ERPLAG Grammar

with

AST Formation Rules

prepared by

GROUP - 29

ANIRUDDHA JAYANT KARAJGI	2017A7PS0084P
SUBHAM KUMAR DASH	2017A7PS0004P
AYUSH GARG	2017A7PS0193P
RAHUL JHA	2017A7PS0036P
MEET KANANI	2017A7PS0128P

IN PARTIAL FULFILMENT OF THE COURSE
COMPILER CONSTRUCTION
(CS F363)

<program> → <moduleDeclarations> <otherModules> <driverModule> <otherModules>

{

program.node = new Node("program", moduleDeclarations.node, otherModules.node,
driverModule.node, otherModules.node);

free(moduleDeclarations);

free(otherModules);

free(driverModule);

free(otherModules);

}

<moduleDeclarations> → <moduleDeclaration> <moduleDeclarations$_1$>

{

moduleDeclarations.node = appendInOrder(moduleDeclaration.node,
moduleDeclarations$_1$.node)

free(moduleDeclaration);

free(moduleDeclarations$_1$);

}

<moduleDeclarations> → e

{

moduleDeclarations.node = NULL;

}

<moduleDeclaration> → DECLARE MODULE ID SEMICOL

{

moduleDeclaration.node = new Node("module",ID);

free(DECLARE);

free(MODULE);

free(SEMICOL);

}

<otherModules> → <module> <otherModules$_1$>

{

otherModules.node = appendInOrder(module.node, otherModules$_1$.node);

free(module);

free(otherModules$_1$);

}

<otherModules> → e

{

otherModules.node = NULL;

}

<driverModule> → DRIVERDEF DRIVER PROGRAM DRIVERENDDEF <moduleDef>

{

driverModule.node = new Node("Driver module", moduleDef.node);

free(DRIVERDEF);

free(DRIVER);

free(PROGRAM);

free(DRIVERENDDEF);

free(moduleDef);

}

<module> → DEF MODULE ID ENDDEF TAKES INPUT SQBO <input_plist> SQBC SEMICOL <ret> <moduleDef>

{

module.node = new Node("module",ID, input_plist.node, ret.node, moduleDef.node);

free(DEF);

free(MODULE);

free(ENDDEF);

free(TAKES);

free(INPUT);

free(SQBO);

free(SQBC);

free(SEMICOL);

}

<ret> → RETURNS SQBO <output_plist> SQBC SEMICOL

{

ret.node = output_plist.node;

free(RETURNS);

free(SQBO);

free(SQBC);

free(SEMICOL);

}

<ret> → e

{

ret.node = NULL;

}

<input_plist> → ID COLON <dataType> <input_plist_dash>

{

input_plist.node = new Node("input_plist", ID, dataType.node, input_plist_dash.node);

free(COLON);

free(dataType);

free(input_plist_dash);

}

<input_plist_dash> → COMMA ID COLON <dataType> <input_plist_dash$_1$>

{

input_plist_dash.node = appendInOrder(ID, dataType.node, input_plist_dash$_1$.node);

free(COMMA);

free(COLON);

free(datatype);

free(input_plist_dash$_1$);

}

<input_plist_dash> → e

{

input_plist_dash.node = NULL;

}

<output_plist> → ID COLON <type> <output_plist_dash>

{

output_plist.node=new Node("output_plist", ID, type.node, output_plist_dash.node);

free(COLON);

free(type);

free(output_plist_dash);

}

<output_plist_dash> → COMMA ID COLON <type> <output_plist_dash$_1$>

{

output_plist_dash = appendInOrder(ID, type.node, output_plist_dash$_1$.node);

free(COMMA);

free(COLON);

free(type);

free(output_plist_dash$_1$);

}

<output_plist_dash> → e

{

Output_plist_dash.node = NULL;

}

<dataType> → INTEGER

{

dataType.node = new Leaf("integer",INTEGER);

}

<dataType> → BOOLEAN

{

dataType.node = new Leaf("boolean", BOOLEAN);

}

<dataType> → REAL

{

dataType.node = new Leaf("real", REAL);

}

<dataType> → ARRAY SQBO <range_array> SQBC OF <type>

{

dataType.node = new Node("dataType", range_array.node, type.node);

free(ARRAY);

free(SQBO);

free(SQBC);

free(OF);

free(range_array);

free(type);

}

<range_array> → <index$_1$> RANGEOP <index$_2$>

{

range_array.node = new Node("range_array", index$_1$.node, index$_2$.node);

free(index$_1$);

free(index$_2$);

free(RANGEOP);

}

<type> → INTEGER

{

type.node = new Leaf("integer",INTEGER);

}

<type> → REAL

{

type.node = new Leaf("real", REAL);

}

<type> →BOOLEAN

{

type.node = new Leaf("boolean", BOOLEAN);

}

<moduleDef> → START <statements> END

{

moduleDef.node = statements.node;

free(statements);

free(START);

free(END);

}

<statements> → <statement> <statements$_1$>

{

statements.node = appendInOrder(statement.node, statements$_1$.node);

free(statement);

free(statements$_1$);

}

<statements> → e

{

statements.node = NULL:

}

<statement> → <ioStmt>

{

Statement.node = ioStmt.node;

free(ioStmt);

}

<statement> → <simpleStmt>

{

statement.node = simpleStmt.node;

free(simpleStmt);

}

<statement> → <declareStmt>

{

statement.node = declareStmt.node;

free(declareStmt);

}

<statement> → <conditionalStmt>

{

statement.node = conditionalStmt.node;

free(conditionalStmt);

}

<statement> → <iterativeStmt>

{

statement.node = iterativeStmt.node;

free(iterativeStmt);

}

<ioStmt> → GET_VALUE BO ID BC SEMICOL

{

ioStmt.node = new Node('ioStmt_get_value', ID);

free(GET_VALUE);

free(BO);

free(BC);

free(SEMICOL);

}

<ioStmt> → PRINT BO <var> BC SEMICOL

{

ioStmt.node = new Node('ioStmt_print', var.node);

free(PRINT);

free(BO);

free(BC);

free(var);

free(SEMICOL);

}

<var> → <var_id_num>

{

var.node = var_id_num.node;

free(var_id_num);

}

<var> → <boolConst>

{

var.node = boolConst.node;

free(boolConst);

}

<var_id_num> → ID <whichId>

{

var_id_num.node = new Node("var_id_num", ID, whichId.node);

free(whichId);

}

<var_id_num> → NUM

{

var_id_num.node = new Leaf("num", NUM);

}

<var_id_num> → RNUM
{
    var_id_num.node = new Leaf("rnum", RNUM);
}

<boolConst> → TRUE
{
    boolConst.node = new Leaf("true", TRUE);
}

<boolConst> → FALSE
{
    boolConst.node = new Leaf("false",FALSE);
}

<whichId> → SQBO <index> SQBC
{
    whichId.node = index.node
    free(SQBO)
    free(SQBC)
    free(index)
}

<whichId> → e
{
    whichId.node = NULL
}

<simpleStmt> → <assignmentStmt>
{
    simpleStmt.node = assignmentStmt.node
    free(assignmentStmt)
}

<simpleStmt> → <moduleReuseStmt>
{
    simpleStmt.node = moduleReuseStmt.node
    free(moduleReuseStmt)
}

<assignmentStmt> → ID <whichStmt>

{

assignmentStmt.node = new Node("assignmentStmt", ID, whichStmt.node);

free(whichStmt);

}

<whichStmt> → <lvalueIDStmt>

{

whichStmt.node = lvalueIDStmt.node;

free(lvalueIDStmt);

}

<whichStmt> → <lvalueARRStmt>

{

whichStmt.node = lvalueARRStmt.node;

free(lvalueARRStmt);

}

<lvalueIDStmt> → ASSIGNOP <expression> SEMICOL

{

lvalueIDStmt.node = new Node("lvalueIDStmt, ASSIGNOP, expression.node)

free(SEMICOL);

free(expression);

}

<lvalueARRStmt> → SQBO <index> SQBC ASSIGNOP <expression> SEMICOL

{

lvalueARRStmt.node = new Node("lvalueARRStmt",index.node, ASSIGNOP, expression.node)

free(SQBO);

free(SQBC);

free(SEMICOL);

free(index);

free(expression);

}

<index> → NUM

{

index.node = new Leaf("num", NUM);

}

<index> → ID

{

index.node = new Leaf("id",ID);

}

<moduleReuseStmt> → <optional> USE MODULE ID WITH PARAMETERS <idList> SEMICOL

{

moduleReuseStmt.node = new Node("moduleReuseStmt", optional.node, ID, idList.node)

free(USE);

free(MODULE);

free(WITH);

free(PARAMETERS);

free(SEMICOL);

free(optional);

free(idList);

}

<optional> → SQBO <idList> SQBC ASSIGNOP

{

optional.node = new Node("optional", idList.node, ASSIGNOP);

free(SQBO);

free(SQBC);

free(idList);

}

<optional> → e

{

optional.node = NULL

}

<idList> → ID <idList_dash>

{

idList.node = new Node("idList", ID, idList_dash.node);

free(idList_dash);

}

<idList_dash$_1$> → COMMA ID <idList_dash$_2$>

{

idList_dash$_1$.node = appendInOrder(ID, idList_dash$_2$.node);

free(COMMA);

free(idList_dash$_2$);

}

<idList_dash> → e

```
{
        idList_dash.node = NULL;
}
```

<expression> → <arithmeticOrBooleanExpr>

```
{
        expression.node = arithmeticOrBooleanExpr.node;
        free(arithmeticOrBooleanExpr);
}
```

<expression> → <op_plus_minus> <unaryOrExpr>

```
{
        (
                unaryOrExpr.inh = new createNode("expression", op_plus_minus.val);
                free(op_plus_minus);
        )
        (
                expression.node = unaryOrExpr.inh;
                free(unaryOrExpr);
        )
}
```

<unaryOrExpr> → BO <arithmeticExpr> BC

```
{
        //child pointer for the node created above using createNode() is populated here
        unaryOrExpr.inh.node = arithmeticExpr.syn;

        free(BO);
        free(BC);
        free(arithmeticExpr);
}
```

<unaryOrExpr> → <var_id_num>

```
{
        unaryOrExpr.inh.node = var_id_num.node;
        free(var_id_num);
}
```

<arithmeticOrBooleanExpr> → <anyTerm> <bool>

{

(

bool.inh=anyTerm.syn;

free(anyTerm);

)

(

arithmeticOrBooleanExpr.node = bool.syn;

free(bool);

)

}

<anyTerm> → <arithmeticExpr> <arithmeticExpr_dash>

{

(

arithmeticExpr_dash.inh = arithmeticExpr.syn;

free(arithmeticExpr);

)

(

anyTerm.syn=arithmeticExpr_dash.syn;

free(arithmeticExpr_dash);

)

}

<anyTerm>→ <boolConst>

{

anyTerm.syn = boolConst.syn;

free(boolConst);

}

<arithmeticExpr_dash> → <relationalOp> <arithmeticExpr>

{

(

arithmeticExpr.inh=new createNode("arithmeticExpr_dash",relationalOp.val,

arithmeticExpr_dash.inh);

free(relationalOp);

)

(

```
            arithmeticExpr_dash.syn = arithmeticExpr.syn;
            free(arithmeticExpr);
        )
}
```

<arithmeticExpr_dash> → e

```
{
        arithmeticExpr_dash.syn = arithmeticExpr_dash.inh;
}
```

<bool$_1$> → <logicalOp> <anyTerm> <bool$_2$>

```
{
        (
            bool2.inh =new createNode("bool", logicalOp.val, bool1.inh, anyTerm.syn);
            free(logicalOp);
            free(anyTerm);
        )
        (
            bool1.syn=bool2.syn;
            free(bool2);
        )
}
```

<bool> → e

```
{
         bool.syn = bool.inh;
}
```

<arithmeticExpr> → <term><arithmeticExpr_recur>

```
{
        (
            arithmeticExpr_recur.inh = term.syn;
            free(term);
        )
        (
            arithmeticExpr.syn = arithmeticExpr_recur.syn;
            free(arithmeticExpr_recur);
        )
}
```

<arithmeticExpr_recur$_1$> → <op_plus_minus> <term> <arithmeticExpr_recur$_2$>

{

(

arithmeticExpr_recur$_2$.inh = new createNode("arithmeticExpr_recur",op_plus_minus.val, arithmeticExpr_recur$_1$.inh, term.syn);

free(op_plus_minus);

free(term);

)

(

arithmeticExpr_recur$_1$.syn = arithmeticExpr_recur$_2$.syn;

free(arithmeticExpr_recur$_2$);

)

}

<arithmeticExpr_recur> → e

{

arithmeticExpr_recur.syn = arithmeticExpr_recur.inh ;

}

<term> → <factor> <term_dash>

{

(

term_dash.inh = factor.syn;

free(factor);

)

(

term.syn = term_dash.syn;

free(term_dash);

)

}

<term_dash$_1$> → <op_mul_div> <factor> <term_dash$_2$>

{

(

term_dash$_2$.inh = new createNode("term_dash",op_plus_minus.val, term_dash$_1$.inh, factor.syn);

free(op_mul_div);

free(factor);

)

(

```
            term_dash1.syn = term_dash2.syn;
            free(term_dash2);
        )
}

<term_dash> → e
{
        term_dash.syn = term_dash.inh;
}

<factor> →  BO <arithmeticOrBooleanExpr> BC
{
        factor.syn =  arithmeticOrBooleanExpr.node;
        free(BO);
        free(arithmeticOrBooleanExpr);
        free(BC);
}

<factor> → <var_id_num>
{
        factor.syn =  var_id_num.node;
        free(var_id_num);
}

<op_plus_minus> →  PLUS
{
        op_plus_minus.node = new Leaf("plus",PLUS);
}

<op_plus_minus> →  MINUS
{
        op_plus_minus.node = new Leaf("minus",MINUS);
}

<op_mul_div> → MUL
{
        op_mul_div.node = new Leaf("mul", MUL);
 }

<op_mul_div> → DIV
{
        op_mul_div.node = new Leaf("div", DIV);
```

```
}

<logicalOp> →  AND
{
        logicalOp.node = new Leaf("and", AND);
}

<logicalOp> →  OR
{
        logicalOp.node = new Leaf("or", OR);
}

<relationalOp> →  GT
{
        relationalOp.node = new Leaf("gt",GT);
}

<relationalOp> →  LT
{
        relationalOp.node = new Leaf("lt", LT);
}

<relationalOp> →  GE
{
        relationalOp.node = new Leaf("ge", GE);
}

<relationalOp> →  LE
{
        relationalOp.node = new Leaf("le", LE);
}

<relationalOp> →  EQ
{
        relationalOp.node = new Leaf("eq", EQ);
}

<relationalOp> →  NE
{
        relationalOp.node = new Leaf("ne", NE);
}
```

<declareStmt> → DECLARE <idList> COLON <dataType> SEMICOL

{

declareStmt.node = new Node("declareStmt",idList.node, dataType.node);

free(DECLARE);

free(COLON);

free(SEMICOL);

free(idList);

free(dataType);

}

<conditionalStmt> → SWITCH BO ID BC START <caseStmts> <default> END

{

conditionalStmt.node = new Node("conditionalStmt",ID, caseStmts.node, default.node);

free(SWITCH);

free(BO);

free(BC);

free(START);

free(END);

free(caseStmts);

free(default);

}

<caseStmts> → CASE <value> COLON <statements> BREAK SEMICOL <caseStmt>

{

caseStmts.node = new Node("caseStmts", value.node, statements.node, caseStmt.node);

free(CASE);

free(COLON);

free(BREAK);

free(SEMICOL);

free(value);

free(statements);

free(caseStmt);

}

<caseStmt> → CASE <value> COLON <statements> BREAK SEMICOL <caseStmt$_1$>

{

caseStmt.node = appendInOrder(value.node, statements.node, caseStmt$_1$.node);

free(CASE);

free(COLON);

free(BREAK);

free(SEMICOL);

```
        free(value);
        free(statements);
        free(caseStmt1);
}

<caseStmt> → e
{
        caseStmt.node = NULL;
}


<value> →  NUM
{
        value.node = new Leaf("num", NUM);
}

<value> → TRUE
{
        value.node = new Leaf("true", TRUE);
}

<value> → FALSE
{
        value.node = new Leaf("false", FALSE);
}

<default> →  DEFAULT COLON <statements> BREAK SEMICOL
{
        default.node = statements.node;
        free(DEFAULT);
        free(COLON);
        free(BREAK);
        free(SEMICOL);
        free(statements);
}
<default> → e
{
        default.node = NULL;
}
```

<iterativeStmt> → FOR BO ID IN <range> BC START <statements> END

{

iterativeStmt.node = new Node("for", ID, range.node, statements.node);

free(FOR);

free(BO);

free(IN);

free(range);

free(BC);

free(START);

free(statements);

free(END);

}

<iterativeStmt> → WHILE BO <arithmeticOrBooleanExpr> BC START <statements> END

{

iterativeStmt.node = new Node("while", arithmeticOrBooleanExpr.node, statements.node);

free(WHILE);

free(BO);

free(arithmeticOrBooleanExpr);

free(BC);

free(START);

free(statements);

free(END);

}

<range> → NUM_1 RANGEOP NUM_2

{

range.node = new Node("range", NUM_1.node, RANGEOP.node, NUM_2.node);

}

Attributes used for the AST struct:

- **node** → pointer to the children of the current node
- **syn** → pointer to the child node that should be passed to the parent
- **inh** → contains the pointer formed by inheriting parent and/or sibling nodes
- **val** → contains the metadata of the node

Functions used for AST construction:

- **Node()** → populates the node attribute by forming a linked list of the supplied parameters.
- **createNode()** → creates a new AST Node with the value passed as the metadata and the pointers passed as the children of the node. If no pointers are passed, node attribute is initialized to NULL.
- **appendInOrder()** → appends the pointers passed to the linked list in the node attribute.
- **Leaf()** → creates a new Leaf Node with the parameters passed as meta data.