvalue>;
387    (addIndentation indentLevel) ++ "read " ++ (strLValue lValue) ++ semicolon ++ newline
```

```
388  strStmt indentLevel (Write exp) =
389    -- write <exp>;
390    (addIndentation indentLevel) ++ "write " ++ (strExp exp) ++ semicolon ++ newline
391  strStmt indentLevel (Writeln exp) =
392    -- writeln <exp>;
393    (addIndentation indentLevel) ++ "writeln " ++ (strExp exp) ++ semicolon ++ newline
394  strStmt indentLevel (Call ident exps) =
395    -- call <id>(<exp-list>);
396    --     where <exp-list> is a (possibly empty according to parser) comma-separated list of expressions.
397    (addIndentation indentLevel) ++ "call " ++ ident ++ surroundByParens (intercalate comma (map strExp exps
398  -- thenStmts is non-empty according to parser, elseStmts is possible empty according to parser
399  strStmt indentLevel (IfThen exp thenStmts) =
400    -- if <exp> then <stmt-list> fi
401      -- "if ... then" should be printed on one line, irrespective of the size of the intervening expression
402      (addIndentation indentLevel) ++ "if " ++ (strExp exp) ++ " then" ++ newline ++
403      -- more indentation
404      (concatMap (strStmt (indentLevel+1)) thenStmts) ++
405      -- the terminating fi should be indented exactly as the corresponding if.
406      (addIndentation indentLevel) ++ "fi" ++ newline
407  strStmt indentLevel (IfThenElse exp thenStmts elseStmts) =
408    -- if <expr> then <stmt-list> else <stmt-list> fi
409      -- "if ... then" should be printed on one line, irrespective of the size of the intervening expression
410      (addIndentation indentLevel) ++ "if " ++ (strExp exp) ++ " then" ++ newline ++
411      -- more indentation
412      (concatMap (strStmt (indentLevel+1)) thenStmts) ++
413      (addIndentation indentLevel) ++ "else" ++ newline ++
414      -- more indentation
415      (concatMap (strStmt (indentLevel+1)) elseStmts) ++
416      -- the terminating fi and else should be indented exactly as the corresponding if.
417      (addIndentation indentLevel) ++ "fi" ++ newline
418  -- stmts is non-empty according to parser
419  strStmt indentLevel (While exp stmts) =
420    -- In a while statement, "while ... do" should be printed on one line,
421    --     irrespective of the size of the intervening expression
422    (addIndentation indentLevel) ++ "while " ++ (strExp exp) ++ " do" ++ newline ++
423    -- more indentation
424    (concatMap (strStmt (indentLevel+1)) stmts) ++
425    -- The terminating od should be indented exactly as the corresponding while.
426    (addIndentation indentLevel) ++ "od" ++ newline
427
428  ----------------------------------------------------------------
429  -- Record toString function
430  ----------------------------------------------------------------
431  -- field declaration is of:
432  --   1. boolean or integer
433  --   2. followed by an identifier (the field name).
434  strFieldDecl :: FieldDecl -> String
435  strFieldDecl (FieldDecl baseType fieldName) = (strBaseType baseType) ++ " " ++ fieldName
436
437  -- non-empty input list fieldDecls@(x:xs) according to parser
438  strFieldDecls :: [FieldDecl] -> String
439  -- first field decl starts with {
440  strFieldDecls (x:xs) = (addIndentation 1) ++ "{ " ++ (strFieldDecl x) ++ newline ++
441  -- rest start with ;
442                         (concatMap
443                           (\y -> (addIndentation 1) ++ "; " ++ (strFieldDecl y) ++ newline)
```

```
444                          xs
445                      )
446
447    -- convert record to string
448    -- A record type definition involving n fields should be printed on n + 2 lines,
449    -- as follows:
450    --     1. The first line contains the word record.
451    --     2. The remaining lines should be indented, with the first n containing one field declaration each
452    --        (the first preceded by a left brace and a single space, the rest preceded
453    --          by a semicolon and a single space),
454    --          see above strFieldDecls
455    --     3. and with the last line containing the record name, preceded by a right
456    --        brace and a single space, and followed by a semicolon;
457    strRecord :: Record -> String
458    strRecord (Record fieldDecls recordName) =
459      "record" ++ newline ++
460      (strFieldDecls fieldDecls) ++
461      (addIndentation 1) ++ "} " ++ recordName ++ semicolon ++ newline
462
463
464    -----------------------------------------------------------------
465    --  Array toString function
466    -----------------------------------------------------------------
467    -- An array type definition should be printed on a single line.
468    -- It contains the word array,
469    -- followed by a positive integer in square brackets all without intervening
470    --      white space.
471    -- That string, the type, and the type alias, should be separated by single
472    --    spaces, and the whole line terminated by a semicolon.
473    strArray :: Array -> String
474    strArray (Array arraySize arrayType arrayName) =
475      "array" ++ surroundByBrackets (show arraySize) ++ " " ++ (strDataType arrayType) ++ " " ++ arrayName ++
476
477
478    -----------------------------------------------------------------
479    --  Procedure toString functions
480    -----------------------------------------------------------------
481    strParameter :: Parameter -> String
482    strParameter (DataParameter dataType paraName) = (strDataType dataType) ++ " " ++ paraName
483    strParameter (BooleanVal paraName) = "boolean val " ++ paraName
484    strParameter (IntegerVal paraName) = "integer val " ++ paraName
485
486    -- The procedure head (that is, the keyword, procedure name, and list of formal
487    --      parameters) should be on a single line.
488    strProcedureHeader :: ProcedureHeader -> String
489    strProcedureHeader (ProcedureHeader procedureName parameters) =
490      procedureName ++ " " ++ (surroundByParens (intercalate comma (map strParameter parameters)))
491
492    -- A variable declaration consists of
493    --   a) a type name (boolean, integer, or a type alias),
494    --   b) followed by a 1+ comma-separated list of
495    --        identifiers,
496    --     i)  the list terminated with a semicolon.
497    --     ii) There may be any number of variable declarations, in any order.
498    strVariableDecl :: VariableDecl -> String
499    strVariableDecl (VariableDecl dataType varNames) =
```

24

```
500    -- Within each procedure, declarations and top-level statements should be indented.
501    (addIndentation 1) ++ (strDataType dataType) ++ " " ++ (intercalate comma varNames)  ++ semicolon ++
502    -- Each variable declaration should be on a separate line.
503    newline
504
505    -- variableDecls can be empty according to parser
506    -- stmts is non-empty according to parser
507    strProcedureBody :: ProcedureBody -> String
508    strProcedureBody (ProcedureBody variableDecls stmts) =
509      (concatMap strVariableDecl variableDecls) ++
510      -- The { and } that surround a procedure body should begin at the start of a
511      --    line (no indentation).
512      -- Moreover, these delimiters should appear alone, each making up a single line.
513      "{" ++ newline ++
514      -- Within each procedure, declarations and top-level statements should be indented.
515      concatMap (strStmt 1) stmts ++
516      "}" ++ newline
517
518    -- convert procedure to string
519    strProcedure :: Procedure -> String
520    -- The keyword procedure should begin at the start of a line (no indentation)
521    -- The procedure head (that is, the keyword, procedure name, and list of formal
522    --    parameters) should be on a single line.
523    strProcedure (Procedure ph pb) =
524      "procedure " ++ (strProcedureHeader ph) ++ newline ++
525      (strProcedureBody pb)
526
527
528    ------------------------------------------------------------------
529    --  Program toString function
530    ------------------------------------------------------------------
531    strProgram :: Program -> String
532    -- If there are no record and array type definitions, the first procedure should
533    --    start on line 1.
534    strProgram (Program [] [] procedures) = intercalate newline (map strProcedure procedures)
535    -- Otherwise there should be a single blank line between the type definitions
536    --    and the first procedure.
537    strProgram (Program records arraies procedures) =
538      -- Each type definition should start on a new line, and there should be no
539      --    blank lines between type definitions. So below two has no newline in between
540      concatMap strRecord records ++
541      concatMap strArray arraies ++
542      newline ++
543      -- Consecutive procedure definitions should be separated by a single blank line.
544      intercalate newline (map strProcedure procedures)
```

```haskell
1  -----------------------------------------------------------------
2  -- COMP90045 Programming Language Implementation Project --
3  --                   Roo Compiler                        --
4  --   Implemented by Xulin Yang                           --
5  -----------------------------------------------------------------
6  module Main (main)
7  where
8  import RooParser (ast)
9  import PrettyRoo (pp)
10 import RooAnalyser (analyse, Result(..))
11 import OzCode (writeCode)
12 import Codegen (ozCode, Consequence(..))
13 import System.Environment (getProgName, getArgs)
14 import System.Exit (exitWith, ExitCode(..))
15
16 data Task
17   = Parse | Pprint | NoOz | Compile
18      deriving (Eq, Show)
19
20 main :: IO ()
21 main
22   = do
23       progname <- getProgName
24       args <- getArgs
25       task <- checkArgs progname args
26       case task of
27         Compile
28           -> do
29                input <- readFile (head args)
30                let output = ast input
31                case output of
32                  Right tree
33                    -> do let pt = analyse tree
34                          case pt of
35                            Left err
36                              -> do putStrLn err
37                                    exitWith (ExitFailure 2)
38                            Right table
39                              -> do
40                                   let code = ozCode table tree
41                                   case code of
42                                     Left err ->
43                                       do
44                                         putStrLn err
45                                         exitWith (ExitFailure 2)
46                                     Right instructions ->
47                                       do
48                                         putStrLn (writeCode instructions)
49                  Left err
50                    -> do putStr "Parse error at "
51                          print err
```

```
52                          exitWith (ExitFailure 2)
53                      exitWith ExitSuccess
54          NoOz
55            -> do
56                  let [_, filename] = args
57                  input <- readFile filename
58                  let output = ast input
59                  case output of
60                    Right tree
61                      -> do let pt = analyse tree
62                            case pt of
63                              Left err
64                                -> do putStrLn err
65                                      exitWith (ExitFailure 2)
66                              Right _
67                                -> putStrLn "Roo program appears well-formed"
68                    Left err
69                      -> do putStr "Parse error at "
70                            print err
71                            exitWith (ExitFailure 2)
72                  exitWith ExitSuccess
73          Parse
74            -> do
75                  let [_, filename] = args
76                  input <- readFile filename
77                  let output = ast input
78                  case output of
79                    Right tree
80                      -> putStrLn (show tree)
81                    Left err
82                      -> do putStrLn "Parse error at "
83                            print err
84                            exitWith (ExitFailure 2)
85          Pprint
86            -> do
87                  let [_, filename] = args
88                  input <- readFile filename
89                  let output = ast input
90                  case output of
91                    Right tree
92                      -> putStr (pp tree)
93                    Left err
94                      -> do putStrLn "Parse error at "
95                            print err
96                            exitWith (ExitFailure 2)
97
98  checkArgs :: String -> [String] -> IO Task
99  checkArgs _ ['-':_]
100    = do
101        putStrLn ("Missing filename")
102        exitWith (ExitFailure 1)
103 checkArgs _ [filename]
104    = return Compile
105 checkArgs _ ["-a", filename]
106    = return Parse
107 checkArgs _ ["-p", filename]
```

```haskell
108      = return Pprint
109  checkArgs _ ["-n", filename]
110      = return NoOz
111  checkArgs progname _
112      = do
113          putStrLn ("Usage: " ++ progname ++ " [-p] filename")
114          exitWith (ExitFailure 1)
115
```

```
1    ----------------------------------------------------------------
2    -- COMP90045 Programming Language Implementation Project --
3    --                    Roo Compiler                       --
4    --  Implemented by Xulin Yang                            --
5    --  Implemented by Team: GNU_project                     --
6    ----------------------------------------------------------------
7    module RooAST where
8
9    ------------------------------------
10   -- Terminology:
11   -- 0+: zero or more/possible empty
12   -- 1+: one or more/ non empty
13   --     both 0+, 1+ are stored in list [] but 1+ will be implemented in parser
14   --      not here
15   ------------------------------------
16
17   ------------------------------------
18   -- Specification of an AST for Roo
19   ------------------------------------
20
21   -- Identifier: String
22   type Ident = String
23
24   -- Base type: boolean, integer type indicator
25   --     Not necessary to have string as no variable/parameter/declaration has
26   --      string type
27   data BaseType
28     = BooleanType
29     | IntegerType
30     | StringType   --Add Stringtype ,because exp has string type
31       deriving (Show, Eq)
32
33   -- User custermized record type, stored as string
34   type AliasType = String
35
36   -- for Array, VariableDecl: they have either boolean, integer, or a type alias
37   --   data type
38   --     factored out for reuse purpose
39   data DataType
40     = BaseDataType BaseType
41     | AliasDataType AliasType
42       deriving (Show, Eq)
43
44   -- An lvalue (<lvalue>) has four (and only four) possible forms:
45   --     An example lvalue is point[0].xCoord
46   data LValue
47     = LId Ident                    -- <id>
48     | LDot Ident Ident             -- <id>.<id>
49     | LBrackets Ident Exp          -- <id>[<exp>]
50     | LBracketsDot Ident Exp Ident -- <id>[<exp>].<id>
51       deriving (Show, Eq)
```

```haskell
52
53   -- expression operators:
54   --     All the operators on the same line have the same precedence,
55   --          and the ones on later lines have lower precedence;
56   --     The six relational operators are non-associative
57   --          so, for example, a = b = c is not a well-formed expression).
58   --     The six remaining binary operators are left-associative.
59   -- -                  |unary          |
60   -- * /                |binary and infix |left-associative
61   -- + -                |binary and infix |left-associative
62   -- = != < <= > >=     |binary and infix |relational, non-associative
63   -- not                |unary          |
64   -- and                |binary and infix |left-associative
65   -- or                 |binary and infix |left-associative
66   data Exp
67     = Lval LValue              -- <lvalue>
68     | BoolConst Bool   -- <const> where <const> is the syntactic category of
69                        --  boolean, integer, and string literals.
70     | IntConst Int
71     | StrConst String
72                               -- ( <exp> ) is ignored here but handelled in
73                               --  parser
74     | Op_or Exp Exp            -- <exp> <binop: or> <exp>
75     | Op_and Exp Exp           -- <exp> <binop: and> <exp>
76     | Op_eq   Exp Exp          -- <exp> <binop: "="> <exp>
77     | Op_neq  Exp Exp          -- <exp> <binop: "!="> <exp>
78     | Op_less  Exp Exp         -- <exp> <binop: "<"> <exp>
79     | Op_less_eq  Exp Exp      -- <exp> <binop: "<="> <exp>
80     | Op_large  Exp Exp        -- <exp> <binop: ">"> <exp>
81     | Op_large_eq  Exp Exp     -- <exp> <binop: ">="> <exp>
82     | Op_add  Exp Exp          -- <exp> <binop: "+"> <exp>
83     | Op_sub  Exp Exp          -- <exp> <binop: "-"> <exp>
84     | Op_mul  Exp Exp          -- <exp> <binop: "*"> <exp>
85     | Op_div  Exp Exp          -- <exp> <binop: "/"> <exp>
86     | Op_not Exp               -- <unop: not> <exp>
87     | Op_neg Exp               -- <unop: "-"> <exp>
88       deriving (Show, Eq)
89
90   -- Stmt has following two category:
91   --   1) atom statement:
92   --       <lvalue> <- <exp> ;
93   --       read <lvalue> ;
94   --       write <exp> ;
95   --       writeln <exp> ;
96   --       call <id>(<exp-list>) ;
97   --           where <exp-list> is a 0+ comma-separated list of expressions.
98   --   2) composite statement:
99   --       if <exp> then <stmt-list> else <stmt-list> fi
100  --       if <exp> then <stmt-list> fi # just make above second [Stmt] empty
101  --       while <exp> do <stmt-list> od
102  --           where <stmt-list> is a 1+ sequence of statements, atomic or composite
103  --
104  -- the data structure for above grammer are given accordingly below
105  data Stmt
106    -- 1) atom statement:
107    = Assign LValue Exp
```

```haskell
108      | Read LValue
109      | Write Exp
110      | Writeln Exp
111      | Call Ident [Exp]
112      -- 2) composite statement:
113      | IfThen Exp [Stmt]
114      | IfThenElse Exp [Stmt] [Stmt]
115      | While Exp [Stmt]
116        deriving (Show, Eq)
117
118  -- Each formal parameter has two components (in the given order):
119  --   1. a parameter type/mode indicator, which is one of these five:
120  --      a) a type alias,
121  --      b) boolean,
122  --      c) integer,
123  --      d) boolean val
124  --      e) integer val
125  --   2. an identifier
126  data Parameter
127    = DataParameter DataType Ident -- a) b) c) above
128    | BooleanVal Ident            -- d)        above
129    | IntegerVal Ident            -- e)        above
130        deriving (Show, Eq)
131
132  -- The header has two components (in this order):
133  --   1. an identifier (the procedure's name), and
134  --   2. a comma-separated list of 0+ formal parameters within a pair
135  --        of parentheses (so the parentheses are always present).
136  data ProcedureHeader
137    = ProcedureHeader Ident [Parameter]
138        deriving (Show, Eq)
139
140  -- A variable declaration consists of
141  --   a) a type name (boolean, integer, or a type alias),
142  --   b) followed by a 1+ comma-separated list of
143  --        identifiers,
144  --     i)  the list terminated with a semicolon.
145  --     ii) There may be any number of variable declarations, in any order.
146  data VariableDecl
147    = VariableDecl DataType [Ident]
148        deriving (Show, Eq)
149
150  -- procedure body consists of 0+ local variable declarations,
151  --   1. A variable declaration consists of
152  --      a) a type name (boolean, integer, or a type alias),
153  --      b) followed by a 1+ comma-separated list of identifiers,
154  --        i)  the list terminated with a semicolon.
155  --        ii) There may be any number of variable declarations, in any order.
156  --   2. followed by a 1+ sequence of statements,
157  data ProcedureBody
158    = ProcedureBody [VariableDecl] [Stmt]
159        deriving (Show, Eq)
160
161  -- Each procedure consists of (in the given order):
162  --   1. the keyword procedure,
163  --   2. a procedure header, and
```

```haskell
164  --    3. a procedure body.
165  data Procedure
166    = Procedure ProcedureHeader ProcedureBody
167      deriving (Show, Eq)
168
169  -- array type definition consists of (in the given order):
170  --    1. the keyword array,
171  --    2. a (positive) integer literal enclosed in square brackets,
172  --    3. a type name which is either an identifier (a type alias) or one of
173  --       boolean and integer,
174  --    4. an identifier (giving a name to the array type), and
175  --    5. a semicolon.
176  data Array
177    = Array Int DataType Ident
178      deriving (Show, Eq)
179
180  -- field declaration is of:
181  --    1. boolean or integer
182  --    2. followed by an identifier (the field name).
183  data FieldDecl
184    = FieldDecl BaseType Ident
185      deriving (Show, Eq)
186
187  -- record consists of:
188  --    1. the keyword record,
189  --    2. a 1+ list of field declarations, separated by semicolons,
190  --         the whole list enclosed in braces,
191  --    3. an identifier, and
192  --    4. a semicolon.
193  data Record
194    = Record [FieldDecl] Ident
195      deriving (Show, Eq)
196
197  -- A Roo program consists of
198  --    1. 0+ record type definitions, followed by
199  --    2. 0+ array type definitions, followed by
200  --    3. 1+ procedure definitions.
201  data Program
202    = Program [Record] [Array] [Procedure]
203      deriving (Show, Eq)
```

```
1   ----------------------------------------------------------------
2   -- COMP90045 Programming Language Implementation Project --
3   --                    Roo Compiler                        --
4   --   Implemented by Xulin Yang, Wenrui Zhang, Xu Shi       --
5   --   Implemented by Team: GNU_project                      --
6   ----------------------------------------------------------------
7   module RooAnalyser(analyse, Result(..)) where
8
9   import Control.Monad
10  import Control.Monad.State
11  import Control.Monad.Except
12  import RooAST
13  import SymbolTable
14  import Data.Map (Map, (!),delete)
15  import qualified Data.Map as Map
16  import Data.Either
17
18  type Result = Either String SymTable
19
20  ------------------main function of semantic analysis-----------------------
21
22  analyse :: Program -> Result
23  analyse prog
24    = evalStateT (semanticCheckRooProgram prog) initialSymTable
25
26  ----------------- semantic analysis on a roo program----------------------
27  -- semanticCheckRooProgram
28  -- 1.insert records arraies and procedures into symbol table
29  -- 2.Check that there is one and only one main procedure, and its arity is 0
30  -- 3.Check all procedures
31
32  semanticCheckRooProgram :: Program -> SymTableState SymTable
33  semanticCheckRooProgram prog
34    =
35      do
36        constructSymbolTable prog
37        checkArityProcedure "main" 0
38
39        st <- get
40        let procedures = pt st
41        mapM_ checkOneProcedures procedures
42        st2 <- get
43        return st2
44
45  -----------------------------------------------------------------------------
46  --------------------Semantic checking on a procedures----------------------
47  -- checkOneProcedures
48  -- 1.initialize local variable table
49  -- 2.insert procedure's parameter and variable into lvt
50  -- 3.check all statements
51  -- 4.Empty lvt
```

```
52
53   checkOneProcedures :: ([(Bool, DataType)], Procedure) -> SymTableState ()
54   checkOneProcedures  (_, procCalled@(Procedure _ (ProcedureBody _ procStmts)))
55      =
56        do
57            pushLocalVariableTable
58            insertProcedureVariable procCalled
59            checkStmts procStmts
60            popLocalVariableTable
61
62   --------------------------------------------------------------------------------
63   ------------Semantic checking on all statement of a procedure-----------------
64   checkStmts :: [Stmt] -> SymTableState ()
65   checkStmts = mapM_ checkStmt
66
67
68   --check one procedure:
69   checkStmt :: Stmt -> SymTableState ()
70   -- check assign:
71   -- 1.Check if lvalue and exp use the correct format
72   -- 2.check if lvalue and exp have same data type
73   -- 3. if exp are record or array type, both lvalue and exp
74   -- pass by reference
75   checkStmt (Assign lvalue exp)
76      =
77        do
78          checkLValue lvalue
79          checkExp exp
80          iInfo <- getDataTypeOfLValue lvalue
81          let (byV, identi) = iInfo
82          expType <- getExpType exp
83          let showLValueName = getLValueName lvalue
84          let showTypeName = getDataT expType
85
86          if  not (identi == expType) then
87            liftEither $ throwError $ "assign a wrong type "
88            ++ showTypeName ++ " to " ++ showLValueName
89          else
90            if expIsLvalue exp then
91               do
92
93                 let (Lval expLvalue) = exp
94                 iRA <- lvalueIsRerAry expLvalue
95                 if iRA then
96                   do
97                     iInfo2 <- getDataTypeOfLValue expLvalue--jiancha
98                     let (byV2, dataType2) = iInfo2
99                     if (byV2 == False) && (byV == False) then
100                        return ()
101                     else
102                        liftEither $ throwError $
103                        "Both side of this assignment must pass by reference "
104                  else
105                     return ()
106
107             else
```

34

```
108            return ()
109  -- check read
110  -- 1.Check if lvalue uses the correct format
111  -- 2.Check lvalue is Boolean or Integer type
112  checkStmt (Read lvalue)
113    =
114      do
115        checkLValue lvalue
116        lvalueInfo <- getDataTypeOfLValue lvalue
117        let (byV,lvalueType) = lvalueInfo
118        if (lvalueType == BaseDataType BooleanType)
119          || (lvalueType == BaseDataType IntegerType) then
120          return ()
121        else
122          liftEither $ throwError
123          ("Read lvalue,lvalue is not a boolean or integer type lvalue")
124  -- check write
125  -- 1.Check if exp uses the correct format
126  -- 2.Check if exp is Boolean Integer or String literal
127  checkStmt (Write exp)
128    =
129      do
130        checkExp exp
131        expType<-getExpType exp
132        if (expType == BaseDataType BooleanType)
133          || (expType == BaseDataType IntegerType)
134          || (expType == BaseDataType StringType) then
135          return ()
136        else
137          liftEither $ throwError
138          ("write exp, exp is not a boolean or integer, or a string literal.")
139  -- Check writeln: same as write
140  checkStmt (Writeln exp)
141    =
142      do
143        checkExp exp
144        expType <- getExpType exp
145        if (expType == BaseDataType BooleanType)
146          || (expType == BaseDataType IntegerType)
147          || (expType == BaseDataType StringType) then
148          return ()
149        else
150          liftEither $ throwError
151          ("writeln exp, exp is not a boolean or integer, or a string literal.")
152
153  -- Check Ifthen
154  -- 1.Check if exp uses the correct format
155  -- 2.Check if exp is Boolean type
156  -- 3.check stmts
157  checkStmt (IfThen exp stmts)
158    =
159      do
160        checkExp exp
161        expType <- getExpType exp
162        if not (expType == (BaseDataType BooleanType)) then
163            liftEither $ throwError ("IfThen exp, exp is not boolean type")
```

```
164        else
165          do
166            checkStmts stmts
167  -- Check IfThenElse
168  -- 1.Check if exp uses the correct format
169  -- 2.Check if exp is Boolean type
170  -- 3.check two "stmts"
171  checkStmt (IfThenElse exp stmts1 stmts2)
172    =
173      do
174        checkExp exp
175        expType <- getExpType exp
176        if not (expType == (BaseDataType BooleanType)) then
177            liftEither $ throwError
178            ("IfThenElse exp,exp is not boolean type")
179        else
180          do
181            checkStmts stmts1
182            checkStmts stmts2
183  -- Check While
184  -- 1.Check if exp uses the correct format
185  -- 2.Check if exp is Boolean type
186  -- 3.check stmts
187  checkStmt (While exp stmts )
188    =
189      do
190        checkExp exp
191        expType <- getExpType exp
192        if not (expType == (BaseDataType BooleanType)) then
193            liftEither $ throwError
194            ("While exp, exp is not boolean type")
195        else
196          do
197            checkStmts stmts
198  -- Check Call
199  -- 1.Check if exps use the correct format
200  -- 2.Check if procedure called "procedureName" is exist
201  -- 3.The number of actual parameters in a call must be equal to the number
202  --    of formal parameters in the procedure's definition.
203  -- 4.The type of actual parameters in a call must match the type of
204  --    formal parameters in the procedure's definition.
205  checkStmt (Call procedureName exps)
206    =
207      do
208        mapM_ checkExp exps
209        (proParams, procCalled) <- getProcedure procedureName
210        let nParamsFound = length exps
211        let nParamsExpected = length proParams
212        if nParamsFound /= nParamsExpected then
213          liftEither $ throwError (show nParamsExpected ++
214          " parameters expected for procedure: \"" ++ procedureName ++ "\" " ++
215          show nParamsFound ++ " parameters found")
216        else
217          do
218            temp <- hasSameElem exps proParams
219            if not temp then
```

36

```
220      liftEither $ throwError
221        ("The type of the parameter does not match what you are calling")
222    else
223      do
224        return ()


227  --------------------------------------------------------------------------------
228  ----------------------Semantic checking on lvalue----------------------------
229  -- Check all lvalue
230  -- Check four kinds of lvalue
231  -- 1.all variable should been declared before using
232  -- 2.<varname>
233  --   <recordName>.<fieldname>
234  --       <recordName> should be record type,and <fieldname> should been in
235  --     this kind of record.
236  --   <arrayName>[index]
237  --       <arrayName> should be array type.
238  --       [index] this exp should be integer type.
239  --   <arrayName>[index].<fieldname>
240  --       <arrayName> should be array type storing record type,and <fieldname>
241  --     should been in this kind of record.
242  --       [index] this exp should be integer type.
243  --
244  checkLValue :: LValue -> SymTableState ()
245  -- <varname>
246  checkLValue (LId varName)
247    =
248      do
249        cvt <- getCurVariableTable
250        if (Map.member varName (vtt cvt)) then
251          do
252            return()
253        else
254          liftEither $ throwError $ "Undeclared variable name: " ++ varName

256  -- <recordvarname>.<fieldname>
257  checkLValue (LDot recordVarname fieldName)
258    =
259      do
260        cvt <- getCurVariableTable
261        -- if this variable has been declared before
262        if (Map.member recordVarname (vtt cvt)) then
263          do
264            st <- get
265            c <- getVariableType recordVarname
266            let (bool, int1, variableType, int2) = c
267            --check if variable name corresponds to a record variable
268            if varIsRecordType variableType then
269              do
270                let (RecordVar recordType) = variableType
271                let ck = CompositeKey recordType fieldName
272                -- this <>.<> is exist in record variable table
273                if (Map.member ck (rft st)) then
274                    return ()
275                else
```

```
276            liftEither $ throwError $ "Record.field: " ++
277            recordVarname ++ "." ++ fieldName
278            ++ " does not exist"
279          else
280              liftEither $ throwError $
281              recordVarname++" is not a record name"
282        else
283          liftEither $ throwError $
284          "Undeclared variable name: " ++ recordVarname
285
286  -- <arrayVarName> [index]
287  checkLValue (LBrackets arrayName int)
288    =
289      do
290        cvt <- getCurVariableTable
291        -- if this variable has been declared before
292        if (Map.member arrayName (vtt cvt)) then
293          do
294            varInfo <- (getVariableType arrayName)
295            let (bool,int1,vartype,arraysize) = varInfo
296            --check if variable name corresponds to a array variable
297            if varIsArrayType vartype then
298              do
299                  let (ArrayVar arrayType) = vartype
300                  artype <- getArrayType arrayType
301                  let (intt, dataType) = artype
302
303                  indextype <- getExpType int
304                  --if expression <int> is a integer type
305                  if indextype == BaseDataType IntegerType then
306                    return()
307                  else
308                    liftEither $ throwError $
309                    "Array's index should be an integer type "
310            else
311              liftEither $ throwError $
312              arrayName ++ " is not array variable name "
313
314        else
315          liftEither $ throwError $
316          "Undeclared variable name: " ++ arrayName
317  -- <arrayVarName>[index].<fieldname>
318  checkLValue (LBracketsDot arrayName int fieldName)
319    =
320      do
321        cvt <- getCurVariableTable
322        -- if this variable has been declared before
323        if (Map.member arrayName (vtt cvt)) then
324          do
325            c <- getVariableType arrayName
326            let (bool, int1, variableType, int2) = c
327            --check if variable name corresponds to a array variable
328            if varIsArrayType variableType then
329              do
330                  let (ArrayVar arrayType) = variableType
331                  artype <- getArrayType arrayType
```

```
332          let (intt, dataType) = artype
333          --if this array <>[] is storing a record type
334          if dataIsRecordTypeStoreInArray dataType then
335             do
336                let (AliasDataType alsName) = dataType
337                st <- get
338                let ck = CompositeKey alsName fieldName
339                -- this <>.<> is exist in rrecord variable table
340                if (Map.member ck (rft st)) then
341                   do
342                      indexType <- getExpType int
343                      --if expression <index> is a integer type
344                      if indexType == BaseDataType IntegerType then
345                         return()
346                      else
347                         liftEither $ throwError $
348                         "Array's index should be an integer type "
349                   else
350                      liftEither $ throwError $
351                      "Record.field: " ++ arrayName ++ "[]." ++ fieldName ++
352                                          " does not exist"
353             else
354                liftEither $ throwError $ arrayName ++
355                " is not a array storing record "
356          else
357             liftEither $ throwError $  arrayName ++
358             " is not array variable name "
359       else
360          liftEither $ throwError $ "Undeclared variable name: " ++ arrayName
361

362 --------------------------------------------------------------------------------
363 -----------------semantic check on all kinds of expression--------------------
364 -- •The type of a Boolean constant isboolean.
365 -- •The type of an integer constant isinteger.
366 -- •The type of a string literal isstring.
367 -- •The two operands of a relational operator must have the same primitive
368 --  type, eitherbooleanorinteger. The result is of typeboolean.
369 -- •The two operands of a binary arithmetic operator must have typeinteger,
370 --  and the resultis of typeinteger.
371 -- •The operand of unary minus must be of typeinteger, and the result
372 --  type is the same.
373 -- •Check lvalue as CheckLValue

374
375 checkExp :: Exp -> SymTableState ()
376 checkExp (BoolConst bool)
377   =
378     do return()
379
380 checkExp (IntConst int)
381   =
382     do return()
383
384 checkExp (StrConst string)
385   =
386     do return()
387
```

```
388  checkExp (Op_or exp exp2)
389    =
390      do
391        checkExp exp
392        checkExp exp2
393        type1 <- getExpType exp
394        type2 <- getExpType exp2
395        if (dataIsBoolType type1) && (dataIsBoolType type2) then
396          return ()
397        else
398          liftEither $ throwError $
399          "two exps in or operation must be in boolean type"
400
401
402  checkExp (Op_and exp exp2)
403    =
404      do
405        checkExp exp
406        checkExp exp2
407        type1 <- getExpType exp
408        type2 <- getExpType exp2
409        if (dataIsBoolType type1) && (dataIsBoolType type2) then
410          return ()
411        else
412          liftEither $ throwError $
413          "two exps in and operation must be in boolean type"
414
415  checkExp (Op_eq exp exp2)
416    =
417      do
418        checkExp exp
419        checkExp exp2
420        type1 <- getExpType exp
421        type2 <- getExpType exp2
422        if (dataIsBolIntType type1) && (dataIsBolIntType type2) then
423          if type1==type2 then
424            return()
425          else
426            liftEither $ throwError $
427            "two exps in eq operation must be same type"
428        else
429          liftEither $ throwError $
430          "two exps in eq operation must be boolean or integer type"
431
432  checkExp (Op_neq exp exp2)
433    =
434      do
435        checkExp exp
436        checkExp exp2
437        type1 <- getExpType exp
438        type2 <- getExpType exp2
439        if (dataIsBolIntType type1) && (dataIsBolIntType type2) then
440          if type1 == type2 then
441            return()
442          else
443            liftEither $ throwError $
```

```
444              "two exps in neq operation must be same type"
445          else
446            liftEither $ throwError $
447              "two exps in neq operation must be boolean or integer type"
448
449  checkExp (Op_less exp exp2)
450      =
451      do
452        checkExp exp
453        checkExp exp2
454        type1 <- getExpType exp
455        type2 <- getExpType exp2
456        if (dataIsBolIntType type1) && (dataIsBolIntType type2) then
457          if type1 == type2 then
458            return()
459          else
460            liftEither $ throwError $
461              "two exps in less operation must be same type"
462        else
463          liftEither $ throwError $
464            "two exps in less operation must be boolean or integer type"
465
466  checkExp (Op_less_eq exp exp2)
467      =
468      do
469        checkExp exp
470        checkExp exp2
471        type1 <- getExpType exp
472        type2 <- getExpType exp2
473        if (dataIsBolIntType type1) && (dataIsBolIntType type2) then
474          if type1 == type2 then
475            return()
476          else
477            liftEither $ throwError $
478              "two exps in <= operation must be same type"
479        else
480          liftEither $ throwError $
481            "two exps in <= operation must be boolean or integer type"
482
483  checkExp (Op_large exp exp2)
484      =
485      do
486        checkExp exp
487        checkExp exp2
488        type1 <- getExpType exp
489        type2 <- getExpType exp2
490        if (dataIsBolIntType type1) && (dataIsBolIntType type2) then
491          if type1 == type2 then
492            return()
493          else
494            liftEither $ throwError $
495              "two exps in large operation must be same type"
496        else
497          liftEither $ throwError $
498            "two exps in large operation must be boolean or integer type"
499
```

```
500  checkExp (Op_large_eq exp exp2)
501    =
502      do
503        checkExp exp
504        checkExp exp2
505        type1 <- getExpType exp
506        type2 <- getExpType exp2
507        if (dataIsBolIntType type1) && (dataIsBolIntType type2) then
508          if type1 == type2 then
509            return()
510          else
511            liftEither $ throwError $
512            "two exps in >= operation must be same type"
513        else
514          liftEither $ throwError $
515          "two exps in >= operation must be boolean or integer type"
516
517  checkExp (Op_add exp exp2)
518    =
519      do
520        checkExp exp
521        checkExp exp2
522        type1 <- getExpType exp
523        type2 <- getExpType exp2
524        if (dataIsIntegerType type1) && (dataIsIntegerType type2) then
525          return ()
526        else
527          liftEither $ throwError $
528          "two exps in add operation must be in integer type"
529
530  checkExp (Op_sub exp exp2)
531    =
532      do
533        checkExp exp
534        checkExp exp2
535        type1 <- getExpType exp
536        type2 <- getExpType exp2
537        if (dataIsIntegerType type1) && (dataIsIntegerType type2) then
538          return ()
539        else
540          liftEither $ throwError $
541          "two exps in sub operation must be in integer type"
542
543  checkExp (Op_mul exp exp2)
544    =
545      do
546        checkExp exp
547        checkExp exp2
548        type1 <- getExpType exp
549        type2 <- getExpType exp2
550        if (dataIsIntegerType type1) && (dataIsIntegerType type2) then
551          return ()
552        else
553          liftEither $ throwError $
554          "two exps in mul operation must be in integer type"
555
```

```haskell
556   checkExp (Op_div exp exp2)
557     =
558       do
559         checkExp exp
560         checkExp exp2
561         type1 <- getExpType exp
562         type2 <- getExpType exp2
563         if (dataIsIntegerType type1)&&(dataIsIntegerType type2) then
564           return ()
565         else
566           liftEither $ throwError $
567           "two exps in div operation must be in integer type"
568
569   checkExp (Op_not exp )
570     =
571       do
572         checkExp exp
573         type1 <- getExpType exp
574         if (dataIsBoolType type1) then
575           return ()
576         else
577           liftEither $ throwError $
578           " exp after not operation must be in boolean type"
579
580   checkExp (Op_neg exp)
581     =
582       do
583         checkExp exp
584         type1 <- getExpType exp
585         if (dataIsIntegerType type1) then
586           return ()
587         else
588           liftEither $ throwError $
589           "exp after neg operation must be in integer type"
590
591   checkExp (Lval lvalue)
592     =
593       do
594         checkLValue lvalue
595
596
597   --------------------------------------------------------------------------------
598   ------------------------- help functions----------------------------------------
599
600   --------------------------------------------------------------------------------
601   -- check arity of a procedure
602   checkArityProcedure :: String -> Int -> SymTableState ()
603   checkArityProcedure procedureName arity
604     =
605       do
606         st <- get
607         let (procedureParams, _) = (pt st) Map.! procedureName
608         let procedureArity = length $ procedureParams
609         if procedureArity == arity then
610           return ()
611         else
```

43

```
612        liftEither $ throwError ("Unmatched arity(" ++ (show arity) ++
613                                  ") for procedure: \"" ++ procedureName ++
614                                  "\" found arity = " ++ (show procedureArity)
615                              )
616
617
618  -- insert records arraies and procedures into symbol table
619  constructSymbolTable :: Program -> SymTableState ()
620  constructSymbolTable prog@(Program records arraies procedures)
621    =
622      do
623        st <- get
624        mapM_ insertRecordType records
625        mapM_ insertArrayType arraies
626        mapM_ insertProcedure procedures
627
628
629  -- function name tell us everyting
630  -- expression is lvalue
631  expIsLvalue :: Exp -> Bool
632  expIsLvalue (Lval lValue) = True
633  expIsLvalue _ = False
634  -- this lvalue is record or array type
635  lvalueIsRerAry :: LValue ->  SymTableState Bool
636  lvalueIsRerAry (LId ident)
637    =
638      do
639        varInfo <- getVariableType ident
640        let (bool,int,vt,int2) = varInfo
641        case vt of
642          RecordVar string1 ->
643            do return True
644          ArrayVar string2 ->
645            do return True
646          _ ->
647            do return False
648  lvalueIsRerAry (LBrackets ident exp)
649    =
650      do
651        varInfo <- getVariableType ident
652        let (bool,int,vt,int2) = varInfo
653        case vt of
654          ArrayVar string2 ->
655            do
656              arrayInfo <- getArrayType string2
657              let (a,arrayType) = arrayInfo
658              case arrayType of
659                AliasDataType aliasType ->
660                  do return True
661                _ ->
662                  do return False
663          _ ->
664            do return False
665
666  lvalueIsRerAry _ = do return False
667  -- This variable is array type
```

```haskell
668  varIsArrayType :: VariableType -> Bool
669  varIsArrayType (ArrayVar _) = True
670  varIsArrayType _ = False
671  -- This variable is record type
672  varIsRecordType :: VariableType -> Bool
673  varIsRecordType (RecordVar _) = True
674  varIsRecordType _ = False
675  -- This data is integer type
676  dataIsIntegerType :: DataType -> Bool
677  dataIsIntegerType (BaseDataType IntegerType) = True
678  dataIsIntegerType _ = False
679  -- This data is boolean or integer type
680  dataIsBolIntType :: DataType -> Bool
681  dataIsBolIntType (BaseDataType IntegerType) = True
682  dataIsBolIntType (BaseDataType BooleanType) = True
683  dataIsBolIntType _=False
684  -- This data is boolean type
685  dataIsBoolType :: DataType -> Bool
686  dataIsBoolType (BaseDataType BooleanType) = True
687  dataIsBoolType _ = False
688  -- The data stored in this array is record type
689  dataIsRecordTypeStoreInArray :: DataType -> Bool
690  dataIsRecordTypeStoreInArray (BaseDataType _) = False
691  dataIsRecordTypeStoreInArray (AliasDataType _) = True
692
693
694
695  --actual parameter type match formal parameter type
696  hasSameElem :: [Exp] -> [(Bool, DataType)] -> SymTableState Bool
697  hasSameElem (x:xs) ((_,y):ys)
698    =
699      do
700        expType<-getExpType x
701        if expType == y then
702          hasSameElem xs ys
703        else
704          return False
705  hasSameElem [] [] = do return True
706  hasSameElem _ _ = do return True
707
708
709  --------------------------------------------------------------------------------
710  --get expression's type(datatype)
711  getExpType :: Exp -> SymTableState DataType
712
713  getExpType (BoolConst _) = do return (BaseDataType BooleanType)
714  getExpType (IntConst _) = do return (BaseDataType IntegerType)
715  getExpType (StrConst _) = do return (BaseDataType StringType)
716  getExpType (Op_or _ _) = do return (BaseDataType BooleanType)
717  getExpType (Op_and _ _) = do return (BaseDataType BooleanType)
718  getExpType (Op_eq  _ _) = do return (BaseDataType BooleanType)
719  getExpType (Op_neq  _ _) = do return (BaseDataType BooleanType)
720  getExpType (Op_less  _ _) = do return (BaseDataType BooleanType)
721  getExpType (Op_less_eq  _ _) = do return (BaseDataType BooleanType)
722  getExpType (Op_large _ _) = do return (BaseDataType BooleanType)
723  getExpType (Op_large_eq _ _) = do return (BaseDataType BooleanType)
```

45

```
724   getExpType (Op_not _) = do return (BaseDataType BooleanType)
725   getExpType (Op_add _ _) = do return (BaseDataType IntegerType)
726   getExpType (Op_sub _ _) = do return (BaseDataType IntegerType)
727   getExpType (Op_mul _ _) = do return (BaseDataType IntegerType)
728   getExpType (Op_div _ _) = do return (BaseDataType IntegerType)
729   getExpType (Op_neg _) = do return (BaseDataType IntegerType)
730   getExpType (Lval lvalue )
731     =
732       do
733         dataInfo <- getDataTypeOfLValue lvalue
734         let (byV,dataType) = dataInfo
735         return dataType
736
737   --get the data type of a lvalue
738   --this should be used after a checkvalue
739
740   getDataTypeOfLValue :: LValue -> SymTableState (Bool,DataType)
741   -- <id>
742   getDataTypeOfLValue (LId varname)
743     =
744       do
745         varInfo <- getVariableType varname
746         let (bool,int,vt,int2) = varInfo
747         let datatype = (varTypeToDataType vt) in return (bool,datatype)
748
749   -- <id>.<id>
750   getDataTypeOfLValue (LDot recordName fieldname)
751     =
752       do
753         c <- getVariableType recordName
754         let (bool, int1, variableType, int2) = c
755         let (RecordVar recordType) = variableType
756         b <- getRecordField recordType fieldname
757         let datatype = fst b in return (bool,(BaseDataType datatype))
758
759
760   -- <id> [Int]
761   getDataTypeOfLValue (LBrackets arrayName int)
762     =
763       do
764         c <- getVariableType arrayName
765         let (bool, int, variableType, int2) = c
766         let (ArrayVar arrayType) = variableType
767         a <- getArrayType arrayType
768         let datatype =snd a in return (bool,datatype)
769   -- <id> [Int]. <id>
770   getDataTypeOfLValue (LBracketsDot arrayName int fieldname)
771     =
772       do
773         c<-getVariableType arrayName
774         let (bool, int, variableType, int2) = c
775         let (ArrayVar arrayType) = variableType
776         a <- getArrayType arrayType
777         let AliasDataType recordName = snd a
778
779         b <- getRecordField recordName fieldname
```

```
780          let datatype = fst b in return (bool,(BaseDataType datatype))
781
782
783  -- variable type-> data type
784  varTypeToDataType :: VariableType -> DataType
785  varTypeToDataType (BooleanVar) = BaseDataType BooleanType
786  varTypeToDataType (IntegerVar) = BaseDataType IntegerType
787  varTypeToDataType (RecordVar alias) = AliasDataType alias
788  varTypeToDataType (ArrayVar alias) = AliasDataType alias
789
790  -- used to report error
791  getLValueName :: LValue -> String
792  getLValueName (LId ident) = ident
793  getLValueName (LDot ident ident2) = ident
794  getLValueName (LBrackets ident exp ) = ident
795  getLValueName (LBracketsDot ident exp ident2) = ident
796
797  -- used to report error
798  getDataT :: DataType -> String
799  getDataT (BaseDataType BooleanType) = "Boolean"
800  getDataT (BaseDataType IntegerType) = "Integer"
801  getDataT (BaseDataType StringType) = "String"
802  getDataT (AliasDataType aliasType ) = aliasType
```

RooParser.hs

```
1   ----------------------------------------------------------------
2   -- COMP90045 Programming Language Implementation Project --
3   --                    Roo Compiler                      --
4   --   Implemented by Xulin Yang                          --
5   --   Implemented by Team: GNU_project                   --
6   ----------------------------------------------------------------
7   module RooParser (ast)
8   where
9   import RooAST
10  import Text.Parsec
11  import Text.Parsec.Language (emptyDef)
12  import Text.Parsec.Expr
13  import qualified Text.Parsec.Token as Q
14  import System.Environment
15  import System.Exit
16  import Debug.Trace (trace)
17
18  type Parser a
19      = Parsec String Int a
20
21  scanner :: Q.TokenParser Int
22  scanner
23      = Q.makeTokenParser
24        (emptyDef
25        { Q.commentLine     = "#"
26        , Q.nestedComments  = True
27        , Q.identStart      = letter
28        -- An identifir is a non-empty sequence of alphanumeric characters,
29        --  underscore and apostrophe ('), and it must start with a (lower or upper
30        --  case) letter.
31        , Q.identLetter     = alphaNum <|> char '_' <|> char '\''
32        , Q.opStart         = oneOf "+-*<"
33        , Q.opLetter        = oneOf "="
34        , Q.reservedNames   = rooReserved
35        , Q.reservedOpNames = rooOpnames
36        })
37
38  whiteSpace    = Q.whiteSpace scanner
39  lexeme        = Q.lexeme scanner
40  natural       = Q.natural scanner
41  identifier    = Q.identifier scanner
42  semi          = Q.semi scanner
43  comma         = Q.comma scanner
44  dot           = Q.dot scanner
45  parens        = Q.parens scanner
46  braces        = Q.braces scanner
47  brackets      = Q.brackets scanner
48  squares       = Q.squares scanner
49  reserved      = Q.reserved scanner
50  reservedOp    = Q.reservedOp scanner
51  stringLiteral = Q.stringLiteral scanner
```

```
52
53  rooReserved, rooOpnames :: [String]
54
55  -- reserved words according to the specification
56  rooReserved
57    = ["and", "array", "boolean", "call", "do", "else", "false", "fi", "if",
58      "integer", "not", "od", "or", "procedure", "read", "record", "then",
59      "true", "val", "while", "write", "writeln"]
60
61  -- reserved operators from specification
62  -- 12 binary oprator (and, or; above reserved string);
63  -- 2 unary: not (above reserved string), -;
64  -- assignment operator <-
65  rooOpnames
66    = [ "+", "-", "*", "/", "=", "!=", "<", "<=", ">", ">=", "<-"]
67
68  -----------------------------------------------------------------
69  --  Note: 0+ is ensured using many/sepBy and 1+ using many1/sepBy1
70  -----------------------------------------------------------------
71
72  -----------------------------------------------------------------
73  -- parse reused base type integer/boolean; no string here as mentioned in AST
74  -- parse reused data type integer/boolean/type alias
75  -----------------------------------------------------------------
76  pBaseType :: Parser BaseType
77  pBaseType
78    = do
79        reserved "boolean"
80        return BooleanType
81    <|>
82    do
83        reserved "integer"
84        return IntegerType
85    <?>
86      "base type"
87
88  pDataType :: Parser DataType
89  pDataType
90    =
91      do
92        baseType <- pBaseType
93        return (BaseDataType baseType)
94    <|>
95      do
96        alias <- identifier
97        return (AliasDataType alias)
98    <?>
99      "data type"
100
101
102  -----------------------------------------------------------------
103  -- parse literals
104  -----------------------------------------------------------------
105  pBool :: Parser Bool
106  pBool
107    =
```

```
108      do { reserved "true"; return (True) }
109      <|>
110      do { reserved "false"; return (False) }
111      <?>
112         "boolean literal"
113
114  pInt :: Parser Int
115  pInt
116    =
117       do
118         n <- natural <?> "number"
119         return (fromInteger n :: Int)
120       <?>
121         "Integer Literal"
122
123  -- don't accept newline, tab, quote but "\n", "\t", "\"" <- two character
124  -- string should still be accepted
125  pcharacter :: Parser String
126  pcharacter
127    =
128       try(
129         do
130           string ('\\':['n'])
131           return (['\\', 'n'])
132       )
133       <|>
134       try(
135         do
136           string ('\\':['t'])
137           return (['\\', 't'])
138       )
139       <|>
140       try(
141         do
142           string ('\\':['"'])
143           return (['\\', '"'])
144       )
145       <|>
146       do
147         c <- noneOf ['\n', '\t', '"']
148         return ([c])
149       <?>
150         "any character except newline, tab, quote"
151
152  -- Parser for string
153  pString :: Parser String
154  pString
155    =
156       do
157         -- String is surrounded by two quotes
158         char '"'
159         -- Parse characters except newline / tab characters and quotes
160         str <- many pcharacter
161         char '"' <?> "\'\"\' to wrap the string"
162         whiteSpace -- consumes following spaces
163         return (concat str)
```

50

```haskell
164        <?>
165          "string cannot has newline, quote, tab"
166
167    --------------------------------------------------------------------
168    -- An lvalue (<lvalue>) has four (and only four) possible forms:
169    --        <id>
170    --        <id>.<id>
171    --        <id>[<exp>]
172    --        <id>[<exp>].<id>
173    -- An example lvalue is point[0].xCoord
174    --------------------------------------------------------------------
175    pLValue :: Parser LValue
176    pLValue
177      = try (
178          do
179            lValue <- pLBracketsDot
180            return lValue
181        )
182      <|>
183        try (
184          do
185            lValue <- pLBrackets
186            return lValue
187        )
188      <|>
189        try (
190          do
191            lValue <- pLDot
192            return lValue
193        )
194      <|>
195        do
196          lValue <- pLId
197          return lValue
198        <?>
199          "LValue"
200
201    pLBracketsDot :: Parser LValue
202    pLBracketsDot
203      =
204        do
205          ident1 <- identifier
206          exp <- brackets pExp
207          dot
208          ident2 <- identifier
209          return (LBracketsDot ident1 exp ident2)
210        <?>
211          "LBracketsDot"
212
213    pLBrackets :: Parser LValue
214    pLBrackets
215      =
216        do
217          ident <- identifier
218          exp <- brackets pExp
219          return (LBrackets ident exp)
```

```
220        <?>
221          "pLBrackets"
222
223    pLDot :: Parser LValue
224    pLDot
225      =
226        do
227          ident1 <- identifier
228          dot
229          ident2 <- identifier
230          return (LDot ident1 ident2)
231        <?>
232          "pLDot"
233
234    pLId :: Parser LValue
235    pLId
236      =
237        do
238          ident <- identifier
239          return (LId ident)
240        <?>
241          "pLId"
242
243    --------------------------------------------------------------
244    --  pExp is the main parser for expression.
245    --
246    --  It is built using Parces's powerful
247    --  buildExpressionParser and takes into account the operator
248    --  precedences and associativity specified in 'opTable' below.
249    --------------------------------------------------------------
250    prefix name fun
251      = Prefix (do { reservedOp name
252                   ; return fun
253                   }
254             )
255
256    binary name op
257      = Infix (do { reservedOp name
258                  ; return op
259                  }
260            ) AssocLeft
261
262    relation name rel
263      = Infix (do { reservedOp name
264                  ; return rel
265                  }
266            ) AssocNone
267
268    -- expression operators:
269    --     All the operators on the same line have the same precedence,
270    --         and the ones on later lines have lower precedence;
271    --     The six relational operators are non-associative
272    --         so, for example, a = b = c is not a well-formed expression).
273    --     The six remaining binary operators are left-associative.
274    -- -                 |unary          |
275    -- * /               |binary and infix |left-associative
```

```
276   -- + -                /binary and infix /left-associative
277   -- = != < <= > >= /binary and infix /relational, non-associative
278   -- not                /unary          /
279   -- and                /binary and infix /left-associative
280   -- or                 /binary and infix /left-associative
281   opTable
282     = [ [ prefix    "-"   Op_neg    ]
283       , [ binary   "*"   Op_mul    , binary   "/"  Op_div  ]
284       , [ binary   "+"   Op_add    , binary   "-"  Op_sub  ]
285       , [ relation "="   Op_eq     , relation "!=" Op_neq  ,
286           relation "<"   Op_less
287         , relation "<="  Op_less_eq, relation ">"  Op_large,
288           relation ">="  Op_large_eq ]
289       , [ prefix    "not" Op_not    ]
290       , [ binary   "and" Op_and    ]
291       , [ binary   "or"  Op_or     ]
292       ]
293
294   pExp :: Parser Exp
295   pExp
296     = buildExpressionParser opTable pFac
297       <?>
298         "expression"
299
300   pFac :: Parser Exp
301   pFac
302     = choice [parens pExp, -- ( <exp> )
303               pLval,
304               pBoolConst,
305               pIntConst,
306               pStrConst,
307               pNeg          -- used to parse expression like ------1 (arbitrary
308                             -- unary minus before expression)
309             ]
310       <?>
311         "simple expression"
312
313   pLval, pBoolConst, pIntConst, pStrConst :: Parser Exp
314   pLval
315     =
316       do
317         lval <- pLValue
318         return (Lval lval)
319       <?>
320         "lval expression"
321
322   pBoolConst
323     =
324       do
325         b <- pBool
326         return (BoolConst b)
327       <?>
328         "bool const/literal expression"
329
330   pIntConst
331     =
```

```
332      do
333        i <- pInt
334        return (IntConst i)
335      <?>
336        "int const/literal expression"
337
338  pStrConst
339    =
340      do
341        s <- pString
342        return (StrConst s)
343      <?>
344        "string const/literal expression"
345
346  -- parse arbitrary unary minus in pFac
347  pNeg :: Parser Exp
348  pNeg
349    =
350      do
351      reservedOp "-"
352      exp <- pExp
353      return (Op_neg exp)
354    <?>
355      "unary minus"
356
357
358  -----------------------------------------------------------------
359  --  pStmt is the main parser for statements.
360  --  Statement related parsers
361  -----------------------------------------------------------------
362  pStmt, pStmtAtom, pStmtComp :: Parser Stmt
363  -- atom statement:
364  --     <lvalue> <- <exp> ;
365  --     read <lvalue> ;
366  --     write <exp> ;
367  --     writeln <exp> ;
368  --     call <id> ( <exp-list> ) ;
369  --         where <exp-list> is a (possibly empty) comma-separated list of
370    --            expressions.
371  -- composite statement:
372  --     if <expr> then <stmt-list> else <stmt-list> fi
373  --     if <exp> then <stmt-list> fi # just make second [Stmt] empty
374  --     while <expr> do <stmt-list> od
375  --         where <stmt-list> is a non-empty sequence of statements, atomic or
376  --             composite
377  pStmt = choice [pStmtAtom, pStmtComp]
378
379  pStmtAtom
380    =
381      do
382      r <- choice [pAsg, pRead, pWrite, pWriteln, pCall]
383      -- all atomic stmt's semicolon is comsumed here
384      semi
385      return r
386    <?>
387      "atomic statement"
```

```
388
389  pAsg, pRead, pWrite, pWriteln, pCall :: Parser Stmt
390
391  -- parse: <lvalue> <- <exp> ;
392  pAsg
393    =
394      do
395        lvalue <- pLValue
396        reservedOp "<-"
397        rvalue <- pExp
398        return (Assign lvalue rvalue)
399      <?>
400        "assign"
401
402  -- parse: read <lvalue> ;
403  pRead
404    = do
405        reserved "read"
406        lvalue <- pLValue
407        return (Read lvalue)
408      <?>
409        "read"
410
411  -- parse: write <exp> ;
412  pWrite
413    = do
414        reserved "write"
415        expr <- pExp
416        return (Write expr)
417      <?>
418        "write"
419
420  -- parse: writeln <exp> ;
421  pWriteln
422    = do
423        reserved "writeln"
424        expr <- pExp
425        return (Writeln expr)
426      <?>
427        "writeln"
428
429  -- parse: call <id>(<exp-list>) ;
430  pCall
431    = do
432        reserved "call"
433        ident <- identifier
434        -- 0+ comma-separated list of expressions
435        exprs <- parens (pExp `sepBy` comma)
436        return (Call ident exprs)
437      <?>
438        "call"
439
440  pStmtComp = (choice [pIf, pWhile]) <?> "composite statement"
441
442  pIf, pWhile :: Parser Stmt
443
```

```
444   -- parse:
445   --    if <exp> then <stmt-list> else <stmt-list> fi
446   --    if <exp> then <stmt-list> fi
447   pIf
448     =
449       do
450         reserved "if"
451         exp <- pExp
452         reserved "then"
453         thenStmts <- many1 pStmt
454         -- check if there is an else statment
455         -- if not, return empty
456         res <- (
457           do
458             reserved "fi"
459             return (IfThen exp thenStmts)
460           <|>
461           do
462             reserved "else"
463             -- else body can not be empty
464             elseStmts <- many1 pStmt
465             reserved "fi"
466             return (IfThenElse exp thenStmts elseStmts)
467         )
468         return res
469       <?>
470         "if"

472   -- parse: while <exp> do <stmt-list> od
473   pWhile
474     = do
475         reserved "while"
476         exp <- pExp
477         reserved "do"
478         stmts <- many1 pStmt -- a 1+ sequence of statements, atomic or composite
479         reserved "od"
480         return (While exp stmts)
481       <?>
482         "while"

484   --------------------------------------------------------------------
485   --  Procedure related parser
486   --------------------------------------------------------------------
487   -- Each formal parameter has two components (in the given order):
488   -- 1. a parameter type/mode indicator, which is one of these five:
489   --    a) a type alias,
490   --    b) boolean,
491   --    c) integer,
492   --    d) boolean val
493   --    e) integer val
494   -- 2. an identifier
495   pParameter :: Parser Parameter
496   pParameter
497     =
498       try(
499         do
```

```
500         -- parse boolean val variable
501         reserved "boolean"
502         reserved "val"
503         name <- identifier
504         return (BooleanVal name)
505       )
506     <|>
507     try(
508       do
509         -- parse integer val variable
510         reserved "integer"
511         reserved "val"
512         name <- identifier
513         return (IntegerVal name)
514       )
515     <|>
516     do
517       -- parse boolean/integer/type_alias variable
518       paraType <- pDataType
519       name <- identifier
520       return (DataParameter paraType name)
521     <?>
522       "parameter"
523
524 -- The header has two components (in this order):
525 --    1. an identifier (the procedure's name), and
526 --    2. a comma-separated list of 0+ formal parameters within a pair
527 --         of parentheses (so the parentheses are always present).
528 pProcedureHeader :: Parser ProcedureHeader
529 pProcedureHeader
530   =
531     do
532       procedureName <- identifier
533       parameters <- parens (pParameter `sepBy` comma)
534       return (ProcedureHeader procedureName parameters)
535     <?>
536       "procedure header"
537
538 -- A variable declaration consists of
539 --    a) a type name (boolean, integer, or a type alias),
540 --    b) followed by a 1+ comma-separated list of
541 --         identifiers,
542 --    i)   the list terminated with a semicolon.
543 --    ii) There may be any number of variable declarations, in any order.
544 pVariable :: Parser VariableDecl
545 pVariable
546   =
547     do
548       varType <- pDataType
549       varNames <- (identifier `sepBy1` comma)
550       semi
551       return (VariableDecl varType varNames)
552     <?>
553       "variable"
554
555
```

```
556   -- procedure body consists of 0+ local variable declarations,
557   --    1. A variable declaration consists of
558   --       a) a type name (boolean, integer, or a type alias),
559   --       b) followed by a 1+ comma-separated list of identifiers,
560   --          i)  the list terminated with a semicolon.
561   --          ii) There may be any number of variable declarations, in any order.
562   --    2. followed by a 1+ sequence of statements,
563   pProcedureBody :: Parser ProcedureBody
564   pProcedureBody
565     =
566       do
567         vars <- many pVariable
568         stmts <- braces (many1 pStmt)
569         return (ProcedureBody vars stmts)
570     <?>
571       "procedure body"
572
573   -- Each procedure consists of (in the given order):
574   --    1. the keyword procedure,
575   --    2. a procedure header, and
576   --    3. a procedure body.
577   pProcedure :: Parser Procedure
578   pProcedure
579     =
580       do
581         reserved "procedure"
582         procedureHeader <- pProcedureHeader
583         procedureBody <- pProcedureBody
584         return (Procedure procedureHeader procedureBody)
585     <?>
586       "procedure"
587
588   -----------------------------------------------------------------
589   --  Array related parser
590   -----------------------------------------------------------------
591   -- array type definition consists of (in the given order):
592   --    1. the keyword array,
593   --    2. a (positive) integer literal enclosed in square brackets,
594   --    3. a type name which is either an identifier (a type alias) or one of
595   --       boolean and integer,
596   --    4. an identifier (giving a name to the array type), and
597   --    5. a semicolon.
598   pArray :: Parser Array
599   pArray
600     =
601       do
602         reserved "array"
603         pos <- getPosition
604         arraySize <- brackets pInt
605         -- need to check arraySize > 0 (positive integer)
606         if arraySize <= 0
607         then
608           error ("array size sould not be <= 0 at line: " ++
609             (show (sourceLine pos))
610             ++ ", column: " ++ (show (sourceColumn pos + 1))) -- +1 to skip '['
611         else do
```

```
612      arrayType <- pDataType
613      arrayName <- identifier
614      semi
615      return (Array arraySize arrayType arrayName)
616   <?>
617      "array"
618
619   ----------------------------------------------------------------
620   --  Record related parser
621   ----------------------------------------------------------------
622   -- field declaration is of:
623   --   1. boolean or integer
624   --   2. followed by an identifier (the field name).
625   pFieldDecl :: Parser FieldDecl
626   pFieldDecl
627     =
628       do
629         fieldType <- pBaseType
630         fieldName <- identifier
631         return (FieldDecl fieldType fieldName)
632       <?>
633         "field declaration"
634
635   -- record consists of:
636   --   1. the keyword record,
637   --   2. a 1+ list of field declarations, separated by semicolons,
638   --        the whole list enclosed in braces,
639   --   3. an identifier, and
640   --   4. a semicolon.
641   pRecord :: Parser Record
642   pRecord
643     =
644       do
645         reserved "record"
646         recordFieldDecls <- braces (pFieldDecl `sepBy1` semi)
647         recordName <- identifier
648         semi
649         return (Record recordFieldDecls recordName)
650       <?>
651         "record"
652
653   ----------------------------------------------------------------
654   --  Program related parser
655   ----------------------------------------------------------------
656   -- A Roo program consists of
657   --   1. 0+ record type definitions, followed by
658   --   2. 0+ array type definitions, followed by
659   --   3. 1+ procedure definitions.
660   pProgram :: Parser Program
661   pProgram
662     = do
663         records <- many pRecord
664         arraies <- many pArray
665         procedures <- many1 pProcedure
666         return (Program records arraies procedures)
667       <?>
```

```
668        "program"
669
670
671    ------------------------------------------------------------------
672    -- main (given skeleton code)
673    ------------------------------------------------------------------
674
675    rooParse :: Parser Program
676    rooParse
677      = do
678          whiteSpace
679          p <- pProgram
680          eof
681          return p
682
683    ast :: String -> Either ParseError Program
684    ast input
685      =  runParser rooParse 0 "" input
686
687    pMain :: Parser Program
688    pMain
689      = do
690          whiteSpace
691          p <- pProgram
692          eof
693          return p
694
695    main :: IO ()
696    main
697      = do { progname <- getProgName
698           ; args <- getArgs
699           ; checkArgs progname args
700           ; input <- readFile (head args)
701           ; let output = runParser pMain 0 "" input
702           ; case output of
703               Right ast -> print ast
704               Left  err -> do { putStr "Parse error at "
705                               ; print err
706                               }
707           }
708
709    checkArgs :: String -> [String] -> IO ()
710    checkArgs _ [filename]
711      = return ()
712    checkArgs progname _
713      = do { putStrLn ("Usage: " ++ progname ++ " filename\n\n")
714           ; exitWith (ExitFailure 1)
715           }
```

===

SymbolTable.hs

===

```
1   ----------------------------------------------------------
2   -- COMP90045 Programming Language Implementation Project --
3   --                  Roo Compiler                        --
4   --  Implemented by Xulin Yangm, Wenrui Zhang            --
5   --  Implemented by Team: GNU_project                    --
6   ----------------------------------------------------------
7   module SymbolTable where
8
9   import Control.Monad
10  import Control.Monad.State
11  import Control.Monad.Except
12  import Data.Map (Map, (!))
13  import qualified Data.Map as Map
14  import Text.Parsec.Pos
15  import RooAST
16  import OzCode
17
18  -- ------------------------------------------------------------------------------
19  -- Termonology:
20  -- 0. st: symbol table
21  -- 1. global type table: holds information about type aliases and the composite
22  --    types then name;
23  --     a) att: global alias type table
24  --     b) rft: global record field table
25  -- 2. global procedure table: holds procedure parameter type, whether by
26  --    reference information
27  --     a) pt: global procedure table
28  -- 3. local variable table: which provides information about formal parameters
29  --    and variables in the procedure that is currently being processed.
30  --     a) lts: stack of local variable tables
31  --     b) vtt: variable type table
32  --     c) cvt: current procedure's variable table
33  -- ------------------------------------------------------------------------------
34
35  data CompositeKey = CompositeKey String String
36    deriving (Show, Eq, Ord)
37
38  type SymTableState a = StateT SymTable (Either String) a
39
40  -- A short hand form for variable type
41  data VariableType = BooleanVar
42                    | IntegerVar
43                    | RecordVar String
44                    | ArrayVar String
45            deriving (Show, Eq)
46
47  -- 1. available slot number
48  -- 2. available register number
49  -- 3. mapping of variable name with
50  --     a) true if it is pass by value and
51  --     b) allocated slot number and
```

```haskell
52  --       c) its type and
53  --       d) #elements for array, #fields for record, 1 for boolean/integer
54  data LocalVariableTable
55    = LocalVariableTable
56      { slotCounter :: Int
57      , registerCounter :: Int
58      , vtt :: Map String (Bool, Int, VariableType, Int)
59      }
60
61  -- Array:  array size, type
62  -- Record: #fields,    [field's definition]
63  data AliasTypeInfo
64    = ArrayInfo  (Int, DataType)
65    | RecordInfo (Int, [FieldDecl])
66
67  -- 1. att  :: global alias type table
68  -- 2. rft  :: global record field table
69  --            = map of (record name, field name) with
70  --                 a) field's type and
71  --                 b) index of the field in record
72  -- 3. pt   :: global procedure table
73  --            = map of procedure name with
74  --                 a) [true if pass by value, parameter's type]
75  --                 b) procedure's deifnition
76  -- 4. lvts :: stack of local variable table
77  data SymTable
78    = SymTable
79      { att :: Map String AliasTypeInfo
80      , rft :: Map CompositeKey (BaseType, Int)
81      , pt  :: Map String ([(Bool, DataType)], Procedure)
82      , lvts :: [LocalVariableTable]
83      , labelCounter :: Int
84      , instructions :: [OzInstruction]
85      }
86
87  initialSymTable :: SymTable
88  initialSymTable = SymTable { att = Map.empty
89                             , rft = Map.empty
90                             , pt  = Map.empty
91                             , lvts = []
92                             , labelCounter = 0
93                             , instructions = []
94                             }
95
96  initialLocalVariableTable :: LocalVariableTable
97  initialLocalVariableTable = LocalVariableTable
98    { slotCounter     = 0
99    , registerCounter = 0
100   , vtt             = Map.empty
101   }
102
103 -- ----------------------------------------------------------------------------
104 -- TypeTable related helper methods
105 -- ----------------------------------------------------------------------------
106
107 insertArrayType :: Array -> SymTableState ()
```

```
108  insertArrayType (Array arraySize dataType arrayName)
109      =
110          do
111              st <- get
112              -- duplicate array definition
113              if (Map.member arrayName (att st)) then
114                  liftEither $ throwError ("Duplicated alias type: " ++ arrayName)
115              -- insert an array definition
116              else
117                  put $ st { att =  Map.insert arrayName
118                                              (ArrayInfo (arraySize, dataType))
119                                              (att st)
120                          }
121
122  getArrayType :: String -> SymTableState (Int, DataType)
123  getArrayType arrayName
124      =
125          do
126              st <- get
127              -- get an array definition
128              if (Map.member arrayName (att st)) then
129                  let (ArrayInfo info) = ((att st) Map.! arrayName) in return info
130              -- no array definition
131              else
132                  liftEither $ throwError $ "Array named " ++ arrayName ++
133                                              " does not exist"
134
135  insertRecordType :: Record -> SymTableState ()
136  insertRecordType (Record fieldDecls recordName)
137      =
138          do
139              st <- get
140              let recordSize = length fieldDecls
141              -- duplicate record definition
142              if (Map.member recordName (att st)) then
143                  liftEither $ throwError $ "Duplicated alias type: " ++ recordName
144              -- insert a record definition
145              else
146                  do
147                      put $ st { att = Map.insert recordName
148                                              (RecordInfo (recordSize, fieldDecls))
149                                              (att st)
150                              }
151                      insertRecordFields recordName fieldDecls 0
152
153  getRecordType :: String -> SymTableState (Int, [FieldDecl])
154  getRecordType recordName
155      =
156          do
157              st <- get
158              -- get an record definition
159              if (Map.member recordName (att st)) then
160                  let (RecordInfo info) = (att st) Map.! recordName in return info
161              -- no record definition
162              else
163                  liftEither $ throwError $ "Record named " ++ recordName ++
```

```
164                              " does not exist"
165
166  insertRecordFields :: String -> [FieldDecl] -> Int -> SymTableState ()
167  insertRecordFields _ [] _ = return ()
168  insertRecordFields recordName (x:xs) index
169    =
170       do
171          insertRecordField recordName x index
172          insertRecordFields recordName xs (index+1)
173
174  insertRecordField :: String -> FieldDecl -> Int -> SymTableState ()
175  insertRecordField recordName (FieldDecl baseType fieldName) index
176    =
177       do
178          st <- get
179          let ck = CompositeKey recordName fieldName
180          -- duplicate (record name, field name) definition
181          if (Map.member ck (rft st)) then
182            liftEither $ throwError $ "Duplicated record field: " ++
183                                  recordName ++ "." ++ fieldName
184          -- insert a (record name, field name) definition
185          else
186            put $ st { rft = Map.insert ck (baseType, index) (rft st) }
187
188  getRecordField :: String -> String -> SymTableState (BaseType, Int)
189  getRecordField recordName fieldName
190    =
191       do
192          st <- get
193          let ck = CompositeKey recordName fieldName
194          -- get a (record name, field name) definition
195          if (Map.member ck (rft st)) then
196            return $ (rft st) Map.! ck
197          -- no (record name, field name) definition
198          else
199            liftEither $ throwError $ "Record.field: " ++
200                                  recordName ++ "." ++ fieldName ++
201                                  " does not exist"
202
203  getTypeAlias :: String -> SymTableState VariableType
204  getTypeAlias typeName
205    =
206       do
207          st <- get
208          if (Map.member typeName (att st)) then
209            do
210              case (att st) Map.! typeName of
211                (RecordInfo _) -> return (RecordVar typeName)
212                (ArrayInfo  _) -> return (ArrayVar  typeName)
213          else
214            liftEither $ throwError $ "Undefiend alias type: " ++ typeName
215
216  -- return label counter and auto step with +1
217  getlabelCounter :: SymTableState String
218  getlabelCounter
219    =
```

```
220      do
221        st <- get
222        let currentCount = (labelCounter st)
223        put st{labelCounter = currentCount + 1}
224        return $ "label_" ++ show currentCount
225
226    -- --------------------------------------------------------------------------------
227    -- ProcedureTable related helper methods
228    -- --------------------------------------------------------------------------------
229    -- insert a procedure's definition as well as identifying whether parameters
230    -- are pass by value/reference
231    insertProcedure :: Procedure -> SymTableState ()
232    insertProcedure p@(Procedure (ProcedureHeader ident params) _)
233      = do
234        let formalParams = map createformalParam params
235        putProcedure ident formalParams p
236
237    putProcedure :: String -> [(Bool, DataType)] -> Procedure -> SymTableState ()
238    putProcedure procedureName formalParams p
239      = do
240          st <- get
241          -- duplicate procedure definition
242          if (Map.member procedureName (pt st)) then
243            liftEither $ throwError $ "Duplicated procedure name: " ++
244                                      procedureName
245          -- insert a procedure definition
246          else
247            put $ st {pt = Map.insert procedureName (formalParams, p) (pt st)}
248
249    -- get procedure's type info
250    getProcedure :: String -> SymTableState ([(Bool, DataType)], Procedure)
251    getProcedure procedureName
252      = do
253          st <- get
254          if (Map.member procedureName (pt st)) then
255            return $ (pt st) Map.! procedureName
256          else
257            liftEither $ throwError $ "Procedure named " ++ procedureName ++
258                                      " does not exist"
259
260    -- convert Parameter definted in AST to a tuple (is passed by value, type)
261    createformalParam :: Parameter -> (Bool, DataType)
262    createformalParam (BooleanVal _) = (True, BaseDataType BooleanType)
263    createformalParam (IntegerVal _) = (True, BaseDataType IntegerType)
264    createformalParam (DataParameter dataType _) = (False, dataType)
265
266    -- --------------------------------------------------------------------------------
267    -- VariableTable related helper methods
268    -- --------------------------------------------------------------------------------
269    -- push and pop to mimic a stack's behavior
270    pushLocalVariableTable :: SymTableState ()
271    pushLocalVariableTable
272      =
273        do
274          st <- get
275          let newLvts = (lvts st) ++ [initialLocalVariableTable]
```

```
276        put $ st { lvts = newLvts }
277
278   popLocalVariableTable :: SymTableState ()
279   popLocalVariableTable
280     =
281       do
282         st <- get
283         let newLvts = init (lvts st)
284         put $ st { lvts = newLvts }
285
286   getCurVariableTable :: SymTableState LocalVariableTable
287   getCurVariableTable
288     =
289       do
290         st <- get
291         return $ last $ lvts st
292
293   updateCurVariableTable :: LocalVariableTable -> SymTableState ()
294   updateCurVariableTable newLVT
295     =
296       do
297         popLocalVariableTable
298         st <- get
299         let newLvts = (lvts st) ++ [newLVT]
300         put $ st { lvts = newLvts }
301
302   -- check variable name not exist in the local variable table
303   checkVariableNotDefined :: String -> SymTableState ()
304   checkVariableNotDefined varName
305     =
306       do
307         cvt <- getCurVariableTable
308         if (Map.member varName (vtt cvt)) then
309           liftEither $ throwError $ "Duplicated variable name: " ++ varName
310         else
311           return ()
312
313   -- get variable's type information by variable's name
314   getVariableType :: String -> SymTableState (Bool, Int, VariableType, Int)
315   getVariableType varName
316     =
317       do
318         cvt <- getCurVariableTable
319         if (Map.member varName (vtt cvt)) then
320           return $ (vtt cvt) Map.! varName
321         else
322           liftEither $ throwError $ "Unknown variable name: " ++ varName
323
324   -- get variable's type information by variable's name for records
325   -- (e.g. student.id)
326   getVarRecordField :: String -> String -> SymTableState (BaseType, Int)
327   getVarRecordField varName fieldName
328     =
329       do
330         st <- get
331         cvt <- getCurVariableTable
```

```
332        (_, _, varType, _) <- getVariableType varName
333        case varType of
334          (RecordVar recordName) -> getRecordField recordName varName
335          _ -> liftEither $ throwError $ "Variable name: " ++ varName ++
336                                         " is not field type"

338  getSlotCounter :: SymTableState Int
339  getSlotCounter
340    =
341      do
342        cvt <- getCurVariableTable
343        return (slotCounter cvt)

345  -- return the current register counter and increase register counter by 1
346  getRegisterCounter :: SymTableState Int
347  getRegisterCounter
348    =
349      do
350        cvt <- getCurVariableTable
351        let regCounter = registerCounter cvt
352        if regCounter >= 1024 then
353          liftEither $ throwError $ "Register used > 1024"
354        else
355          do
356            updateCurVariableTable cvt { registerCounter = regCounter + 1 }
357            return regCounter

359  setRegisterCounter :: Int -> SymTableState ()
360  setRegisterCounter newReg
361    =
362      do
363        cvt <- getCurVariableTable
364        updateCurVariableTable cvt { registerCounter = newReg }

366  -- -----------------------------------------------------------------------------
367  -- VariableTable construction methods
368  -- -----------------------------------------------------------------------------
369  -- insert procedure's parameters and local variables
370  insertProcedureVariable :: Procedure -> SymTableState ()
371  insertProcedureVariable (Procedure (ProcedureHeader ident params)
372                                     (ProcedureBody variableDecls _ ))
373    =
374      do
375        mapM_ insertProcedureParameter params
376        mapM_ insertProcedureVariableDecl variableDecls

378  -- insert procedure's parameter and identifying whether it is pass by
379  -- value/reference
380  insertProcedureParameter :: Parameter -> SymTableState ()
381  insertProcedureParameter (BooleanVal varName)
382    = insertVariable BooleanVar True varName
383  insertProcedureParameter (IntegerVal varName)
384    = insertVariable IntegerVar True varName
385  insertProcedureParameter (DataParameter (BaseDataType BooleanType)
386                                          varName)
387    = insertVariable BooleanVar False varName
```

```
388  insertProcedureParameter (DataParameter (BaseDataType IntegerType)
389                                            varName)
390    = insertVariable IntegerVar False varName
391  insertProcedureParameter (DataParameter (AliasDataType typeName)
392                                            varName)
393    =
394      do
395        aliasType <- getTypeAlias typeName
396        insertVariable aliasType False varName
397
398  -- insert procedure's local variables and they are pass by value by default
399  insertProcedureVariableDecl :: VariableDecl -> SymTableState ()
400  insertProcedureVariableDecl (VariableDecl (BaseDataType BooleanType)
401                                             variableNames)
402    =
403      do
404        mapM_ (insertVariable BooleanVar True) variableNames
405  insertProcedureVariableDecl (VariableDecl (BaseDataType IntegerType)
406                                             variableNames)
407    =
408      do
409        mapM_ (insertVariable IntegerVar True) variableNames
410  insertProcedureVariableDecl (VariableDecl (AliasDataType typeName)
411                                             variableNames)
412    =
413      do
414        aliasType <- getTypeAlias typeName
415        mapM_ (insertVariable aliasType True) variableNames
416
417  -- insert a variable's type info to the local variable table
418  insertVariable ::  VariableType -> Bool -> String -> SymTableState ()
419  insertVariable BooleanVar byValue varName
420    =
421      do
422        checkVariableNotDefined varName
423        availableSlot <- getSlotCounter
424        -- no matter it is pass by value or by reference, 1 slot required as it
425        -- is boolean
426        let newSlotCounter = availableSlot + 1
427        updateNewVariableToLVT newSlotCounter
428                                varName
429                                (byValue, availableSlot, BooleanVar, 1)
430  insertVariable IntegerVar byValue varName
431    =
432      do
433        checkVariableNotDefined varName
434        availableSlot <- getSlotCounter
435        -- no matter it is pass by value or by reference, 1 slot required as it
436        -- is integer
437        let newSlotCounter = availableSlot + 1
438        updateNewVariableToLVT newSlotCounter
439                                varName
440                                (byValue, availableSlot, IntegerVar, 1)
441  insertVariable recVar@(RecordVar recordName) byValue varName
442    =
443      do
```

```
444        checkVariableNotDefined varName
445        availableSlot <- getSlotCounter
446        (recordSize, _) <- getRecordType recordName
447        if byValue then
448          updateNewVariableToLVT (availableSlot + recordSize)
449                                 varName
450                                 (byValue, availableSlot, recVar, recordSize)
451        else -- pass by reference, only allocate 1 slot
452          updateNewVariableToLVT (availableSlot + 1 )
453                                 varName
454                                 (byValue, availableSlot, recVar, recordSize)
455  insertVariable arr@(ArrayVar arrayName) byValue varName
456    =
457      do
458        checkVariableNotDefined varName
459        availableSlot <- getSlotCounter
460        (arraySize, arrayType) <- getArrayType arrayName
461        case arrayType of
462          BaseDataType _ ->
463            do
464              if byValue then
465                updateNewVariableToLVT (availableSlot + arraySize)
466                                       varName
467                                       (byValue, availableSlot, arr, arraySize)
468              else -- pass by reference, only allocate 1 slot
469                updateNewVariableToLVT (availableSlot + 1)
470                                       varName
471                                       (byValue, availableSlot, arr, arraySize)
472          -- as an array cannot has alias type: array
473          AliasDataType recordName ->
474            do
475              (recordSize, _) <- getRecordType recordName
476              -- array size * #record's fields
477              let nSlotsRequired = recordSize * arraySize
478              if byValue then
479                updateNewVariableToLVT (availableSlot + nSlotsRequired)
480                                       varName
481                                       (byValue, availableSlot, arr,
482                                       nSlotsRequired)
483              else -- pass by reference, only allocate 1 slot
484                updateNewVariableToLVT (availableSlot + 1)
485                                       varName
486                                       (byValue, availableSlot, arr,
487                                       nSlotsRequired)
488
489  updateNewVariableToLVT :: Int -> String -> (Bool, Int, VariableType, Int)
490                            -> SymTableState ()
491  updateNewVariableToLVT newSlotCounter
492                     varName
493                     (byValue, availableSlot, varType, slotRequired)
494    =
495      do
496        cvt <- getCurVariableTable
497        updateCurVariableTable cvt
498          { slotCounter = newSlotCounter
499          , vtt = Map.insert varName
```

```
                              (byValue, availableSlot, varType, slotRequired)
                              (vtt cvt)
          }


-- ------------------------------------------------------------------------
-- instructions related helper methods
-- ------------------------------------------------------------------------
appendInstruction :: OzInstruction -> SymTableState ()
appendInstruction newIns
  =
     do
        st <- get
        let oldIns = instructions st
        put $ st { instructions = oldIns ++ [newIns] }
```