Quiz: Types

A. What is the static type of:

```c
int f(int x, float y) { return x - y; }
```

B. What are the values of the static type?

C. What are the operations of this type?
Oberon Types Rules (Phase I Checks)

**Numeric types**
The INTEGER and REAL types

**Same types**
Variables $a$ and $b$ with types $T_a$ and $T_b$ are of the same type if
1. $T_a$ and $T_b$ are both denoted by the same type identifier

**Type inclusion**
Numeric types include (the values of) smaller numeric types according to the following hierarchy:  REAL $\geq$ INTEGER

**Assignment compatible**
An expression $e$ of type $T_e$ is assignment compatible with a variable $v$ of type $T_v$ if one of the following conditions hold:
1. $T_e$ and $T_v$ are the same type;
2. $T_e$ and $T_v$ are numeric types and $T_v$ includes $T_e$;
## Expression Compatibility

<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>numeric</td>
<td>numeric</td>
<td>REAL</td>
</tr>
<tr>
<td>DIV MOD</td>
<td>INTEGER</td>
<td>INTEGER</td>
<td>INTEGER</td>
</tr>
<tr>
<td>OR &amp; ~</td>
<td>BOOLEAN</td>
<td>BOOLEAN</td>
<td>BOOLEAN</td>
</tr>
<tr>
<td>= # &lt;</td>
<td>numeric</td>
<td>numeric</td>
<td>BOOLEAN</td>
</tr>
<tr>
<td>= #</td>
<td>BOOLEAN</td>
<td>BOOLEAN</td>
<td>BOOLEAN</td>
</tr>
</tbody>
</table>
So many operators, so many rules!

- Take one check at a time (not whole phase)
- Take one grammar rule at a time (E Op E)
- Take one type at a time (INTEGER)

Experiences with early pieces lends insight to later ones, making them easier.

At each step, compiler should be runnable, and hence testable and gradeable.

The following is an incremental development process
Syntax-Directed Type Checking

First, map abstract syntax in specification to concrete syntax in grammar file

Detect a *type conflict in an expression* -- that is, expressions of $X \text{ OP } Y$ where the types of either $X$ or $Y$ are incompatible with $\text{OP}$.

*Maps to*

\[
\text{Expr} ::= \text{Expr} : e_1 \text{ Op} : \text{op} \text{ Expr} : e_2 \\
\{ : \text{RESULT} = \text{BinaryExpr}.\text{action}(e_1, \text{op}, e_2); : \}
\]
Design Pattern: Singleton

- Known that there will be only one value of the class
- Methods and fields are declared static, no “new”:

```java
class BinaryExpression extends Syntax {
    static STO action(STO e1, int op, STO e2) {
        ...
        TypeValues is a singleton, too
    }
}
```

It’s common to name class in lower case (e.g., binaryExpression) so that it looks like a value (since it kind of is a singleton value)
Simplified Grammar, Oberon Semantics

P ::= D ';' S
D ::= D ';' D
D ::= ID ':' T
T ::= integer | real | boolean
E ::= ID | INTLIT | REALLIT | BOOLLIT
E ::= E '+' E
E ::= E '<' E
S ::= S ';' S
S ::= ID ':=' E
S ::= IF E THEN S
Implementing E + E check

Expr ::= Expr:e1 Op:op Expr:e2
{ RESULT = BinaryExpr.action(e1, op, e2); }

Operator | first operand       | second operand      | result type
-----------------------------------------------------------------------------------------------------
| +     * | numeric             | numeric             | smallest numeric type including both operands
|       | numeric             | numeric             | INTEGER
|       | INTEGER             | INTEGER             | INTEGER

One at a time: only INTEGER, only ‘+’
We’re “coding to the spec”

• Write the code so that it sounds like the spec
  – Terminology
  – Structure
  – Reading code out loud should sound a bit like reading the spec
  – Easier to trace back/forward between code/spec

• This is basically “top down” design/coding
  – The code you write won’t be runnable right away
  – Still have to write the code that makes this code run
ExprSTO action(ExprSTO result, expr a, String op, expr b) {
    ExprSTO result = new ExprSTO();   /* don't know type yet */
    /* also has no name; give it fake one */
    /* t1, t2, ... use static class field for counter */
    if (!BinaryExpr.elaborate(result, a, op, b))
        oberon.stopCodeGeneration();  // don't have to do this, but would for real
    return result;
}

boolean elaborate(ExprSTO result, expr a, String op, expr b) {
    if (a.type.code == Type.INT && b.type.code == Type.INT) {
        result.type = a.type;
        return true;
    }
    else {
        /* reportError automatically prints line number */
        oberon.reportError("Add operand type mismatch (%s + %s)",
            a.type.typeName(), b.type.typeName());
        /* reportError automatically prints line number */
        result.type = Type.INT;
        return false;
    }
}
BinaryExpr.action – Ints, all Ops

- “a.isInt() && b.isInt()” works for a few Ops, others?
- The general concept is “expression compatibility”, which also computes a “result type”

```
OpSTO opSTO = (OpSTO) symTab.lookup(op);
if (!BinaryExpr.elaborate(result, a, opSTO, b)) { ... }
```

```
boolean elaborate(ExprSTO result, STO a, OpSTO op, STO b) {
    if (op.compatible(a, b)) {  // a, b in ParamList later
        result.putType(op.resultType(a, b));
        return true;
    }
    else // compute error type
        ...
}
```

```
Boolean compatible(ExprSTO a, ExprSTO b) {
    return (a.isInt() && b.isInt());  // copy/pasted
} // in OpSTO (temporarily)
```

Bodies of new operations merely contain code that they replaced – better modularity and abstraction! (Still int/+), but now easily expandable to others
Design Pattern: Façade

- Layer of abstraction that provides a convenient, high-level “one stop” interface to a complex set of functionality
- In our case, **BinaryExpression** is a façade for type elaboration, checking, error reporting, and code generation