\_\_\_\_\_\_

\_\_\_\_\_\_

## PrettyRoo.hs

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

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

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

```
-> False
164
     isHigherPrecendence exp1@(Op_large_eq _ _) exp2 =
165
       case exp2 of
166
         (Op_not _ ) -> <u>True</u>
167
         (Op_and _ _) -> <u>True</u>
168
         (Op_or _ _) -> <u>True</u>
169
         -- constants has lowest precendence
170
         (<u>Lval</u> _) -> <u>True</u>
171
         (BoolConst _) -> True
172
         (IntConst _) -> True
173
         (StrConst _) -> True
174
                          -> False
175
     isHigherPrecendence exp1@(Op_not _) exp2 =
176
       case exp2 of
177
         (Op_and _ _) -> <u>True</u>
         (Op_or _ _) -> <u>True</u>
179
         -- constants has lowest precendence
180
         (Lval ) -> True
181
         (BoolConst _) -> True
182
         (<u>IntConst</u> _) -> <u>True</u>
183
         (StrConst _) -> True
184
                         -> False
185
     isHigherPrecendence exp1@(Op_and _ _) exp2 =
186
       case exp2 of
187
         (Op_or _ _) -> <u>True</u>
188
         -- constants has lowest precendence
189
         (<u>Lval</u> _) -> <u>True</u>
190
         (BoolConst _) -> True
191
         (IntConst ) -> True
192
         (StrConst _) -> True
193
                          -> False
194
     isHigherPrecendence exp1@(Op_or _ _) exp2
        -- constants has lowest precendence
196
       | (not (hasOperatorExp exp2)) = True
197
       | otherwise = False
198
     isHigherPrecendence _ _ = False
199
200
     -- exp1 has same precendence as exp2
201
     isSamePrecendence :: Exp -> Exp -> Bool
202
     isSamePrecendence exp1@(Op_neg _) exp2 =
203
       case exp2 of
204
         (Op_neg _ ) -> <u>True</u>
205
                        -> False
206
     isSamePrecendence exp1@(Op_mul _ _) exp2 =
207
       case exp2 of
208
         (Op_mul _ _) -> <u>True</u>
209
         (Op_div _ _) -> <u>True</u>
210
                        -> False
211
     isSamePrecendence exp1@(Op_div _ _) exp2 =
       case exp2 of
213
         (Op_mul _ _) -> <u>True</u>
         (Op_div _ _) -> <u>True</u>
215
                        -> False
216
     isSamePrecendence exp10(Op_add _ _) exp2 =
217
       case exp2 of
218
         (Op_add _ _) -> <u>True</u>
219
```

```
(Op_sub _ _) -> <u>True</u>
220
                     -> False
221
    isSamePrecendence exp1@(Op_sub _ _) exp2 =
222
      case exp2 of
223
        (Op_add _ _) -> <u>True</u>
224
        (Op_sub _ _) -> <u>True</u>
225
                      -> False
226
    isSamePrecendence exp1@(Op_eq _ _) exp2
                                                    = False -- relational
227
    isSamePrecendence exp1@(Op_neq _ _) exp2
                                                    = False -- relational
    isSamePrecendence exp1@(Op_less _ _) exp2
                                                    = False -- relational
229
    isSamePrecendence exp10(\overline{Op\_less\_eq}\_\_) exp2 = \underline{False} -- relational
    isSamePrecendence exp1@(Op_large _ _) exp2
                                                    = False -- relational
231
    isSamePrecendence exp1@(0\overline{p\_large\_eq\_}) exp2 = \underline{False} -- relational
    isSamePrecendence exp1@(Op_not _) exp2 =
233
      case exp2 of
        (Op_not _)
                    -> <u>True</u>
235
                       -> False
    isSamePrecendence exp10(Op_and _ _) exp2 =
237
      case exp2 of
238
        (Op_and _ _) -> <u>True</u>
239
                       -> False
240
    isSamePrecendence exp1@(Op_or _ _) exp2 =
241
      case exp2 of
242
        (Op_or _ _) -> <u>True</u>
243
                       -> <u>False</u>
244
    isSamePrecendence _ _ = False
245
246
    -- exp1 has smaller precendence than exp2 if it is not higher nor same
247
    isSamllerPrecendence :: Exp -> Exp -> Bool
248
    isSamllerPrecendence exp1 exp2 = (not (isHigherPrecendence exp1 exp2)) &&
249
                                       (not (isSamePrecendence
                                                                  exp1 exp2))
250
251
    -- does expression has operator in it?
252
    hasOperatorExp :: Exp -> Bool
    hasOperatorExp (Lval_{-}) = False
254
    hasOperatorExp (BoolConst _) = False
    hasOperatorExp (IntConst _) = False
256
    hasOperatorExp (StrConst _) = False
257
    hasOperatorExp _ = True
258
259
    -- return true if parent is sub/div and child has same precendence as parent
260
          pexp: parent expression
261
          cexp: child expression
262
    isDivSubParentSamePrecChild :: Exp -> Exp -> Bool
263
    isDivSubParentSamePrecChild pexp@(Op_div _ _) cexp@(Op_div _ _) = True -- / with a right child of / need
264
    265
    isDivSubParentSamePrecChild pexp@(Op_sub _ _) cexp@(Op_sub _ _) = <u>True</u> -- - (sub) with a right child of -
    isDivSubParentSamePrecChild pexp@(Op_sub _ _) cexp@(Op_add _ _) = True -- (sub) with a right child of +
267
    isDivSubParentSamePrecChild _ _ = False
269
    -- True if Integer division happens
          Integer division: 3 * (5 / 3) = 3 * 1 = 3, but (3 * 5) / 3 = 15 / 3 = 5, so parens needed
271
    isIntegerDision :: Exp -> Exp -> Bool
    isIntegerDision pexp@(Op_mul _ _) cexp@(Op_div _ _) = \underline{\text{True}}
273
    isIntegerDision _ _ = False
275
```

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

```
| (hasOperatorExp exp) && (not (isSamePrecendence pexp exp)) = "-" ++ surroundByParens (strExp exp) -- n
332
                           = "-" ++ (strExp exp) -- no need to parens constants/Lval
      | otherwise
333
    -- <exp> <binop: "*"> <exp>
34
    -- Single space should surround 12 binary operators.
35
    strExp pexp@(Op_mul exp1 exp2) = (strBinaryExpLChild pexp exp1) ++ " * " ++ (strBinaryExpRChild pexp exp2)
336
    -- <exp> <binop: "/"> <exp>
337
    -- Single space should surround 12 binary operators.
338
    strExp pexp@(Op_div exp1 exp2) = (strBinaryExpLChild pexp exp1) ++ " / " ++ (strBinaryExpRChild pexp exp2)
339
    -- <exp> <binop: "+"> <exp>
340
    -- Single space should surround 12 binary operators.
341
    strExp pexp@(Op_add exp1 exp2) = (strBinaryExpLChild pexp exp1) ++ " + " ++ (strBinaryExpRChild pexp exp2)
    -- <exp> <binop: "-"> <exp>
343
    -- Single space should surround 12 binary operators.
    strExp pexp@(Op_sub exp1 exp2) = (strBinaryExpLChild pexp exp1) ++ " - " ++ (strBinaryExpRChild pexp exp2)
345
    -- <exp> <binop: "="> <exp>
    -- Single space should surround 12 binary operators.
347
    strExp pexp@(Op_eq exp1 exp2) = (strBinaryExpLChild pexp exp1) ++ " = " ++ (strBinaryExpRChild pexp exp2)
    -- <exp> <binop: "!="> <exp>
349
    -- Single space should surround 12 binary operators.
350
    strExp pexp@(Op_neq exp1 exp2) = (strBinaryExpLChild pexp exp1) ++ " != " ++ (strBinaryExpRChild pexp exp2
351
    -- <exp> <binop: "<"> <exp>
352
    -- Single space should surround 12 binary operators.
353
    strExp pexp@(Op_less exp1 exp2) = (strBinaryExpLChild pexp exp1) ++ " < " ++ (strBinaryExpRChild pexp exp2
354
    -- <exp> <binop: "<="> <exp>
355
    -- Single space should surround 12 binary operators.
356
    strExp pexp@(Op_less_eq exp1 exp2) = (strBinaryExpLChild pexp exp1) ++ " <= " ++ (strBinaryExpRChild pexp
    -- <exp> <binop: ">"> <exp>
358
    -- Single space should surround 12 binary operators.
    strExp pexp@(Op large exp1 exp2) = (strBinaryExpLChild pexp exp1) ++ " > " ++ (strBinaryExpRChild pexp exp
360
    -- <exp> <binop: ">="> <exp>
361
    -- Single space should surround 12 binary operators.
362
    strExp pexp@(Op_large_eq exp1 exp2) = (strBinaryExpLChild pexp exp1) ++ " >= " ++ (strBinaryExpRChild pexp
    -- <unop: not> <exp>
364
    -- There should be a single space after not.
    strExp pexp@(Op not exp)
366
      -- need to parens expression with operator and (child's precendence is smaller)
367
      | (hasOperatorExp exp) && (isSamllerPrecendence exp pexp) = "not " ++ surroundByParens (strExp exp)
368
                           = "not " ++ (strExp exp) -- no need to parens constants/Lval
      | otherwise
369
    -- <exp> <binop: and> <exp>
370
    -- Single space should surround 12 binary operators.
371
    strExp pexp@(Op_and exp1 exp2) = (strBinaryExpLChild pexp exp1) ++ " and " ++ (strBinaryExpRChild pexp exp
372
    -- <exp> <binop: or> <exp>
373
    -- Single space should surround 12 binary operators.
374
    strExp pexp@(Op_or exp1 exp2) = (strBinaryExpLChild pexp exp1) ++ " or " ++ (strBinaryExpRChild pexp exp2)
375
376
377
    -- Int: indentation level
378
    strStmt :: Int -> Stmt -> String
379
    -- In a procedure body, each statement should start on a new line. So ++ newline in each's end
    strStmt indentLevel (Assign lValue exp) =
381
      -- <lualue> <- <exp>;
      -- Single spaces should surround the assignment operator <-
383
      (addIndentation indentLevel) ++ (strLValue lValue) ++ " <- " ++ (strExp exp) ++ semicolon ++ newline
384
    strStmt indentLevel (Read lValue) =
385
      -- read <lvalue>;
386
      (addIndentation indentLevel) ++ "read " ++ (strLValue 1Value) ++ semicolon ++ newline
387
```

```
strStmt indentLevel (Write exp) =
388
      -- write <exp>;
389
      (addIndentation indentLevel) ++ "write " ++ (strExp exp) ++ semicolon ++ newline
390
    strStmt indentLevel (Writeln exp) =
391
      -- writeln <exp>;
392
      (addIndentation indentLevel) ++ "writeln " ++ (strExp exp) ++ semicolon ++ newline
393
    strStmt indentLevel (Call ident exps) =
      -- call <id>(<exp-list>);
395
             where <exp-list> is a (possibly empty according to parser) comma-separated list of expressions.
396
      (addIndentation indentLevel) ++ "call " ++ ident ++ surroundByParens (intercalate comma (map strExp exps
397
    -- thenStmts is non-empty according to parser, elseStmts is possible empty according to parser
    strStmt indentLevel (<u>If</u> exp thenStmts elseStmts) =
399
      -- IF elseStmts is empty: according to parser: if <exp> then <stmt-list> fi
400
      if null elseStmts then
401
        -- "if ... then" should be printed on one line, irrespective of the size of the intervening expression
        (addIndentation indentLevel) ++ "if " ++ (strExp exp) ++ " then" ++ newline ++
403
        -- more indentation
        (concatMap (strStmt (indentLevel+1)) thenStmts) ++
405
        -- the terminating fi should be indented exactly as the corresponding if.
406
        (addIndentation indentLevel) ++ "fi" ++ newline
407
      -- OTHERWISE
                               : according to parser: if <expr> then <stmt-list> else <stmt-list> fi
408
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
        (concatMap (strStmt (indentLevel+1)) elseStmts) ++
416
        -- the terminating fi and else should be indented exactly as the corresponding if.
        (addIndentation indentLevel) ++ "fi" ++ newline
418
    -- stmts is non-empty according to parser
    strStmt indentLevel (While exp stmts) =
420
      -- In a while statement, "while ... do" should be printed on one line,
            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
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
437
    -- non-empty input list fieldDecls@(x:xs) according to parser
    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
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),
                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
    strRecord (Record fieldDecls recordName) =
459
      "record" ++ newline ++
      (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,
    -- followed by a positive integer in square brackets all without intervening
470
            white space.
    -- That string, the type, and the type alias, should be separated by single
472
          spaces, and the whole line terminated by a semicolon.
    strArray :: Array -> String
474
    strArray (Array arraySize arrayType arrayName) =
      "array" ++ surroundByBrackets (show arraySize) ++ " " ++ (strDataType arrayType) ++ " " ++ arrayName ++
476
478
    _____
    -- Procedure toString functions
480
    strParameter :: Parameter -> String
482
    strParameter (DataParameter dataType paraName) = (strDataType dataType) ++ " " ++ paraName
483
    strParameter (BooleanVal paraName) = "boolean val " ++ paraName
    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.
    strProcedureHeader :: ProcedureHeader -> String
489
    strProcedureHeader (ProcedureHeader procedureName parameters) =
      procedureName ++ " " ++ (surroundByParens (intercalate comma (map strParameter parameters)))
491
    -- A variable declaration consists of
493
         a) a type name (boolean, integer, or a type alias),
         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.
    strVariableDecl :: VariableDecl -> String
499
```

```
strVariableDecl (VariableDecl dataType varNames) =
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
    strProcedureBody (ProcedureBody variableDecls stmts) =
509
      (concatMap strVariableDecl variableDecls) ++
      -- 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 ++
      -- Within each procedure, declarations and top-level statements should be indented.
515
      concatMap (strStmt 1) stmts ++
      "}" ++ 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 ++
      (strProcedureBody pb)
526
528
529
    -- Program toString function
530
    strProgram :: Program -> String
532
    -- If there are no record and array type definitions, the first procedure should
          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)
545
```

\_\_\_\_\_\_

\_\_\_\_\_\_

## Roo.hs

= do

```
-- COMP90045 Programming Language Implementation Project --
                             Roo Compiler
       Implemented by Xulin Yang
   module Main (main)
   where
   import RooParser (ast)
   import PrettyRoo (pp)
    import System.Environment (getProgName, getArgs)
10
    import System.Exit (exitWith, ExitCode(...))
11
12
   data Task
13
      = Parse | Pprint
14
        deriving (Eq, Show)
15
16
   main :: <u>IO</u> ()
17
   main
      = do
19
          progname <- getProgName</pre>
          args <- getArgs
21
          task <- checkArgs progname args</pre>
22
          case task of
23
            Parse
               -> do
25
                    let [_, filename] = args
26
                    input <- readFile filename</pre>
27
                    let output = ast input
28
                    case output of
29
                       Right tree
30
                         -> putStrLn (show tree)
31
                       Left err
32
                         -> do putStrLn "Parse error at "
33
                               print err
34
                               exitWith (ExitFailure 2)
35
            Pprint
36
               -> do
                    let [_, filename] = args
38
                    input <- readFile filename</pre>
39
                    let output = ast input
40
                    case output of
41
                       Right tree
42
                         -> putStr (pp tree)
43
                      Left err
44
                         -> do putStrLn "Parse error at "
45
                               print err
46
                               exitWith (ExitFailure 2)
47
48
    checkArgs :: String -> [String] -> 10 Task
49
    checkArgs [ ['-':_]
50
```

```
putStrLn ("Missing filename")
52
          exitWith (ExitFailure 1)
53
    checkArgs _ ["-a", filename]
54
     = return Parse
55
   checkArgs _ ["-p", filename]
56
     = return Pprint
57
   checkArgs progname _
59
          putStrLn ("Usage: " ++ progname ++ " [-p] filename")
60
          exitWith (ExitFailure 1)
61
```

-----

## RooAST.hs

```
______
   -- COMP90045 Programming Language Implementation Project --
                          Roo Compiler
   -- Implemented by Xulin Yang
   -- read from the bottom to top
   module RooAST where
   _____
   -- Terminology:
10
   -- O+: zero or more/possible empty
11
   -- 1+: one or more/ non empty
12
          both 0+, 1+ are stored in list [] but 1+ will be implemented in parser not here
13
15
16
   -- Specification of an AST for Roo
17
19
   -- Identifier: String
   type <a href="Ident">Ident</a> = String
21
22
   -- Base type: boolean, integer type indicator
23
          Not necessary to have string as no variable/parameter/declaration has string type
   data BaseType
25
     = BooleanType
     | IntegerType
27
       deriving (Show, Eq)
28
   -- A boolean literal is false or true.
30
   type BooleanLiteral = Bool
31
   -- An integer literal is a sequence of digits, only stores natural number in our parser implementation
32
   type IntegerLiteral = Int
33
   -- A string literal is a sequence of characters between double quotes.
34
        The sequence itself cannot contain double quotes or newline/tab characters.
35
        It may, however, contain '"', '\n', and '\t', respectively, to represent
36
        those characters.
   type StringLiteral = String
38
39
   -- User custermized record type, stored as string
40
   type AliasType = String
42
   -- for Array, VariableDecl: they have either boolean, integer, or a type alias data type
43
          factored out for reuse purpose
44
   data DataType
45
     = BasyDataType BaseType
46
     | AliasDataType AliasType
47
       deriving (Show, Eq)
48
49
   -- An lvalue (<lvalue>) has four (and only four) possible forms:
50
          An example lvalue is point[0].xCoord
51
```

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

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

```
-- Each procedure consists of (in the given order):
164
         1. the keyword procedure,
165
         2. a procedure header, and
166
         3. a procedure body.
167
    data Procedure
168
      = Procedure ProcedureHeader ProcedureBody
169
         deriving (Show, Eq)
170
171
    -- array type definition consists of (in the given order):
172
         1. the keyword array,
173
         2. a (positive) integer literal enclosed in square brackets,
         3. a type name which is either an identifier (a type alias) or one of
175
             boolean and integer,
176
         4. an identifier (giving a name to the array type), and
177
         5. a semicolon.
    data Array
179
      = Array IntegerLiteral DataType <u>Ident</u>
180
         deriving (Show, Eq)
181
182
    -- field declaration is of:
183
         1. boolean or integer
184
         2. followed by an identifier (the field name).
185
    data FieldDecl
186
      = FieldDecl BaseType Ident
187
         deriving (Show, Eq)
188
    -- record consists of:
190
          1. the keyword record,
         2. a 1+ list of field declarations, separated by semicolons,
192
               the whole list enclosed in braces,
193
         3. an identifier, and
194
         4. a semicolon.
    data Record
196
      = Record [FieldDecl] Ident
197
         deriving (Show, Eq)
198
199
    -- A Roo program consists of
200
          1. O+ record type definitions, followed by
201
         2. O+ array type definitions, followed by
202
          3. 1+ procedure definitions.
203
204
    data Program
      = Program [Record] [Array] [Procedure]
205
         deriving (Show, Eq)
206
```

\_\_\_\_\_\_

\_\_\_\_\_\_

## RooParser.hs

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

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

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

```
164
     pStringLiteral :: Parser StringLiteral
165
     pStringLiteral
166
167
          do
168
             s <- pString
169
            return (s)
170
171
             "string literal"
172
173
175
     -- An lvalue (<lvalue>) has four (and only four) possible forms:
176
              \langle id \rangle
177
              \langle id \rangle. \langle id \rangle
              \langle id \rangle [\langle exp \rangle]
179
              \langle id \rangle [\langle exp \rangle] . \langle id \rangle
180
     -- An example lvalue is point[0].xCoord
181
182
     pLValue :: Parser LValue
183
     pLValue
184
       = try (
185
            do
186
               ident1 <- identifier
187
               exp <- brackets pExp
188
               dot
189
               ident2 <- identifier
190
               return (LBracketsDot ident1 exp ident2)
191
         )
          <|>
          try (
194
             do
195
               ident <- identifier
196
               exp <- brackets pExp</pre>
197
               return (LBrackets ident exp)
198
199
          )
          <|>
200
          try (
201
            do
202
               ident1 <- identifier
203
204
               ident2 <- identifier
205
               return (LDot ident1 ident2)
206
          )
207
          <|>
208
          do
209
             ident <- identifier
210
            return (LId ident)
211
          <?>
             "LValue"
213
215
          pExp is the main parser for expression.
216
217
          It is built using Parces's powerful
218
          buildExpressionParser and takes into account the operator
219
```

```
-- precedences and associativity specified in 'opTable' below.
220
221
    prefix name fun
222
       = Prefix (do { reservedOp name
223
                     ; return fun
224
                     }
225
                )
226
227
    binary name op
228
       = Infix (do { reservedOp name
229
                    ; return op
230
231
               ) AssocLeft
232
233
    relation name rel
       = <u>Infix</u> (do { reservedOp name
235
                    ; return rel
236
                    }
237
               ) AssocNone
238
239
     -- expression operators:
240
            All the operators on the same line have the same precedence,
241
                and the ones on later lines have lower precedence;
242
            The six relational operators are non-associative
243
                so, for example, a = b = c is not a well-formed expression).
244
            The six remaining binary operators are left-associative.
245
                        /unary
246
                        /binary and infix /left-associative
    -- * /
247
                        |binary and infix |left-associative
248
     -- = != < <= > >= |binary and infix |relational, non-associative
249
                        /unary
     -- not
250
                        /binary and infix /left-associative
    -- a.n.d.
251
    -- or
                        |binary and infix |left-associative
252
    opTable
253
      = [ [ prefix
                             Op_neg
254
                       "*"
                                                     "/"
                                                          Op_div ]
255
         , [ binary
                             Op_mul
                                         , binary
                       "+"
                              Op_add
                                                     "-"
                                                          Op_sub
         , [ binary
                                         , binary
256
         , [ relation "="
                                         , relation "!=" Op_neq , relation "<" Op_less
                              Op_eq
257
           , relation "<="
                             Op_less_eq, relation ">" Op_large, relation ">=" Op_large_eq ]
258
                       "not" Op_not
         , [ prefix
259
         , [ binary
                       "and" Op_and
                                         ]
260
                       "or"
           [ binary
                             Op_or
                                         ]
261
262
263
    pExp :: Parser Exp
264
265
      = buildExpressionParser opTable pFac
266
267
           "expression"
268
269
    pFac :: Parser Exp
271
      = choice [parens pExp, -- ( <exp>)
272
                 pLval,
273
                 pBoolConst,
274
                 pIntConst,
275
```

```
pStrConst,
276
                                 -- used to parse expression like -----1 (arbitrary unary minus before expression
                  pNeg
277
278
         <?>
           "simple expression"
280
281
    pLval, pBoolConst, pIntConst, pStrConst :: Parser Exp
    pLval
283
284
         do
285
           lval <- pLValue</pre>
           return (Lval lval)
287
           "lval expression"
289
    pBoolConst
291
         do
293
           b <- pBooleanLiteral
294
           return (BoolConst b)
295
         <?>
296
           "bool const/literal expression"
297
298
    pIntConst
299
300
         do
301
           i <- pIntegerLiteral
302
           return (<u>IntConst</u> i)
304
           "int const/literal expression"
305
306
    pStrConst
307
308
         do
309
           s <- pStringLiteral
310
           return (StrConst s)
311
312
           "string const/literal expression"
313
314
     -- parse arbitrary unary minus in pFac
315
    pNeg :: Parser Exp
316
    pNeg
317
318
319
         reservedOp "-"
320
         exp <- pExp
321
         return (Op_neg exp)
322
323
         "unary minus"
325
327
        pStmt is the main parser for statements.
328
        Statement related parsers
329
330
    pStmt, pStmtAtom, pStmtComp :: Parser Stmt
331
```

```
-- atom statement:
332
            <lvalue> <- <exp> ;
333
            read <lvalue> ;
334
            write <exp>;
335
            writeln <exp>;
336
            call < id > ( < exp-list > ) ;
337
                 where \langle exp-list \rangle is a (possibly empty) comma-separated list of expressions.
338
     -- composite statement:
339
            if <expr> then <stmt-list> else <stmt-list> fi
340
            if <exp> then <stmt-list> fi # just make second [Stmt] empty
341
            while <expr> do <stmt-list> od
342
                 where <stmt-list> is a non-empty sequence of statements, atomic or composite
343
    pStmt = choice [pStmtAtom, pStmtComp]
344
345
    pStmtAtom
347
         do
348
           r <- choice [pAsg, pRead, pWrite, pWriteln, pCall]
349
           -- all atomic stmt's semicolon is comsumed here
350
           semi
351
           return r
352
         <?>
353
           "atomic statement"
354
355
    pAsg, pRead, pWrite, pWriteln, pCall :: Parser Stmt
356
357
     -- parse: <lvalue> <- <exp> ;
358
    pAsg
359
360
         do
361
           lvalue <- pLValue
362
           reservedOp "<-"
           rvalue <- pExp
364
           return (Assign lvalue rvalue)
366
367
           "assign"
368
     -- parse: read <lvalue> ;
369
    pRead
370
      = do
371
           reserved "read"
372
           lvalue <- pLValue
373
           return (Read lvalue)
374
         <?>
375
           "read"
376
377
    -- parse: write <exp> ;
    pWrite
379
       = do
380
           reserved "write"
381
           expr <- pExp
           return (Write expr)
383
         <?>
384
           "write"
385
386
    -- parse: writeln <exp>;
387
```

```
pWriteln
388
       = do
389
           reserved "writeln"
390
           expr <- pExp
391
           return (Writeln expr)
392
         <?>
393
           "writeln"
394
395
     -- parse: call <id>(<exp-list>);
396
    pCall
397
       = do
           reserved "call"
399
           ident <- identifier
400
           exprs <- parens (pExp `sepBy` comma) -- O+ comma-separated list of expressions
401
           return (Call ident exprs)
403
           "call"
405
    pStmtComp = (choice [pIf, pWhile]) <?> "composite statement"
406
407
    pIf, pWhile :: Parser Stmt
408
409
     -- parse:
410
           if < exp > then < stmt-list > else < stmt-list > fi
411
           if <exp> then <stmt-list> fi # just make above second [Stmt] empty
412
    pIf
413
      = do
414
415
           reserved "if"
416
           exp <- pExp
417
           reserved "then"
418
           stmts <- many1 pStmt
419
           -- check if there is an else statment
420
           -- if not, return empty
           estmts <- (
422
423
             do
               reserved "fi"
42
               return []
             <|>
426
             do
427
               reserved "else"
428
                -- else body can not be empty
429
               s <- many1 pStmt
430
               reserved "fi"
431
               return s
432
             )
433
           return (<u>If</u> exp stmts estmts)
434
         <?>
435
           "if"
436
437
     -- parse: while <exp> do <stmt-list> od
438
    pWhile
439
       = do
440
           reserved "while"
441
           exp <- pExp
442
           reserved "do"
443
```

```
stmts <- many1 pStmt -- a 1+ sequence of statements, atomic or composite
444
           reserved "od"
445
           return (While exp stmts)
446
         <?>
447
           "while"
448
449
450
        Procedure related parser
451
452
     -- Each formal parameter has two components (in the given order):
453
         1. a parameter type/mode indicator, which is one of these five:
           a) a type alias,
455
           b) boolean,
456
           c) integer,
457
           d) boolean val
           e) integer val
459
    -- 2. an identifier
    pParameter :: Parser Parameter
461
    pParameter
462
463
         try(
464
           do
465
             -- parse boolean val variable
466
             reserved "boolean"
467
             reserved "val"
468
             name <- identifier
469
             return (BooleanVal name)
470
         )
         <|>
472
         try(
474
             -- parse integer val variable
             reserved "integer"
476
             reserved "val"
             name <- identifier
478
             return (IntegerVal name)
         )
480
         <|>
481
         do
482
           -- parse boolean/integer/type_alias variable
483
           paraType <- pDataType</pre>
           name <- identifier
485
           return (DataParameter paraType name)
486
487
           "parameter"
489
     -- The header has two components (in this order):
490
          1. an identifier (the procedure's name), and
491
          2. a comma-separated list of O+ formal parameters within a pair
                of parentheses (so the parentheses are always present).
493
    pProcedureHeader :: Parser ProcedureHeader
    pProcedureHeader
495
496
         do
497
           procedureName <- identifier</pre>
498
           parameters <- parens (pParameter `sepBy` comma)
499
```

```
return (ProcedureHeader procedureName parameters)
500
         <?>
501
           "procedure header"
502
503
     -- A variable declaration consists of
504
          a) a type name (boolean, integer, or a type alias),
505
          b) followed by a 1+ comma-separated list of
506
                identifiers,
507
            i) the list terminated with a semicolon.
508
            ii) There may be any number of variable declarations, in any order.
509
    pVariable :: Parser VariableDecl
    pVariable
511
512
         do
513
           varType <- pDataType</pre>
           varNames <- (identifier `sepBy1` comma)</pre>
515
           return (VariableDecl varType varNames)
517
         <?>
518
           "variable"
519
520
521
     -- procedure body consists of 0+ local variable declarations,
522
          1. A variable declaration consists of
523
            a) a type name (boolean, integer, or a type alias),
524
            b) followed by a 1+ comma-separated list of identifiers,
              i) the list terminated with a semicolon.
526
              ii) There may be any number of variable declarations, in any order.
527
          2. followed by a 1+ sequence of statements,
528
    pProcedureBody :: Parser ProcedureBody
    pProcedureBody
530
531
         do
532
           vars <- many pVariable</pre>
           stmts <- braces (many1 pStmt)
534
           return (ProcedureBody vars stmts)
535
         <?>
536
           "procedure body"
537
538
     -- Each procedure consists of (in the given order):
539
          1. the keyword procedure,
540
          2. a procedure header, and
541
          3. a procedure body.
542
    pProcedure :: Parser Procedure
543
    pProcedure
544
545
546
         do
           reserved "procedure"
547
           procedureHeader <- pProcedureHeader
           procedureBody <- pProcedureBody</pre>
549
           return (<a href="Procedure">Procedure</a> Procedure</a> Body)
         <?>
551
           "procedure"
552
553
554
    -- Array related parser
555
```

```
556
     -- array type definition consists of (in the given order):
557
          1. the keyword array,
558
          2. a (positive) integer literal enclosed in square brackets,
559
          3. a type name which is either an identifier (a type alias) or one of
560
             boolean and integer,
561
          4. an identifier (giving a name to the array type), and
562
          5. a semicolon.
563
    pArray :: Parser Array
    pArray
565
         do
567
           reserved "array"
568
           pos <- getPosition
569
           arraySize <- brackets pIntegerLiteral</pre>
           -- need to check arraySize > 0 (positive integer)
571
           if arraySize == 0
           then
573
             error ("array size sould not be 0 at line: " ++ (show (sourceLine pos)) ++ ", column: " ++ (show (
574
           else do
575
             arrayType <- pDataType</pre>
576
             arrayName <- identifier</pre>
577
             semi
578
             return (Array arraySize arrayType arrayName)
579
           <?>
580
             "array"
582
     -- Record related parser
584
     -- field declaration is of:
586
         1. boolean or integer
          2. followed by an identifier (the field name).
588
    pFieldDecl :: Parser FieldDecl
    pFieldDecl
590
591
         do
592
           fieldType <- pBaseType</pre>
593
           fieldName <- identifier</pre>
594
           return (FieldDecl fieldType fieldName)
595
596
           "field declaration"
597
598
     -- record consists of:
599
          1. the keyword record,
600
          2. a 1+ list of field declarations, separated by semicolons,
601
               the whole list enclosed in braces,
602
          3. an identifier, and
603
          4. a semicolon.
    pRecord :: Parser Record
605
    pRecord
606
607
         do
608
           reserved "record"
609
           recordFieldDecls <- braces (pFieldDecl `sepBy1` semi)</pre>
610
           recordName <- identifier
611
```

```
612
           return (Record recordFieldDecls recordName)
613
         <?>
614
           "record"
615
616
617
     -- Program related parser
618
     ______
619
     -- A Roo program consists of
620
     -- 1. O+ record type definitions, followed by
621
          2. O+ array type definitions, followed by
          3. 1+ procedure definitions.
623
    pProgram :: Parser Program
    pProgram
625
      = do
           records <- many pRecord
627
           arraies <- many pArray
           procedures <- many1 pProcedure</pre>
629
           return (Program records arraies procedures)
630
631
           "program"
632
633
634
635
     -- main (given skeleton code)
636
637
638
    joeyParse :: Parser Program
639
    joeyParse
640
      = do
641
           whiteSpace
642
          p <- pProgram
          eof
644
          return p
646
647
    ast :: String -> Either ParseError Program
    ast input
648
      = runParser joeyParse 0 "" input
649
650
    pMain :: Parser Program
651
652
    pMain
      = do
653
           whiteSpace
654
           p <- pProgram
655
           eof
656
          return p
657
    main :: IO ()
659
    main
660
       = do { progname <- getProgName
661
            ; args <- getArgs
            ; checkArgs progname args
663
            ; input <- readFile (head args)</pre>
664
            ; let output = runParser pMain 0 "" input
665
            ; case output of
666
                Right ast -> print ast
667
```

```
\underline{\text{Left}} err -> do { putStr "Parse error at "
668
                                      ; print err
669
                                      }
670
             }
671
672
     checkArgs :: String \rightarrow [String] \rightarrow <u>IO</u> ()
673
     checkArgs _ [filename]
        = return ()
675
     checkArgs progname _
676
        = do { putStrLn ("Usage: " ++ progname ++ " filename\n\n")
             ; exitWith (ExitFailure 1)
677
679
```