Oberon has a compound type POINTER:

```
VAR px, py : POINTER TO INTEGER;
BEGIN
  ...
  px := py; (* NOTE: assignable if they are the same type *)
```

Assuming that the same type method has already been written for the basic types: Write the same type method for type POINTER, using the technique of \"coding to the specification\". Please make clear:

(a) what class you put this method in  
(b) what you name the method  
(c) the types of the parameters passed to the method

Definition: Same types
Two variables $a$ and $b$ with types $T_a$ and $T_b$ are of the same type if:

4. $T_a$ and $T_b$ are POINTER TO $T_a'$ and POINTER TO $T_b'$ respectively and $T_a'$ and $T_b'$ are the same type.

You may work best from an example. The grammar for assignment is:

```
Stmt : ID T_ASSIGN Expr
```
Phase II checks

1. Constants and literals
2. *Thursday*: Procedure call
3. *Thurs/Fri*: Array and Record types

This may be a little out of order with respect to my notes, but matches the spec 1 checks.
Review: Syntax-Directed Testing

- Testing “ideally” tests all possible inputs
- Example: binary expression type checking
  - All possible binary expression programs, infinite
- Idea: syntax-directed type checking
  - “all possible combinations” of $E \ Op \ E$
  - only vary types, not identifiers, values, etc.

<table>
<thead>
<tr>
<th>Operator</th>
<th>first operand</th>
<th>second operand</th>
<th>result type</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ - *</td>
<td>numeric</td>
<td>numeric</td>
<td>smallest numeric type including both operands</td>
</tr>
<tr>
<td></td>
<td>non-numeric</td>
<td>numeric</td>
<td>“Error”</td>
</tr>
<tr>
<td></td>
<td>numeric</td>
<td>non-numeric</td>
<td></td>
</tr>
<tr>
<td></td>
<td>non-numeric</td>
<td>non-numeric</td>
<td></td>
</tr>
</tbody>
</table>
Syntax-Directed Testing

PROGRAM ArithTest
VAR x, y : INTEGER;
    r, s : REAL;
    b, c : BOOLEAN;
BEGIN
    (* assignments here only to make program syntactically correct *)
    x := x + y;  (* no error, expression is type INTEGER *)
    r := r + s;   (* no error, expression is type REAL    *)
    r := x + r;  (* no error, expression is type REAL    *)
    r := r + x;  (* no error, expression is type REAL    *)
    r := x + b;  (* error, numeric expected, got BOOLEAN *)
    r := b + x;  (* error, numeric expected, got BOOLEAN *)
    r := r + b;  (* error, numeric expected, got BOOLEAN *)
    r := b + r;  (* error, numeric expected, got BOOLEAN *)
    (* might report two errors *)
    x := b + c   (* error, numeric expected, got BOOLEAN *)
END.
Constant Expressions

- Oberon Spec says Exprs must be constant in CONST and ARRAY decls

\[
\text{CONST } S = 4;
\]
\[
\text{VAR A : ARRAY } 10*S \text{ OF REAL;}
\]

- And need the value: allocate bytes for array in C.Gen
- Here are the rules, comes down to handling of Expr:

\[
\text{ConstDecl ::= ID:id EQUAL Expr:e \{ : what add? : \}}
\]
\[
\text{ArrayType ::= ARRAY Expr:e OF Type:T \{ : what add? : \}}
\]

\[
\{ : ... \text{ if ( !e.IsConstant() ) error } ... : \}
\]
How should compiler compute isConstant()?

- Expr rule $\rightarrow$ BinaryExpr.action $\rightarrow$ BinaryExpr.elaborate

```java
boolean elaborate(ExprSTO result, STO a, OpSTO op, STO b) {
    if (opSTO.isCompatible(a, b))
        result = opSTO.resultType(a, b); …
    else {
        opSTO.reportError(a, b);
        ...
    }
}
```

```java
// No need to check if Op is OK
if (e1.isConstant() && e2.isConstant())
    result.setConstant();
    result.setValue(
        new IntLit(e1.getIntValue() +
        e2.getIntValue()));
```

Is that it?
Base Case: Handling of Literals

Expr : INT_LITERAL:litstr
{ : .; RESULT = new ConstSTO(litstr); : }

Knowledge that litstr is “INTEGER” is lost
• Model the domain! Literal of given type, not String
• Not a Constant, but a Literal expression!
{ : ...; RESULT =
    new ExprSTO(new IntLiteral(litstr)); : }
Base class Literal, with subclasses for each type of literal
• Extend ExprSTO to hold a Literal (see setValue())