```java
/****
 *                                                               <p align=center>
 *              ------------------------------------------------            <br>
 *                        SIMPLE INTRODUCTORY VERSION,                      <br>
 *               WITH ASSIGNMENT, ADDITION, AND MULTIPLICATION,             <br>
 *                          ON INTEGER TYPES ONLY                           <br>
 *              ------------------------------------------------            <br>
 *                                                                         </p>
 *                                                                          <p>
 * PascalInterpreter is a parse-tree interpreter for a subset of the Pascal
 * programming language.  Its primary public method is eval, which takes a
 * parse tree and symbol table, and evaluates the tree if it is an executable
 * portion of a program.
 *                                                                          <p>
 * The interpreter works with an Object-valued program memory array.  Prior to
 * calling eval, the initInterpreter method is used to allocate array storage
 * and perform other initialization.  The constructor calls initInterpreter,
 * and it may be called subsequently for reinitialization.
 *                                                                          <p>
 * Calls to eval use the memory array as the working store for the program
 * being evaluated.  The memory retains its state between calls to eval, unless
 * initInterpreter is called again, or an already-allocated memory is
 * re-initialized to null using the clearMemory method.
 *                                                                          <p>
 * All of the public methods called by eval are publicly accessible.
 * However, they are likely of lesser value to outside callers, given that eval
 * dispatches to the appropriate submethods based on the type of tree it
 * receives.
 *                                                                          <p>
 * See the method comments for further details.
 *
 */
public class PascalInterpreter {

    /**
     * Construct this with the given static pool and stack size.  Call
     * initInterpter to do the work.
     */
    public PascalInterpreter (int staticSize, int stackSize) {
        initInterpreter(staticSize, stackSize);
    }

    /**
     * Evaluate the given tree node, in the context of the given symbol table.
     * The tree must be that of an executable Pascal construct, which is one of
     * the following: a list of statements, a single statement, or an
     * expression.
     *                                                                      <p>
     * For statement evaluation, the output of eval is a modified program
     * memory.  For expression evaluation, output is the returned Value, and a
     * modified memory if the expression has side effects.  The evaluation of
     * statements and statement lists always produces a null return value.
     *
     * CSC 330 NOTE: The statement structure of EJay is a bit different than
     * Pascal.  In particular, data decls are intermingled with stmts in a
     * block.  This is not the case in a Pascal compound statement.  So, you'll
     * have to do things a bit differently at the top level of the EJay eval
     * method than is done here for Pascal.  One thing for sure, you should not
     * print out an error if you see a data decl here.  You should either pick
     * out the stmts in advance, or ignore data decls if they come up.
     *
     * ANOTHER 330 NOTE: To simplify Assignment 4, no nested blocks will be
     * part of the test data, so you don't have to deal with them here.
     */
    public Value eval(TreeNode node, SymbolTable symtab) {

        if (isStmtList(node)) {
            evalStmtList((TreeNodeList) node, symtab);
            return null;
        }
        else if (isStmt(node)) {
            evalStmt(node, symtab);
            return null;
        }
        else if (isExpr(node)) {
            return evalExpr(node, symtab);
        }
        else {
            Utilities.error(node,
                "Node type " + symNames.map[node.id] + " is unevaluatable.");
            return null;
        }

    }

    /**
     * Evaluate a stmtlist by looping through each and call eval on it.
     */
    public void evalStmtList(TreeNodeList nodes, SymbolTable symtab) {

        TreeNode node;
        TreeNodeList rest;
        boolean done = false;

        for (node = nodes.node, rest = nodes.siblings; !done; ) {
            eval(node, symtab);
            if (rest == null) {
                done = true;
            }
            else {
                node = rest.node;
                rest = rest.siblings;
            }
        }

    }

    /**
     * Evaluate an individual Pascal statement, which is one of null,
     * assignment, if, procedure call, or compound.
     */
    public void evalStmt(TreeNode stmt, SymbolTable symtab) {
```

```java
        switch (stmt.id) {
            case 0:
                break;
            case sym.ASSMNT:
                evalAssmnt((TreeNode2) stmt, symtab);
                break;
            case sym.IF:
                evalIf((TreeNode3) stmt, symtab);
                break;
            case sym.PROCEDURE:
                evalProcCall((TreeNode2) stmt, symtab);
                break;
            case sym.BEGIN:
                eval(((TreeNode1) stmt).child, symtab);
        }
    }

    /**
     * Evaluate an assignment statement by evaluating the LHS as an l-value and
     * the RHS as an r-value expression.  If the type of the LHS and RHS are
     * the same, then assign the r-value to the memory location designated by
     * the l-value.  If the LHS does not designate a memory location, output an
     * error message.  Also output an error if the LHS and RHS types differ,
     * output an error message.
     */
    public void evalAssmnt(TreeNode2 assmntStmt, SymbolTable symtab) {

        LValue lval = evalIdentLValue((LeafNode) assmntStmt.child1, symtab);
        Value rval = evalExpr(assmntStmt.child2, symtab);

        /*
         * Outta here if the LHS does not designate a memory address.
         */
        if (lval == null) {
            Utilities.error(assmntStmt.child1,
              "The LHS of an assignment must be an assignable designator.");
            return;
        }

        /*
         * Also outta here if RHS is null, which means there was an error
         * during its evaluation.  The error message(s) for this will have
         * already occurred, so don't do any more.
         */
        if (lval == null) {
            return;
        }

        /*
         * Do the assignment if the LHS and RHS are the same type.  This is the
         * BOTTOM LINE when it comes to the evaluation of assignment.
         *
         * Have a close look here -- we're defining assignment in EJay using
         * assignment in Java.  It's taken a bit of work to get here, but
         * ultimately this is what interpretation is about.  I.e., the
         * fundamental constructs of the language being interpreted are defined
         * in terms of the fundamental constructs of the language in which the
         * interpreter is implemented.  Pretty curious, actually.
         */
        if (Types.equiv(lval.type, rval.type)) {
            memory[lval.getVal()] = rval.val;
        }
        else {
            Utilities.error(assmntStmt,
             "The type of the LHS and RHS of an assignment must be the same.");
        }

    }

    /**
     * Eval an if by first evaluating its test expr.  Then, eval the then part
     * of the statement if the test expr is true, else eval the else part if
     * present, or return if the if part is false.  If the value is not
     * boolean, or its value is null, then output and error and return without
     * evaluating either part of the statement.
     */
    public void evalIf(TreeNode3 ifStmt, SymbolTable symtab) {
        Value ifTest = evalExpr(ifStmt.child1, symtab);

        if (Types.isBool(ifTest.type)) {
            /* evaluate the then or else, as appropriate */
        }
        else {
            Utilities.error(ifStmt.child1,
              "The type of the expression in an if statement must be boolean");
        }
    }

    /**
     * Eval a proc call by performing these steps.
     *
     *   (1) looking up the procedure name in the symtab                 <br>
     *   (2) pushing an activation record onto the stack                 <br>
     *   (3) binding actuals to formals, in the callING environment      <br>
     *   (4) eval'ing the proc body, in the callED environment           <br>
     *   (5) popping the activation record and returning the value, if any <br>
     *
     * Output an error message under the following circumstances:
     *
     *   (1) the proc name lookup fails or is not that of a procedure    <br>
     *   (2) the number or types of actuals and formals do not agree     <br>
     */
    public Value evalProcCall(TreeNode2 procCall, SymbolTable symtab) {

        //
        // CSC 330 NOTE: The implementation details of this are left for your
        // work in assignment 4.  We'll discuss conceptual details in class.
        //

        return null;

    }
```

```java

    /**
     * Eval an expr by switching on its node id.  This will the id of a literal
     * value, an identifier, or an operator.  All cases are handled in the
     * appropriate methods.
     */
    public Value evalExpr(TreeNode expr, SymbolTable symtab) {
        switch (expr.id) {
            case sym.INT:
                return evalInt((LeafNode) expr);
            //
            // Other literals go here.
            //

            case sym.IDENT:
                return evalIdent((LeafNode) expr, symtab);

            case sym.PLUS:
                return evalPlus((TreeNode2) expr, symtab);

            case sym.TIMES:
                return evalTimes((TreeNode2) expr, symtab);

            //
            // Other binary and unary exprs go here.
            //

        }

        return null;    // to shut the compiler up

    }

    /**
     * Eval an addition expression by evaluating both operands, adding them
     * together, and returning a Value for the result.  Output an error if both
     * operands are not the same numeric type, and return null.  Also, return
     * null if either or both operands evaluates to null.
     */
    public Value evalPlus(TreeNode2 expr, SymbolTable symtab) {

        Value v1 = eval(expr.child1, symtab);
        Value v2 = eval(expr.child2, symtab);

        /*
         * Propagate a null return from either or both args.  No new error
         * message is necessary here.
         */
        if ((v1 == null) || (v2 == null)) {
            return null;
        }

        /*
         * Output an error if both operands aren't the same numeric type.
         */
        if (! ((v1.type.id == v2.type.id) && Types.equiv(v1.type, v2.type))) {
            Utilities.error(expr,
                "Operands of '+' must be the same numeric type.");
        }

        /*
         * Add the value of v2 to v1, and return v1.  In order to cast out in
         * Java, we have to separate the int and float cases.  Note that in
         * Java 1.5, these casts are not necessary.
         *
         * This is another BOTTOM LINE place, where some real work is getting
         * done.  Here we are implementing EJay addition directly in terms of
         * Java's addition.
         */
        if (Types.isInt(v1.type)) {
            v1.val = new Integer(((Integer)v1.val).intValue() +
                ((Integer) v2.val).intValue());
        }
        else {
            v1.val = new Float(((Float)v1.val).floatValue() +
                ((Float) v2.val).floatValue());
        }

        return v1;

    }

    /**
     * A la evalPlus.
     */
    public Value evalTimes(TreeNode2 expr, SymbolTable symtab) {

        Value v1 = eval(expr.child1, symtab);
        Value v2 = eval(expr.child2, symtab);

        if ((v1 == null) || (v2 == null)) {
            return null;
        }

        if (! ((v1.type.id == v2.type.id) && Types.equiv(v1.type, v2.type))) {
            Utilities.error(expr,
                "Operands of '*' must be the same numeric type.");
        }

        if (Types.isInt(v1.type)) {
            v1.val = new Integer(((Integer)v1.val).intValue() *
                ((Integer) v2.val).intValue());
        }
        else {
            v1.val = new Float(((Float)v1.val).floatValue() *
                ((Float) v2.val).floatValue());
        }

        return v1;

    }

```

```java
    /**
     * Evaluate a designator expression as an r-value.  In Pascal, a designator
     * is either an identifier for an array reference.  In the case of
     * identifier, this method just passes the buck to evalIdent.  The case of
     * an array ref will be dealt with in the next version of the example (Week
     * 6).
     */
    public Value evalDesig(TreeNode desig, SymbolTable symtab) {
        if (isIdent(desig)) {
            return evalIdent((LeafNode) desig, symtab);
        }

        //
        // CSC 330 NOTE: Compute the LValue for an array or struct selector.
        //

        return null;

    }

    /**
     * Eval the given ident as an r-value.  First, look it up in the given
     * symtab.  If an entry is found, get the address of the ident via getAddr.
     * Then return a new Value that contains the memory value at the computed
     * address, and the type of the ident.  If the ident lookup fails, or if
     * the entry is not a variable, then output an error message and return
     * null.
     */
    public Value evalIdent(LeafNode ident, SymbolTable symtab) {

        SymbolTableEntry entry = symtab.lookup((String) ident.value);

        if (entry == null) {
            Utilities.error(ident,
                "Identifier " + ident.value + " is not defined.");
            return null;
        }

        if (! entry.getClass().getName().equals("VariableEntry")) {
            Utilities.error(ident,
                "Identifier " + ident.value + " is not a variable.");
            return null;
        }

        int addr = getAddr((VariableEntry) entry);
        return new Value(memory[addr], entry.type);

    }


    /**
     * Eval the given designator as an l-value.  This follows the same
     * procedure as evalDesig, except here a new LValue is returned with the
     * address value, instead of the contents of memory at that address.
     */
    public LValue evalDesigLValue(TreeNode desig, SymbolTable symtab) {

        if (isIdent(desig)) {
            return evalIdentLValue((LeafNode) desig, symtab);
        }

        //
        // CSC 330 NOTE: Compute the LValue for an array or struct selector.
        //

        return null;

    }

    /**
     * Eval the given ident as an l-value.  This follows the same procedure as
     * evalIdent, except here a new LValue is returned with the address value,
     * instead of the contents of memory at that address.
     */
    public LValue evalIdentLValue(LeafNode ident, SymbolTable symtab) {

        SymbolTableEntry entry = symtab.lookup((String) ident.value);

        if (entry == null) {
            Utilities.error(ident,
                "Identifier " + ident.value + " is not defined.");
            return null;
        }

        if (! entry.getClass().getName().equals("VariableEntry")) {
            Utilities.error(ident,
                "Identifier " + ident.value + " is not a variable.");
            return null;
        }

        int addr = getAddr((VariableEntry) entry);
        return new LValue(new Integer(addr), entry.type);

    }

    /**
     * Eval an integer literal by making a new value for it.
     */
    public Value evalInt(LeafNode integer) {
        return new Value(integer.value, Types.IntType);
    }


    /**
     * Return true if the given node is an identifier.
     */
    public boolean isIdent(TreeNode node) {
        return node.id == sym.IDENT;
    }

    /**
     * Get the memory address of the given variable entry.  If the entry is at
```

```java
     * level 0, then the address is relative to the 0th element of the static
     * pool.  If the lexical level of the entry is > 0, then the address is
     * relative to the current tos.
     */
    public int getAddr(VariableEntry entry) {
        if (entry.level == 0) {
            return entry.memoryLocation;
        }
        else {
            return entry.memoryLocation + tos;
        }
    }

    /**
     * Return true if the given tree node is a statement list.
     */
    public boolean isStmtList(TreeNode node) {
        return node.getClass().getName().equals("TreeNodeList");
    }

    /**
     * Return true if the given tree node is a statement.
     */
    public boolean isStmt(TreeNode node) {
        return
            (node.id == sym.ASSMNT)
                ||
            (node.id == sym.IF)
                ||
            (node.id == sym.PROCEDURE)
                ||
            (node.id == sym.BEGIN)
                ||
            (node.id == 0);             // a null stmt
    }

    /**
     * Return true if the given tree node is an expr.
     */
    public boolean isExpr(TreeNode node) {
        return
            (node.id == sym.IDENT) ||
            (node.id == sym.REAL) ||
            (node.id == sym.INT) ||
            (node.id == sym.CHAR) ||
            (node.id == sym.LEFT_BRKT) ||
            (node.id == sym.LESS) ||
            (node.id == sym.GTR) ||
            (node.id == sym.EQ) ||
            (node.id == sym.LESS_EQ) ||
            (node.id == sym.GTR_EQ) ||
            (node.id == sym.NOT_EQ) ||
            (node.id == sym.PLUS) ||
            (node.id == sym.MINUS) ||
            (node.id == sym.OR) ||
            (node.id == sym.TIMES) ||
```

```java
            (node.id == sym.DIVIDE) ||
            (node.id == sym.AND) ||
            (node.id == sym.NOT);
    }

    /**
     * Initialize the interpreter by allocating the memory and initializing the
     * top-of-stack pointer (tos).   tos is initialized to the value of the
     * memory index after the end of the static pool, which is the value of the
     * given staticSize parameter.  The total size of memory is staticSize +
     * stackSize.  The stackSize and stackSize data fields are also
     * initialized, for subsequent runtime reference.
     */
    public void initInterpreter(int staticSize, int stackSize) {
        memory = Memory.allocate(staticSize + stackSize);
        tos = staticSize;
        curActRecSize = 0;
        this.staticSize = staticSize;
        this.stackSize = stackSize;
    }

    /**
     * Clear an already-allocated memory by setting all of its elements to
     * null.  Initialize the tos to staticSize, i.e., the stack is empty.
     */
    public void clearMemory() {
        Memory.clear(memory);
        tos = staticSize;
        curActRecSize = 0;
    }

    /**
     * Dump all of currently used memory to stdout, both static pool and
     * stack.
     */
    public void dumpMemory() {
        System.out.println();
        dumpStaticPool();
        dumpStack();
    }

    /**
     * Dump the static pool to stdout.
     */
    public void dumpStaticPool() {
        System.out.println("Static Pool:");
        Memory.dump(memory, staticSize);
        System.out.println();
    }

    /**
     * Dump the stack to stdout, starting with tos as the relative 0 address in
     * the dump.
     */
    public void dumpStack() {
        System.out.print("Stack:");
```

```
        if (tos == staticSize) {
            System.out.println(" empty");
        }
        else {
            System.out.println();
            Memory.dumpRelative(memory, staticSize + 1, tos + curActRecSize);
        }
    }


    /** Program memory, organized logically with a static pool on the top, and
     * a growable stack starting after the last allocated element of static
     * memory.  Organizing static and stack storage into a single common memory
     * pool allows uniform handling of static and stack addresses.  Viz., both
     * types of address are an integer index into the common memory.  Static
     * addresses are relative to offset 0.  Stack addresses are relative to the
     * current value of the tos index.  */
    public Object[] memory;

    /** Current top-of-stack index. */
    public int tos;

    /** The fixed size of the static memory pool, as initialized by
     * initInterpreter. */
    public int staticSize;

    /** The maximum size of the stack, as initialized by initInterpreter. */
    public int stackSize;

    /** Size of the most recently pushed activation.  This is only used as a
     * convenience for full stack dumps.
     */
    public int curActRecSize;

}
```