

PrettyRoo.hs

```
=====
PrettyRoo.hs
=====
1  -----
2  -- COMP90045 Programming Language Implementation Project --
3  --                      Roo Compiler                      --
4  -- Implemented by Xulin Yang                               --
5  -- read from the bottom to top                             --
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 (BasyDataType b) = strBaseType b
51
```

```

52 strBooleanLiteral :: BooleanLiteral -> String
53 strBooleanLiteral True = "true"
54 strBooleanLiteral False = "false"
55
56 -- StringLiteral can be directly used as it is String type
57
58 -- IntegerLiteral can be turned to string by show as it is Int type
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 precedence than exp2, higher if exp2 has no operator
77 isHigherPrecedence :: Exp -> Exp -> Bool
78 isHigherPrecedence exp1@(Op_neg _) exp2 =
79     case exp2 of
80         (Op_neg _ ) -> False
81         _            -> True
82 isHigherPrecedence exp1@(Op_mul _ _) exp2 =
83     case exp2 of
84         (Op_neg _ ) -> False
85         (Op_mul _ _) -> False
86         (Op_div _ _) -> False
87         _            -> True
88 isHigherPrecedence exp1@(Op_div _ _) exp2 =
89     case exp2 of
90         (Op_neg _ ) -> False
91         (Op_mul _ _) -> False
92         (Op_div _ _) -> False
93         _            -> True
94 isHigherPrecedence 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 isHigherPrecedence exp1@(Op_sub _ _) exp2 =
103     case exp2 of
104         (Op_neg _ ) -> False
105         (Op_mul _ _) -> False
106         (Op_div _ _) -> False
107         (Op_add _ _) -> False

```

```

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

```

```

164         _ -> False
165 isHigherPrecedence 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 precedence
171         (Lval _) -> True
172         (BoolConst _) -> True
173         (IntConst _) -> True
174         (StrConst _) -> True
175         _ -> False
176 isHigherPrecedence exp1@(Op_not _) exp2 =
177     case exp2 of
178         (Op_and _ _) -> True
179         (Op_or _ _) -> True
180         -- constants has lowest precedence
181         (Lval _) -> True
182         (BoolConst _) -> True
183         (IntConst _) -> True
184         (StrConst _) -> True
185         _ -> False
186 isHigherPrecedence exp1@(Op_and _ _) exp2 =
187     case exp2 of
188         (Op_or _ _) -> True
189         -- constants has lowest precedence
190         (Lval _) -> True
191         (BoolConst _) -> True
192         (IntConst _) -> True
193         (StrConst _) -> True
194         _ -> False
195 isHigherPrecedence exp1@(Op_or _ _) exp2
196     -- constants has lowest precedence
197     | (not (hasOperatorExp exp2)) = True
198     | otherwise = False
199 isHigherPrecedence _ _ = False
200
201 -- exp1 has same precedence as exp2
202 isSamePrecedence :: Exp -> Exp -> Bool
203 isSamePrecedence exp1@(Op_neg _) exp2 =
204     case exp2 of
205         (Op_neg _ _) -> True
206         _ -> False
207 isSamePrecedence exp1@(Op_mul _ _) exp2 =
208     case exp2 of
209         (Op_mul _ _) -> True
210         (Op_div _ _) -> True
211         _ -> False
212 isSamePrecedence exp1@(Op_div _ _) exp2 =
213     case exp2 of
214         (Op_mul _ _) -> True
215         (Op_div _ _) -> True
216         _ -> False
217 isSamePrecedence 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 precedence suggests a parens
284   | (isSmallerPrecedence 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 precedence (except "not" operator) or
301   -- same precedence 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      (isSmallerPrecedence exp2 pexp) ||
307      (isIntegerDivision 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) = strBooleanLiteral 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 (isSamePrecedence 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 exp2)
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 exp2)
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 precedence is smaller)
368 | (hasOperatorExp exp) && (isSmallerPrecedence 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 exp2)
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 (If exp thenStmts elseStmts) =
400   -- IF elseStmts is empty: according to parser: if <exp> then <stmt-list> fi
401   if null elseStmts then
402     -- "if ... then" should be printed on one line, irrespective of the size of the intervening expression
403     (addIndentation indentLevel) ++ "if " ++ (strExp exp) ++ " then" ++ newline ++
404     -- more indentation
405     (concatMap (strStmt (indentLevel+1)) thenStmts) ++
406     -- the terminating fi should be indented exactly as the corresponding if.
407     (addIndentation indentLevel) ++ "fi" ++ newline
408   -- OTHERWISE           : according to parser: if <expr> then <stmt-list> else <stmt-list> fi
409   else
410     -- "if ... then" should be printed on one line, irrespective of the size of the intervening expression
411     (addIndentation indentLevel) ++ "if " ++ (strExp exp) ++ " then" ++ newline ++
412     -- more indentation
413     (concatMap (strStmt (indentLevel+1)) thenStmts) ++
414     (addIndentation indentLevel) ++ "else" ++ newline ++
415     -- more indentation
416     (concatMap (strStmt (indentLevel+1)) elseStmts) ++
417     -- the terminating fi and else should be indented exactly as the corresponding if.
418     (addIndentation indentLevel) ++ "fi" ++ newline
419 -- stmts is non-empty according to parser
420 strStmt indentLevel (While exp stmts) =
421   -- In a while statement, "while ... do" should be printed on one line,
422   --   irrespective of the size of the intervening expression
423   (addIndentation indentLevel) ++ "while " ++ (strExp exp) ++ " do" ++ newline ++
424   -- more indentation
425   (concatMap (strStmt (indentLevel+1)) stmts) ++
426   -- The terminating od should be indented exactly as the corresponding while.
427   (addIndentation indentLevel) ++ "od" ++ newline
428
429 -----
430 -- Record toString function
431 -----
432 -- field declaration is of:
433 --   1. boolean or integer
434 --   2. followed by an identifier (the field name).
435 strFieldDecl :: FieldDecl -> String
436 strFieldDecl (FieldDecl baseType fieldName) = (strBaseType baseType) ++ " " ++ fieldName
437
438 -- non-empty input list fieldDecls@(x:xs) according to parser
439 strFieldDecls :: [FieldDecl] -> String
440 -- first field decl starts with {
441 strFieldDecls (x:xs) = (addIndentation 1) ++ "{ " ++ (strFieldDecl x) ++ newline ++
442 -- rest start with ;
443   (concatMap

```



```

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

```

```

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

```



Roo.hs

```
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 System.Environment (getProgName, getArgs)
11 import System.Exit (exitWith, ExitCode(..))
12
13 data Task
14   = Parse | Pprint
15   deriving (Eq, Show)
16
17 main :: IO ()
18 main
19   = do
20     progname <- getProgName
21     args <- getArgs
22     task <- checkArgs progname args
23     case task of
24       Parse
25         -> do
26           let [_ , filename] = args
27           input <- readFile filename
28           let output = ast input
29           case output of
30             Right tree
31               -> putStrLn (show tree)
32             Left err
33               -> do putStrLn "Parse error at "
34                   print err
35                   exitWith (ExitFailure 2)
36       Pprint
37         -> do
38           let [_ , filename] = args
39           input <- readFile filename
40           let output = ast input
41           case output of
42             Right tree
43               -> putStr (pp tree)
44             Left err
45               -> do putStrLn "Parse error at "
46                   print err
47                   exitWith (ExitFailure 2)
48
49 checkArgs :: String -> [String] -> IO Task
50 checkArgs _ ['-':_]
51   = do
```

```
52     putStrLn ("Missing filename")
53     exitWith (ExitFailure 1)
54 checkArgs _ ["-a", filename]
55   = return Parse
56 checkArgs _ ["-p", filename]
57   = return Pprint
58 checkArgs progname _
59   = do
60     putStrLn ("Usage: " ++ progname ++ " [-p] filename")
61     exitWith (ExitFailure 1)
62
```

RooAST.hs

```
1  -----
2  -- COMP90045 Programming Language Implementation Project --
3  --                      Roo Compiler                      --
4  -- Implemented by Xulin Yang                             --
5  -- read from the bottom to top                           --
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 not here
14 -----
15
16 -----
17 -- Specification of an AST for Roo
18 -----
19
20 -- Identifier: String
21 type Ident = String
22
23 -- Base type: boolean, integer type indicator
24 -- Not necessary to have string as no variable/parameter/declaration has string type
25 data BaseType
26   = BaseType
27   | IntegerType
28   deriving (Show, Eq)
29
30 -- A boolean literal is false or true.
31 type BooleanLiteral = Bool
32 -- An integer literal is a sequence of digits, only stores natural number in our parser implementation
33 type IntegerLiteral = Int
34 -- A string literal is a sequence of characters between double quotes.
35 -- The sequence itself cannot contain double quotes or newline/tab characters.
36 -- It may, however, contain " ", '\n', and '\t', respectively, to represent
37 -- those characters.
38 type StringLiteral = String
39
40 -- User customized record type, stored as string
41 type AliasType = String
42
43 -- for Array, VariableDecl: they have either boolean, integer, or a type alias data type
44 -- factored out for reuse purpose
45 data DataType
46   = BaseType BaseType
47   | AliasType AliasType
48   deriving (Show, Eq)
49
50 -- An lvalue (<lvalue>) has four (and only four) possible forms:
51 -- An example lvalue is point[0].xCoord
```

```

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

```

```

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

```

```

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

```


RooParser.hs

```
=====

RooParser.hs

=====

1  -----
2  -- COMP90045 Programming Language Implementation Project --
3  --                      Roo Compiler                      --
4  --  Implemented by Xulin Yang                          --
5  --  read from the bottom to top                        --
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 identifier is a non-empty sequence of alphanumeric characters,
29         -- underscore and apostrophe ('), and it must start with a (lower or upper case) letter.
30         , Q.identLetter      = alphaNum <|> char '_' <|> char '\''
31         , Q.opStart           = oneOf "+-*<"
32         , Q.opLetter          = oneOf "="
33         , Q.reservedNames     = joeyReserved
34         , Q.reservedOpNames   = joeyOpnames
35         })
36
37 whiteSpace    = Q.whiteSpace scanner
38 lexeme        = Q.lexeme scanner
39 natural       = Q.natural scanner
40 identifier    = Q.identifier scanner
41 semi          = Q.semi scanner
42 comma         = Q.comma scanner
43 dot           = Q.dot scanner
44 parens        = Q.parens scanner
45 braces        = Q.braces scanner
46 brackets      = Q.brackets scanner
47 squares       = Q.squares scanner
48 reserved      = Q.reserved scanner
49 reservedOp    = Q.reservedOp scanner
50 stringLiteral = Q.stringLiteral scanner
51
```

```

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

```

```

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

```

```

164
165 pStringLiteral :: Parser StringLiteral
166 pStringLiteral
167 =
168     do
169         s <- pString
170         return (s)
171     <?>
172     "string literal"
173
174
175 -----
176 -- An lvalue (<lvalue>) has four (and only four) possible forms:
177 --     <id>
178 --     <id>.<id>
179 --     <id>[<exp>]
180 --     <id>[<exp>].<id>
181 -- An example lvalue is point[0].xCoord
182 -----
183 pLValue :: Parser LValue
184 pLValue
185 = try (
186     do
187         ident1 <- identifier
188         exp <- brackets pExp
189         dot
190         ident2 <- identifier
191         return (LBracketsDot ident1 exp ident2)
192 )
193 <|>
194 try (
195     do
196         ident <- identifier
197         exp <- brackets pExp
198         return (LBrackets ident exp)
199 )
200 <|>
201 try (
202     do
203         ident1 <- identifier
204         dot
205         ident2 <- identifier
206         return (LDot ident1 ident2)
207 )
208 <|>
209 do
210     ident <- identifier
211     return (LId ident)
212 <?>
213     "LValue"
214
215 -----
216 -- pExp is the main parser for expression.
217 --
218 -- It is built using Parces's powerful
219 -- buildExpressionParser and takes into account the operator

```

```

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

```

```

276         pStrConst,
277         pNeg      -- used to parse expression like -----1 (arbitrary unary minus before expression)
278     ]
279     <?>
280     "simple expression"
281
282 pLval, pBoolConst, pIntConst, pStrConst :: Parser Exp
283 pLval
284     =
285     do
286         lval <- pLValue
287         return (Lval lval)
288     <?>
289     "lval expression"
290
291 pBoolConst
292     =
293     do
294         b <- pBooleanLiteral
295         return (BoolConst b)
296     <?>
297     "bool const/literal expression"
298
299 pIntConst
300     =
301     do
302         i <- pIntegerLiteral
303         return (IntConst i)
304     <?>
305     "int const/literal expression"
306
307 pStrConst
308     =
309     do
310         s <- pStringLiteral
311         return (StrConst s)
312     <?>
313     "string const/literal expression"
314
315 -- parse arbitrary unary minus in pFac
316 pNeg :: Parser Exp
317 pNeg
318     =
319     do
320         reservedOp "-"
321         exp <- pExp
322         return (Op_neg exp)
323     <?>
324     "unary minus"
325
326
327 -----
328 -- pStmt is the main parser for statements.
329 -- Statement related parsers
330 -----
331 pStmt, pStmtAtom, pStmtComp :: Parser Stmt

```

```

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

```

```

388 pWriteln
389   = do
390     reserved "writeln"
391     expr <- pExp
392     return (Writeln expr)
393   <?>
394     "writeln"
395
396 -- parse: call <id>(<exp-list>) ;
397 pCall
398   = do
399     reserved "call"
400     ident <- identifier
401     exprs <- parens (pExp `sepBy` comma) -- 0+ comma-separated list of expressions
402     return (Call ident exprs)
403   <?>
404     "call"
405
406 pStmtComp = (choice [pIf, pWhile]) <?> "composite statement"
407
408 pIf, pWhile :: Parser Stmt
409
410 -- parse:
411 --   if <exp> then <stmt-list> else <stmt-list> fi
412 --   if <exp> then <stmt-list> fi # just make above second [Stmt] empty
413 pIf
414   = do
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
421     estmts <- (
422       do
423         reserved "fi"
424         return []
425       <|>
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 (If exp stmts estmts)
434   <?>
435     "if"
436
437
438 -- parse: while <exp> do <stmt-list> od
439 pWhile
440   = do
441     reserved "while"
442     exp <- pExp
443     reserved "do"

```



```

444     stmts <- many1 pStmt -- a 1+ sequence of statements, atomic or composite
445     reserved "od"
446     return (While exp stmts)
447   <?>
448     "while"
449
450 -----
451 -- Procedure related parser
452 -----
453 -- Each formal parameter has two components (in the given order):
454 -- 1. a parameter type/mode indicator, which is one of these five:
455 --   a) a type alias,
456 --   b) boolean,
457 --   c) integer,
458 --   d) boolean val
459 --   e) integer val
460 -- 2. an identifier
461 pParameter :: Parser Parameter
462 pParameter
463   =
464     try(
465       do
466         -- parse boolean val variable
467         reserved "boolean"
468         reserved "val"
469         name <- identifier
470         return (BooleanVal name)
471     )
472   <|>
473     try(
474       do
475         -- parse integer val variable
476         reserved "integer"
477         reserved "val"
478         name <- identifier
479         return (IntegerVal name)
480     )
481   <|>
482     do
483       -- parse boolean/integer/type_alias variable
484       paraType <- pDataType
485       name <- identifier
486       return (DataParameter paraType name)
487   <?>
488     "parameter"
489
490 -- The header has two components (in this order):
491 -- 1. an identifier (the procedure's name), and
492 -- 2. a comma-separated list of 0+ formal parameters within a pair
493 --    of parentheses (so the parentheses are always present).
494 pProcedureHeader :: Parser ProcedureHeader
495 pProcedureHeader
496   =
497     do
498       procedureName <- identifier
499       parameters <- parens (pParameter `sepBy` comma)

```

```

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

```

```

556 -----
557 -- array type definition consists of (in the given order):
558 -- 1. the keyword array,
559 -- 2. a (positive) integer literal enclosed in square brackets,
560 -- 3. a type name which is either an identifier (a type alias) or one of
561 --    boolean and integer,
562 -- 4. an identifier (giving a name to the array type), and
563 -- 5. a semicolon.
564 pArray :: Parser Array
565 pArray
566   =
567   do
568     reserved "array"
569     pos <- getPosition
570     arraySize <- brackets pIntegerLiteral
571     -- need to check arraySize > 0 (positive integer)
572     if arraySize == 0
573     then
574       error ("array size should not be 0 at line: " ++ (show (sourceLine pos)) ++ ", column: " ++ (show (
575     else do
576       arrayType <- pDataType
577       arrayName <- identifier
578       semi
579       return (Array arraySize arrayType arrayName)
580     <?>
581     "array"
582
583 -----
584 -- Record related parser
585 -----
586 -- field declaration is of:
587 -- 1. boolean or integer
588 -- 2. followed by an identifier (the field name).
589 pFieldDecl :: Parser FieldDecl
590 pFieldDecl
591   =
592   do
593     fieldType <- pBaseType
594     fieldName <- identifier
595     return (FieldDecl fieldType fieldName)
596     <?>
597     "field declaration"
598
599 -- record consists of:
600 -- 1. the keyword record,
601 -- 2. a 1+ list of field declarations, separated by semicolons,
602 --    the whole list enclosed in braces,
603 -- 3. an identifier, and
604 -- 4. a semicolon.
605 pRecord :: Parser Record
606 pRecord
607   =
608   do
609     reserved "record"
610     recordFieldDecls <- braces (pFieldDecl `sepBy1` semi)
611     recordName <- identifier

```

```

612     semi
613     return (Record recordFieldDecls recordName)
614     <?>
615     "record"
616
617 -----
618 -- Program related parser
619 -----
620 -- A Roo program consists of
621 -- 1. 0+ record type definitions, followed by
622 -- 2. 0+ array type definitions, followed by
623 -- 3. 1+ procedure definitions.
624 pProgram :: Parser Program
625 pProgram
626   = do
627     records <- many pRecord
628     arraies <- many pArray
629     procedures <- many1 pProcedure
630     return (Program records arraies procedures)
631     <?>
632     "program"
633
634 -----
635 -- main (given skeleton code)
636 -----
637
638
639 joeyParse :: Parser Program
640 joeyParse
641   = do
642     whiteSpace
643     p <- pProgram
644     eof
645     return p
646
647 ast :: String -> Either ParseError Program
648 ast input
649   = runParser joeyParse 0 "" input
650
651 pMain :: Parser Program
652 pMain
653   = do
654     whiteSpace
655     p <- pProgram
656     eof
657     return p
658
659 main :: IO ()
660 main
661   = do { progname <- getProgName
662         ; args <- getArgs
663         ; checkArgs progname args
664         ; input <- readFile (head args)
665         ; let output = runParser pMain 0 "" input
666         ; case output of
667             Right ast -> print ast

```

```

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

```

