Group No: 48
Puneet Anand 2016B4A70487P
Mayank Jasoria 2016B1A70703P
Shubham Tiwari 2016B4A70935P
Vibhav Oswal 2016B4A70594P

Common function definitions and assumptions:

1. makeLeaf(TerminalNode* addr)
   This function copies all attributes of a terminal node from the parse tree into a new Node, and returns a node of appropriate type.
   It can handle various types of terminal nodes, like NUMBER, ID, DATATYPE etc.
   @param addr Address of any terminal node.
2. makeLeaf(Char* lexeme)
   This function creates a leaf node for a given lexeme. This function is similar to the addition of a leaf node to a parse tree
   @param lexeme The lexeme for which the node is to be created

Attributes used:

1. syn: bottom-up passage of attributes
2. inh: top-down passage of attributes
3. node: Refers to a current node address
4. val: Represents a TAG used for identification purposes
5. width: the space required by variable in memory
6. tag: label for identifying the type of node.

Semantic rules for AST generation:

program : moduleDeclarations otherModules$_1$ driverModule otherModules$_2$
program.syn = new ProgramNode(moduleDeclarations.syn, otherModules$_1$.syn, driverModule.syn, otherModules$_2$.syn);
/* Synthesize program from moduleDeclarations, otherModules, driverModule */

moduleDeclarations : moduleDeclaration moduleDeclarations$_1$
moduleDeclarations.syn = new DeclarationNode(moduleDeclaration.syn, moduleDeclarations$_1$.syn) /* value of current node results from those synthesized by both child subtrees due to absence of interdependency for computation */

moduleDeclarations : EMPTY
moduleDeclarations.syn = NULL /* return NULL, to mark the end of subtree. No tag needed as we can just check isNull */

moduleDeclaration : DECLARE MODULE ID SEMICOL
moduleDeclaration.syn = makeLeaf(ID.addr) /* Simply create a makeLeaf node corresponding to ID */
moduleDeclaration.tag = ID

```
otherModules          : module otherModules1
     otherModules.syn = new ModuleNode(module.syn, otherModules1.syn) /* value of
current node results from those synthesized by both child subtrees */

otherModules          : EMPTY
     otherModules.syn = NULL /* return NULL, to mark the end of subtree. No tag
needed as we can just check isNull */

driverModule          : DRIVERDEF DRIVER PROGRAM DRIVERENDDEF moduleDef
     driverModule.syn = new ModuleNode(makeLeaf("DRIVER"), NULL, NULL,
moduleDef.syn);  /* No new information added by the rule, simply pass the address.
Driver module is a special module having the name DRIVER, and having no input and
output parameters, just a block of statements. So, we will use ModuleNode with the
above parameters to represent a driver module. */

module           : DEF MODULE ID ENDDEF TAKES INPUT SQBO input_plist SQBC SEMICOL ret
moduleDef
     module.syn = new ModuleNode(makeLeaf(ID.addr), input_plist.syn, ret.syn,
moduleDef.syn);
     /* input_plist will be a linked list of input parameters to the modules
        ret will also be a linked list of return values from the module
        moduleDef will be a linked list of statements
      */
ret              : RETURNS SQBO output_plist SQBC SEMICOL
     ret.syn = output_plist.syn /* output_plist is the only subtree with a value.
Directly propagate it upwards */

ret              :  EMPTY
     ret.syn = NULL; /* return NULL, to mark the end of subtree. No tag needed as we
can just check isNull */

input_plist           : ID COLON dataType n1
     input_plist.syn = new InputListNode(makeLeaf(ID.addr), dataType.syn, n1.syn); /*
Allocate new node consisting of ID, dataType and add to the head represented by
n1.syn */

n1         : COMMA ID COLON dataType n11
     n1.syn = new InputListNode(makeLeaf(ID.addr), dataType.syn, n11.syn); /* Allocate
new node consisting of ID, dataType and add to the head represented by n1.syn */

n1         : EMPTY
     n1.syn = NULL; /* return NULL, to mark the end of subtree. No tag needed as we
can just check isNull */

output_plist          : ID COLON type n2
     output_plist.syn = new OutputListNode(makeLeaf(ID.addr), type.syn, n2.syn);

n2               : COMMA ID COLON type n21
```

```
     n2.syn = new OutputListNode(makeLeaf(ID.addr), type.syn, n2_1.syn);

n2              :  EMPTY
     n2.syn = NULL; /* return NULL, to mark the end of subtree. No tag needed as we
can just check isNull */

type            : INTEGER
     type.syn = makeLeaf(INTEGER.addr) /* Directly propagate the terminal node
upwards */
     type.tag = INTEGER

type            : REAL
     type.syn = makeLeaf(REAL.addr) /* Directly propagate the terminal node upwards
*/
     type.tag = REAL

type            : BOOLEAN
     type.syn = makeLeaf(BOOLEAN.addr) /* Directly propagate the terminal node
upwards */
     type.tag = BOOLEAN

dataType        : type
     dataType.syn = new DataTypeNode(type.syn)  /* type is the only subtree. Directly
propagate it upwards */
     dataType.tag = INT_REAL_BOOL

dataType        : ARRAY SQBO range_arrays SQBC OF type
     dataType.syn = new DataTypeNode(range_arrays.syn, type.syn); /* synthesize
dataType with range_arrays and type */
     dataType.tag = ARRAY

moduleDef       : START statements END
     moduleDef.syn = statements.syn /* statements is the only subtree with a value.
Directly propagate it upwards */

statements      : statement statements_1
     statements.syn = new StatementNode(statement.syn, statements_1.syn) /* value of
current node results from those synthesized by both child subtrees due to absence of
interdependence for computation */
     statements.tag = statement.tag /* TAG to identify the type of statement */

statements      : EMPTY
     statements.syn = NULL /* return NULL, to mark the end of subtree. No tag needed
as we can just check isNull */

statement       : ioStmt
     statement.syn = ioStmt.syn /* ioStmt is the only subtree. Directly propagate it
upwards */
```

```
     statement.tag = IOSTMT; /* TAG to denote the type of statement */

statement       : simpleStmt
     statement.syn = simpleStmt.syn /* simpleStmt is the only subtree. Directly
propagate it upwards */
     statement.tag = SIMPLESTMT /* TAG to denote the type of statement */

statement       : declareStmt
     statement.syn = declareStmt.syn /* declareStmt is the only subtree. Directly
propagate it upwards */
     statement.tag = DECLARESTMT /* TAG to denote the type of statement */

statement       : condionalStmt
     statement.syn = condionalStmt.syn /* conditionalStmt is the only subtree.
Directly propagate it upwards */
     statement.tag = CONDIONALSTMT /* TAG to denote the type of statement */

statement       : iterativeStmt
     statement.syn = iterativeStmt.syn /* iterativeStmt is the only subtree. Directly
propagate it upwards */
     statement.tag = ITERATIVESTMT /* TAG to denote the type of statement */

ioStmt          : GET_VALUE BO ID BC SEMICOL
     ioStmt.syn = makeLeaf(ID.addr) /* Directly propagate the terminal node upwards
*/
     ioStmt.tag = GET_VALUE /* tag to denote type of operation performed by this IO
statement */

ioStmt          : PRINT BO var BC SEMICOL
     ioStmt.syn = var.syn  /* var is the only subtree. Directly propagate it upwards
*/
     ioStmt.tag = PRINT /* tag to denote type of operation performed by this IO
statement */

whichId         : SQBO index SQBC
     whichId.syn = index.syn /* index is the only subtree with a value. Directly
propagate it upwards */

whichId         : EMPTY
     whichId.syn = NULL; /* return NULL, to mark the end of subtree. No tag needed as
we can just check isNull */

index           : NUM
     index.syn = makeLeaf(NUM.addr);  /* Directly propagate the terminal node upwards
*/
     index.tag = NUM

index           : ID
```

```
     index.syn  = makeLeaf(ID.addr);  /* Directly propagate the terminal node upwards
*/
     index.tag = ID

simpleStmt      : assignmentStmt
     simpleStmt.syn = new SimpleStmtNode(assignmentStmt.syn); /* Simply propagate the
assignmentStmt node upwards */
     simpleStmt.tag = ASSIGNSTMT; /* Tag denotes that the statement is of type
'assignment statement' */

simpleStmt           : moduleReuseStmt
     simpleStmt.syn = new SimpleStmtNode(moduleReuseStmt.syn); /* Simply propagate
the moduleReuseStmt node upwards. */
     simpleStmt.tag = MODULEREUSESTMT; /* Tag denotes that the statement is of type
'module reuse statement statement' */

assignmentStmt  : ID whichStmt
     assignmentStmt.syn = new AssignNode(makeLeaf(ID.addr), whichStmt.syn); /*
Synthesize new node consisting of ID and whichStmt */
     if (whichStmt.tag == L_ID_STMT)
          *ID.addr = whichStmt.val
     else
          *(ID.addr + whichStmt.idx * sizeof(ID.type)) = whichStmt.val
/* We can get ID.type from the symbol table. ID.type will be populated during the
declaration statement of ID. If ID is not present in the symbol table, then it was
never declared, and we will throw an error. */

moduleReuseStmt : optional USE MODULE ID WITH PARAMETERS idList SEMICOL
     moduleReuseStmt.syn = new ModuleReuseNode(optional.syn, makeLeaf(ID.addr),
idList.syn); /* Create a new node using optional, ID and idList */

optional        : SQBO idList SQBC ASSIGNOP
     optional.syn = idList.syn; /* Send idList (a linked list) upwards */

optional        : EMPTY
     optional.syn = NULL;  /* return NULL, to mark the end of subtree. No tag needed
as we can just check isNull */

idList               : ID n3
     idList.syn = new idListNode(makeLeaf(ID.addr), n3.syn); /* Create new node of ID
and plug in n3.syn (head of linked list) to the next pointer of the new node, giving
a new linked list, with new node appended to the start*/

n3              : COMMA ID n3_1
     n3.syn = new idListNode(makeLeaf(ID.addr), n3_1.syn); /* Create new node of ID and
plug in n3.syn (head of linked list) to the next pointer of the new node, giving a
new linked list, with new node appended to the start*/
```

```
n3              : EMPTY
     n3.syn = NULL; /* return NULL, to mark the end of subtree. No tag needed as we
can just check isNull */

expression                  :  arithmeticOrBooleanExpr
     expression.syn = new ExprNode(arithmeticOrBooleanExpr.syn); /* Propagate AOBExpr
upwards */
     expression.val = arithmeticOrBooleanExpr.val
     expression.tag = AOBEXPR;                  /* Tag denotes AOBEXPR */

expression      : u
     expression.syn = new ExprNode(u.syn);     /* Propagate unary upwards */
     expression.val = u.val
     expression.tag = UNARY;                   /* Tag denotes UNARY */

arithmeticOrBooleanExpr : anyTerm n7
     n7.inh = anyTerm.syn
     arithmeticOrBooleanExpr.syn = n7.syn; /* AOBExpr synthesized by anyTerm and n7
*/

n7    : logicalOp anyTerm n7[1]
     n7[1].inh = new AOBExprNode(logicalOp.syn, n7.inh, anyTerm.syn)
     n7.syn = n7[1].syn

n7    : EMPTY
     n7.syn = n7.inh

anyTerm         : arithmeticExpr n8
     n8.inh = arithmeticExpr.syn
     anyTerm.syn = n8.syn
     anyTerm.tag = arithmeticExpr.tag

anyTerm         : boolConstt
     anyTerm.syn = boolConstt.syn
     anyTerm.tag = BOOLCONSTT

n8         : relationalOp arithmeticExpr
     n8.syn = new AOBExprNode(relationalOp.syn, n8.inh, arithmeticExpr.syn)

n8         :EMPTY
     n8.syn = n8.inh

arithmeticExpr  : term n4
     n4.inh = term.syn
     arithmeticExpr.syn = n4.syn
     arithmeticExpr.val = n4.val

n4    : op1 term n4[1]
```

```
      n4_1.inh = new AOBExprNode(op1.syn, n4.inh, term.syn)
      n4.syn = n4_1.syn
      if(op1.val == PLUS) {
            n4.val = n4_1.val + factor.val;
      } else {
            n4.val = n4_1.val - factor.val;
      }

n4    : EMPTY
      n4.syn = n4.inh

term        : factor n5
      n5.inh = factor.syn
      term.syn = n5.syn
      term.val = n5.val

n5          : op2 factor n5_1
      n5_1.inh = new AOBExprNode(op2.syn, n5.inh, factor.syn)
      n5.syn = n5_1.syn
      if(op2.val == MUL) {
            n5.val = n5_1.val * factor.val;
      } else {
            n5.val = n5_1.val / factor.val;
      }

n5          : EMPTY
      n5.syn = n5.inh

factor            : BO arithmeticOrBooleanExpr BC
      factor.syn = arithmeticOrBooleanExpr.syn
      factor.val = arithmeticOrBooleanExpr.val
      // factor.tag = EXPR

factor            : var_id_num
      factor.syn = var_id_num.syn /* var_id_num is the only subtree. Directly
propagate it upwards */
      factor.val = var_id_num.val
      // factor.tag = ID_NUM

var         : var_id_num
      var.syn = new AOBExprNode(NULL, NULL, var_id_num.node) /* var_id_num is the only
subtree. Directly propagate it upwards */
      var.tag = var_id_num.tag /* TAG to denote that the var is of type ID or NUM */

var               : boolConstt
      var.syn = new AOBExprNode(NULL, NULL, boolConstt.syn) /* boolConstt is the only
subtree. Directly propagate it upwards */
```

```
     var.tag = BOOLCONSTT /* TAG to denote that the var is of type Boolean Constant
*/

op1             : PLUS
     op1.syn = makeLeaf(PLUS.addr)  /* Directly propagate the terminal node upwards
*/
     op1.tag = PLUS

op1             : MINUS
     op1.syn = makeLeaf(MINUS.addr)  /* Directly propagate the terminal node upwards
*/
     op1.tag = MINUS

op2             : MUL
     op2.syn = makeLeaf(MUL.addr)  /* Directly propagate the terminal node upwards */
     op2.tag = MUL

op2             : DIV
     op2.syn = makeLeaf(DIV.addr)  /* Directly propagate the terminal node upwards */
     op2.tag = DIV

logicalOp       : AND
     logicalOp.syn = makeLeaf(AND.addr)  /* Directly propagate the terminal node
upwards */
     logicalOp.tag = AND

logicalOp       : OR
     logicalOp.syn = makeLeaf(OR.addr)  /* Directly propagate the terminal node
upwards */
     logicalOp.tag = OR

relationalOp         : LT
     relationalOp.syn = makeLeaf(LT.addr)  /* Directly propagate the terminal node
upwards */
     relationalOp.tag = LT

relationalOp         : LE
     relationalOp.syn = makeLeaf(LE.addr)  /* Directly propagate the terminal node
upwards */
     relationalOp.tag = LE

relationalOp         : GT
     relationalOp.syn = makeLeaf(GT.addr)  /* Directly propagate the terminal node
upwards */
     relationalOp.tag = GT

relationalOp         : GE
```

```
     relationalOp.syn = makeLeaf(GE.addr)  /* Directly propagate the terminal node
upwards */
     relationalOp.tag = GE

relationalOp          : EQ
     relationalOp.syn = makeLeaf(EQ.addr)  /* Directly propagate the terminal node
upwards */
     relationalOp.tag = EQ

relationalOp          : NE
     relationalOp.syn = makeLeaf(NE.addr)  /* Directly propagate the terminal node
upwards */
     relationalOp.tag = NE

declareStmt           : DECLARE idList COLON dataType SEMICOL
     declareStmt.syn = new DeclareStmtNode(idList.syn, dataType.syn); /* Synthesize
declareStmt using idList and dataType */
     for(items: idList.syn) { items.width = sizeof(dataType.val); items.type =
dataType.syn }

condionalStmt         : SWITCH BO ID BC START caseStmts default END
     conditionalStmt.syn = new CondStmtNode(makeLeaf(ID.addr), caseStmts.syn,
default.syn); /* Synthesize conditionalStmt using ID, caseStmt and default  */

caseStmts       : CASE value COLON statements BREAK SEMICOL n9
     caseStmt.syn = new CaseStmtNode(value.syn, statements.syn, n9.syn); /* Create
new node for case statement block and add to the head of the linked list of case
statement blocks obtained from n9.syn */

n9         : CASE value COLON statements BREAK SEMICOL n9_1
     n9.syn = new CaseStmtNode(value.syn, statements.syn, n9_1.syn); /* Create new node
for case statement block and add to the head of the linked list of case statement
blocks obtained from n9.syn */

n9              : EMPTY
     n9.syn = NULL; /* return NULL, to mark the end of subtree. No tag needed as we
can just check isNull */

value           : NUM
     value.syn = makeLeaf(NUM.addr)  /* Directly propagate the terminal node upwards
*/
     value.tag = NUM

value           : TRUE
     value.syn = makeLeaf(TRUE.addr)  /* Directly propagate the terminal node upwards
*/
     value.tag = TRUE
```

```
value           : FALSE
     value.syn = makeLeaf(FALSE.addr) /* Directly propagate the terminal node upwards
*/
     value.tag = FALSE

default              : DEFAULT COLON statements BREAK SEMICOL
     default.syn = statements.syn; /* Propagate statements upwards */

default              : EMPTY
     default.syn = NULL; /* return NULL, to mark the end of subtree. No tag needed as
we can just check isNull */

iterativeStmt        : FOR BO ID IN range BC START statements END
     iterativeStmt.syn = new IterStmtNode(new ForNode(makeLeaf(ID.addr), range.syn,
statements.syn)); /* Synthesize iterativeStmt using ID, range statement */
     iterativeStmt.tag = FOR; /* Tag indicates FOR loop */

iterativeStmt        : WHILE BO arithmeticOrBooleanExpr BC START statements END
     iterativeStmt.syn = new IterStmtNode(new WhileNode((arithmeticOrBooleanExpr.syn,
statements.syn)); /* Synthesize iterativeStmt using expression, range statement */
     iterativeStmt.tag = WHILE; /* Tag indicates WHILE loop */

range           : NUM1 RANGEOP NUM2
     range.syn = new RangeArraysNode(makeLeaf(NUM1.addr), makeLeaf(NUM2.addr))

range_arrays         : index1 RANGEOP index2
     range_arrays.syn = new RangeArraysNode(index1.syn, index2.syn)

boolConstt      : TRUE
     boolConstt.syn = makeLeaf(TRUE.addr)  /* Directly propagate the terminal node
upwards */
     boolConstt.tag = TRUE

boolConstt      : FALSE
     boolConstt.syn = makeLeaf(FALSE.addr)  /* Directly propagate the terminal node
upwards */
     boolConstt.tag = FALSE

var_id_num      : ID whichId
     var_id_num.syn = new varIdNumNode(makeLeaf(ID.addr), whichId.syn) /*
     var_id_num.tag = ID

var_id_num      : NUM
     var_id_num.syn = makeLeaf(NUM.addr)  /* Directly propagate the terminal node
upwards */
     var_id_num.tag = NUM

var_id_num      : RNUM
```

```
     var_id_num.syn = makeLeaf(RNUM.addr)  /* Directly propagate the terminal node
upwards */
     var_id_num.tag = RNUM

whichStmt            : lvalueIDStmt
     whichStmt.syn = new WhichStmtNode(lvalueIDStmt.syn)
     whichStmt.tag = L_ID_STMT

whichStmt       : lvalueARRStmt
     whichStmt.syn = new WhichStmtNode(lvalueARRStmt.syn)
     whichStmt.tag = L_ARR_STMT

lvalueIDStmt         : ASSIGNOP expression SEMICOL
     lvalueIDStmt.syn = expression.syn
     lvalueIDStmt.val = expression.val

lvalueARRStmt   : SQBO index SQBC ASSIGNOP expression SEMICOL
     lvalueARRStmt.syn = new lvalueARRStmtNode(index.syn, expression.syn)
     lvalueARRStmt.val = expression.val

u                          : unary_op new_NT
     u.syn = new UnaryNode(unary_op.syn, new_NT.syn)
     if(unary_op.val == MINUS) { u.val = -1 * new_NT.val }

unary_op                   : PLUS
     unary_op.syn = makeLeaf(PLUS.addr)  /* Directly propagate the terminal node
upwards */
     unary_op.tag = PLUS

unary_op        : MINUS
     unary_op.syn = makeLeaf(MINUS.addr)  /* Directly propagate the terminal node
upwards */
     unary_op.tag = MINUS

new_NT                     : BO arithmeticExpr BC
     new_NT.syn = arithmeticExpr.syn
     new_NT.tag = arithmeticExpr.tag

new_NT          :  var_id_num
     new_NT.syn =    new AOBExprNode(NULL, NULL, var_id_num.node)
     new_NT.tag = var_id_num.tag
```

Node	Fields
ProgramNode	declaration_head (Points to the head of the linked list corresponding to module declarations) othermodule$_1$_head (Points to the head of the linked list

	corresponding to modules) drivermodule_ptr (Pointer to the AST node corresponding to driver) othermodule$_2$_head (Points to the head of the link list representing other modules)
DeclarationNode	ptr (Points to the leaf node representing the current node) next (Points to the next DeclarationNode in the linked list)
ModuleListNode	ptr (Points to the node representing current ModuleListNode) next (Points to the node representing next ModuleListNode)
ModuleNode	name_ptr (Points to the record representing the name of the module) input_plist_head (Points to the linked list corresponding to input parameters) output_plist_head (Points to the linked list corresponding to output parameters) moduledef_head (Points to the linked list representing module definition)
InputListNode	ptr (Points to the leaf node corresponding to the current parameter) ptr_type (Points to the leaf node corresponding to the type node current parameter) next (Points to the linked list representing Input Parameters)
OutputListNode	ptr (Points to the leaf node representing identifier) type (Points to the node representing the type of output parameter) next (Points to the next node of the linked list representing output parameters)
DataTypeNode	range_arrays_ptr (Points to the node representing range_arrays) type_ptr (Points to the record representing the type of the data) tag (Tag representing whether it is an array type or primitive type)
RangeArraysNode	index$_1$ (Points to the index leaf node) index$_2$ (Points to the index leaf node)
StatementNode	ptr (Points to the node representing the AST node corresponding to the current statement, it is type Union) next (Points to the node corresponding to the next statement in the linked list) tag (Tag representing the type of statement)
SimpleStmtNode	ptr (Points to the node representing the AST node corresponding to the simple statement node, it is type Union) tag (Tag representing the type of simple statement: Assignment Statement or Module Reuse Statement)
AssignNode	lhs_ptr (Points to the leaf node representing the name of the left operand)

	whichstmt_ptr (Points to the node representing whichStmt)
whichStmtNode	lvalue_ptr (Points to the node representing the left value of the expression) tag (Tag representing either lvalueID or lvalueARR)
ModuleReuseNode	option_ptr (Points to the AST node corresponding to the option field) ptr (Points to the leaf corresponding) ParamListHead (Points to the head of the linked list corresponding to parameter list)
idListNode	id_ptr (Points to the node representing the pointer) next (Points to the next node in the linked list representing list of identifiers)
ExprNode	ptr (Points to the AOBExpr node or unary expression node) val (Stores the value of the node) tag (Represents either AOBExpr node or unary node)
AOBExprNode	op (Points to the AST leaf node corresponding to the operator) firstOperand (Points to the AST node corresponding to the first operand of the expression) secondOperand (Points to the AST node corresponding to the second operand of the expression) tag (Tag represents the type of expression)
DeclareStmtNode	idList_head (Points to the head of the Linked List of idList) ptr (Points to the node storing information about the dataType)
CondStmtNode	ptr (Points to the leaf node corresponding to the switch variable) head (Points to the head of the linked list corresponding to the different cases) defaultPtr (Points to the AST node corresponding to the default statement)
CaseStmtNode	ptr (Points to the leaf node corresponding to the value of the current case) stmtPtr (Points to the head of the linked list of statements to be executed for the current case) next (Points to the next CaseStmtNode in the linked list of cases)
UnaryNode	Operator (Points to the operator to be applied) Operand (Points to the operand on which the operation is applied)
lvalueARRStmtNode	idx (Points to the AST node corresponding to the index) expr (Points to the AST node corresponding to the expression)
IterStmtNode	ptr (Points to the AST node corresponding to the FOR/WHILE node) tag (Tag represents the type of iterative statement)
ForNode	ptr (Points to the AST node corresponding to the ID at leaf)

(temporary node structure)	range (Points to a node specifying the range of values which the loop counter can take) stmts_head (Points to the head of the list of statements contained within the for loop)
WhileNode (temporary node structure)	expr (Points to the AST node corresponding to the expression of the While loop) stmt (Points to the head of the linked list of statements to be executed in each iteration)