```
package parser;

/*************************************************************************
 Java CUP specification for a parser for C-- programs
 *************************************************************************/

import java_cup.runtime.*;
import java.util.*;
import java.io.*;
import ast.*;
import lexer.*;

/* The code below redefines method syntax_error to give better error messages
 * than just "Syntax error"
 */
parser code {:

public void syntax_error(Symbol currToken) {
   if (currToken.value == null) {
      ErrMsg.fatal(0,0, "Syntax error at end of file");
   }
   else {
      ErrMsg.fatal(((TokenVal)currToken.value).linenum,
                   ((TokenVal)currToken.value).charnum,
                   "Syntax error");
   }
   throw new SyntaxErrorException();
   // System.exit(-1);
}
:};


/* Terminals (tokens returned by the scanner) */
terminal            INT;
terminal            BOOL;
terminal            VOID;
terminal TokenVal       TRUE;
terminal TokenVal       FALSE;
terminal            STRUCT;
terminal            CIN;
terminal            COUT;
terminal            IF;
terminal            ELSE;
terminal            WHILE;
terminal            RETURN;
terminal IdTokenVal     ID;
terminal IntLitTokenVal INTLITERAL;
terminal StrLitTokenVal STRINGLITERAL;
terminal            LCURLY;
terminal            RCURLY;
terminal            LPAREN;
terminal            RPAREN;
terminal            SEMICOLON;
terminal            COMMA;
terminal            DOT;
terminal            WRITE;
terminal            READ;
terminal            PLUSPLUS;
terminal            MINUSMINUS;
terminal            PLUS;
terminal            MINUS;
terminal            TIMES;
terminal            DIVIDE;
terminal            NOT;
terminal            AND;
terminal            OR;
terminal            EQUALS;
terminal            NOTEQUALS;
terminal            LESS;
terminal            GREATER;
```

```
terminal            LESSEQ;
terminal            GREATEREQ;
terminal            ASSIGN;


/* Nonterminals
 *
 * NOTE: You will need to add more nonterminals to this list as you
 *       add productions to the grammar below.
 */
non terminal AST.ProgramNode        program;
non terminal LinkedList             declList;
non terminal AST.DeclNode           decl;
non terminal AST.TypeNode           type;
non terminal AST.IdNode             id;
non terminal LinkedList           varDeclList;
non terminal AST.VarDeclNode              varDecl;
non terminal AST.FnDeclNode           fnDecl;
non terminal AST.StructDeclNode     structDecl;
non terminal LinkedList                   structBody;
non terminal LinkedList                   formals;
non terminal LinkedList                   formalsList;
non terminal AST.FormalDeclNode         formalDecl;
non terminal AST.FnBodyNode               fnBody;
non terminal LinkedList                   stmtList;
non terminal AST.StmtNode         stmt;
non terminal AST.ExpNode              assignExp;
non terminal AST.ExpNode          exp;
non terminal AST.ExpNode              term;
non terminal AST.ExpNode              fncall;
non terminal LinkedList                   actualList;
non terminal AST.ExpNode              loc;


/* NOTE: Add precedence and associativity declarations here */
precedence right ASSIGN;
precedence left OR;
precedence left AND;
precedence nonassoc EQUALS, NOTEQUALS, LESS, GREATER, LESSEQ, GREATEREQ;
precedence left PLUS, MINUS;
precedence left TIMES, DIVIDE;
precedence right NOT;

start with program;


/* Grammar with actions
 *
 * NOTE: add more grammar rules below
 */
program         ::= declList: d
                {: RESULT = new AST.ProgramNode(new AST.DeclListNode(d));
                :}
                ;

declList        ::= declList:dl decl:d
                {: dl.addLast(d);
                   RESULT = dl;
                :}
                | /* epsilon */
                {: RESULT = new LinkedList<AST.DeclNode>();
                :}
                ;

decl            ::= varDecl:v
                {: RESULT = v;
                :}
                | fnDecl:f
                {: RESULT = f;
                :}
```

```
            | structDecl:s
            {: RESULT = s;
            :}
            ;

type          ::= INT
            {: RESULT = new AST.IntNode();
            :}
            | BOOL
            {: RESULT = new AST.BoolNode();
            :}
            | VOID
            {: RESULT = new AST.VoidNode();
            :}
            ;

id           ::= ID:i
            {: RESULT = new AST.IdNode(i.linenum, i.charnum, i.idVal);
            :}
            ;

varDeclList    ::= varDeclList:vl varDecl: v
                        {: vl.addLast(v);
                          RESULT = vl;
                        :}
                        | /* epsilon */
                        {: RESULT = new LinkedList<AST.VarDeclNode>();
                        :}
                        ;

varDecl       ::= type:t id:i SEMICOLON
            {: RESULT = new AST.VarDeclNode(t, i, AST.VarDeclNode.NOT_STRUCT);
            :}
            | STRUCT id:i1 id:i2 SEMICOLON
            {: RESULT = new AST.VarDeclNode(new AST.StructNode(i1), i2, 0);
            :}
            ;

/* formals: FormalsListNode*/
fnDecl                  ::= type:t id:i formals:f fnBody:fb
                        {: RESULT = new AST.FnDeclNode(t, i, new AST.FormalsListNode(f), fb);
                        :}
                        ;

structDecl     ::= STRUCT id:i LCURLY structBody:sb RCURLY SEMICOLON
                        {: RESULT = new AST.StructDeclNode(i, new AST.DeclListNode(sb));
                        :}
                        ;

structBody        ::= structBody:sb varDecl:v
                        {: sb.addLast(v);
                          RESULT = sb;
                        :}
                        | varDecl:v
                        {: LinkedList<AST.DeclNode> l = new LinkedList<AST.DeclNode>();
                          l.addLast(v);
                          RESULT = l;
                        :}
                        ;

formals                 ::= LPAREN RPAREN
                        {: RESULT = new LinkedList<AST.FormalDeclNode>();
                        :}
                        | LPAREN formalsList:fl RPAREN
                        {: RESULT = fl;
                        :}
                        ;

formalsList       ::= formalDecl:f
                        {: LinkedList<AST.FormalDeclNode> l = new LinkedList<AST.FormalDeclNode>();
```

```
                                l.addLast(f);
                                RESULT = l;
                              :}
                              | formalDecl:f COMMA formalsList:fl
                              {: fl.addFirst(f);
                                RESULT = fl;
                              :}
                              ;

formalDecl        ::= type:t id:i
                              {: RESULT = new AST.FormalDeclNode(t, i);
                              :}
                              ;

fnBody        ::= LCURLY varDeclList:vl stmtList:sl RCURLY
                              {: RESULT = new AST.FnBodyNode(new AST.DeclListNode(vl), new AST.StmtListNode(sl));
                              :}
            ;

stmtList      ::= stmtList:sl stmt:s
                              {: sl.addLast(s);
                                RESULT = sl;
                              :}
            | /* epsilon */
            {: RESULT = new LinkedList<AST.StmtNode>();
            :}
            ;

stmt          ::= assignExp:a SEMICOLON
                              {: RESULT = new AST.AssignStmtNode((AST.AssignNode)a);
                              :}
            | loc:l PLUSPLUS SEMICOLON
            {: RESULT = new AST.PostIncStmtNode(l);
            :}
            | loc:l MINUSMINUS SEMICOLON
            {: RESULT = new AST.PostDecStmtNode(l);
            :}
            | CIN READ loc:l SEMICOLON
            {: RESULT = new AST.ReadStmtNode(l);
            :}
            | COUT WRITE exp:e SEMICOLON
            {: RESULT = new AST.WriteStmtNode(e);
            :}
            | IF LPAREN exp:e RPAREN LCURLY varDeclList:vl stmtList:sl RCURLY
            {: RESULT = new AST.IfStmtNode(e, new AST.DeclListNode(vl), new AST.StmtListNode(sl));
            :}
            | IF LPAREN exp:e RPAREN LCURLY varDeclList:vl1 stmtList:sl1 RCURLY ELSE LCURLY varDeclList:vl2
stmtList:sl2 RCURLY
            {: RESULT = new AST.IfElseStmtNode(e, new AST.DeclListNode(vl1), new AST.StmtListNode(sl1), new
AST.DeclListNode(vl2), new AST.StmtListNode(sl2));
            :}
            | WHILE LPAREN exp:e RPAREN LCURLY varDeclList:vl stmtList:sl RCURLY
            {: RESULT = new AST.WhileStmtNode(e, new AST.DeclListNode(vl), new AST.StmtListNode(sl));
            :}
            | RETURN exp:e SEMICOLON
            {: RESULT = new AST.ReturnStmtNode(e);
            :}
            | RETURN SEMICOLON
            {: RESULT = new AST.ReturnStmtNode(null);
            :}
            | fncall:fc SEMICOLON
            {: RESULT = new AST.CallStmtNode((AST.CallExpNode)fc);
            :}
            ;

/* right-associative & lowest precedence */
assignExp     ::= loc:l ASSIGN exp:e
                              {: RESULT = new AST.AssignNode(l, e);
                              :}
                              ;
```

```
exp             ::= assignExp:a
                    {: RESULT = a;
                    :}
                    | exp:e1 PLUS exp:e2
                    {: RESULT = new AST.PlusNode(e1, e2);
                    :}
                    | exp:e1 MINUS exp:e2
                    {: RESULT = new AST.MinusNode(e1, e2);
                    :}
                    | exp:e1 TIMES exp:e2
                    {: RESULT = new AST.TimesNode(e1, e2);
                    :}
                    | exp:e1 DIVIDE exp:e2
                    {: RESULT = new AST.DivideNode(e1, e2);
                    :}
                    | NOT exp:e
                    {: RESULT = new AST.NotNode(e);
                    :}
                    | exp:e1 AND exp:e2
                    {: RESULT = new AST.AndNode(e1, e2);
                    :}
                    | exp:e1 OR exp:e2
                    {: RESULT = new AST.OrNode(e1, e2);
                    :}
                    | exp:e1 EQUALS exp:e2
            {: RESULT = new AST.EqualsNode(e1, e2);
            :}
            | exp:e1 NOTEQUALS exp:e2
            {: RESULT = new AST.NotEqualsNode(e1, e2);
            :}
            | exp:e1 LESS exp:e2
            {: RESULT = new AST.LessNode(e1, e2);
            :}
            | exp:e1 GREATER exp:e2
            {: RESULT = new AST.GreaterNode(e1, e2);
            :}
            | exp:e1 LESSEQ exp:e2
            {: RESULT = new AST.LessEqNode(e1, e2);
            :}
            | exp:e1 GREATEREQ exp:e2
            {: RESULT = new AST.GreaterEqNode(e1, e2);
            :}
            | MINUS term:t
            {: RESULT = new AST.UnaryMinusNode(t);
            :}
            | term:t
            {: RESULT = t;
            :}
            ;

term        ::= loc:l
                    {: RESULT = l;
                    :}
            | INTLITERAL:i
            {: RESULT = new AST.IntLitNode(i.linenum, i.charnum, i.intVal);
            :}
            | STRINGLITERAL:s
            {: RESULT = new AST.StringLitNode(s.linenum, s.charnum, s.strVal);
            :}
            | TRUE:t
            {: RESULT = new AST.TrueNode(t.linenum, t.charnum);
            :}
            | FALSE:f
            {: RESULT = new AST.FalseNode(f.linenum, f.charnum);
            :}
            | LPAREN exp:e RPAREN
            {: RESULT = e;
            :}
            | fncall:fc
```

```
            {: RESULT = fc;
            :}
            ;

fncall        ::= id:i LPAREN RPAREN   // fn call with no args
                        {: RESULT = new AST.CallExpNode(i);
                        :}
            | id:i LPAREN actualList:al RPAREN  // with args
            {: RESULT = new AST.CallExpNode(i, new AST.ExpListNode(al));
            :}
            ;

actualList     ::= exp:e
                        {: LinkedList<AST.ExpNode> l = new LinkedList<AST.ExpNode>();
                          l.addLast(e);
                          RESULT = l;
                        :}
            | actualList:al COMMA exp:e
            {: al.addLast(e);
              RESULT = al;
            :}
            ;
loc           ::= ID:i
                        {: RESULT = new AST.IdNode(i.linenum, i.charnum, i.idVal);
                        :}
            | loc:l DOT id:i
            {: RESULT = new AST.DotAccessExpNode(l, i);
            :}
                        ;
```