Abstract Interpretation
Simple Language (from the programming projects)

<expr> ::= const | id [only consider integer vars/consts; in the project also do float]
  | <expr> + <expr> | <expr> - <expr>
  | <expr> * <expr> | <expr> / <expr>
  | ( <expr> )

<cond> ::= true | false | <expr> < <expr> [also <=, >, >=, ==, !=]
  | <cond> && <cond> | <cond> | | <cond>
  | ! <cond> | ( <cond> )
Abstract Memory State (we will just say “Abstract State”)

**Abstract state**: a map $\sigma_a$ from $\text{vars}$ to abstract values

A summarization of many possible concrete states

$$\sigma_a : \text{Vars} \rightarrow \{ \text{Neg}, \text{Zero}, \text{Pos}, \text{AnyInt} \}$$

$\text{Vars}$ is the set of all variable names in the program

In a concrete (non-abstract) state $\sigma$ we map to $\{ 0, -1, 1, -2, 2, \ldots \}$

Here we use an abstraction of this set of concrete values

- $\sigma_a(id) = \text{Neg}$: represents all concrete states with $\sigma(id) < 0$
- $\sigma_a(id) = \text{Zero}$: represents all concrete states with $\sigma(id) = 0$
- $\sigma_a(id) = \text{Pos}$: represents all concrete states with $\sigma(id) > 0$
- $\sigma_a(id) = \text{AnyInt}$: represents all concrete states

For illustration, we will use this abstraction to prove the absence of “division by zero” errors statically, without running the program
Abstract Evaluation

Abstract evaluation relation for arithmetic expressions: triples \(<ae, \sigma_a> \rightarrow v_a\)

- \(ae\) is a parse subtree derived from <expr>
- \(\sigma_a\) is an abstract state
- \(v_a\) is an abstract value \(\in\{Neg, Zero, Pos, AnyInt\}\)

Meaning of \(<ae, \sigma_a> \rightarrow v_a\): the evaluation of \(ae\) from any concrete state \(\sigma\) abstracted by \(\sigma_a\), if it completes successfully, will produce a concrete value \(v\) abstracted by \(v_a\)

Example: \(<x+y+1, [x\mapsto Pos, y\mapsto Pos]> \rightarrow Pos\)

Example: \(<x*y-1, [x\mapsto Zero, y\mapsto Pos]> \rightarrow Neg\)
Evaluation for Arithmetic Expressions

Syntax: \texttt{id} | \texttt{const} | <expr> + <expr> | ... 

\begin{align*}
\text{\texttt{<const}, } \sigma_a & \rightarrow \text{Pos} & \text{if const.lexval is a positive integer; similarly for Zero and Neg} \\
\text{\texttt{id}, } \sigma_a & \rightarrow \sigma_a(\text{id}) & \text{static error if } \sigma_a(\text{id}) \text{ is undefined; use of uninitialized variable} \\
\text{\texttt{<ae_1}, } \sigma_a & \rightarrow v_a_1 & \text{\texttt{<ae_2}, } \sigma_a & \rightarrow v_a_2
\end{align*}

\[
v_a = v_{a_1} +_a v_{a_2}
\]

Here we use abstract addition operator \( +_a \) working on abstract values
Evaluation for Arithmetic Expressions

<table>
<thead>
<tr>
<th>$+_a$</th>
<th>Neg</th>
<th>Zero</th>
<th>Pos</th>
<th>AnyInt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Neg</td>
<td>Neg</td>
<td>Neg</td>
<td>AnyInt</td>
<td>AnyInt</td>
</tr>
<tr>
<td>Zero</td>
<td>Zero</td>
<td>Pos</td>
<td>AnyInt</td>
<td></td>
</tr>
<tr>
<td>Pos</td>
<td></td>
<td>Pos</td>
<td>AnyInt</td>
<td></td>
</tr>
<tr>
<td>AnyInt</td>
<td></td>
<td></td>
<td>AnyInt</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$-_a$</th>
<th>Neg</th>
<th>Zero</th>
<th>Pos</th>
<th>AnyInt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Neg</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pos</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AnyInt</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Let’s try this first ourselves; don’t look at next slide yet
## Evaluation for Arithmetic Expressions

<table>
<thead>
<tr>
<th>+a</th>
<th>Neg</th>
<th>Zero</th>
<th>Pos</th>
<th>AnyInt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Neg</td>
<td>Neg</td>
<td>Neg</td>
<td>AnyInt</td>
<td>AnyInt</td>
</tr>
<tr>
<td>Zero</td>
<td>Zero</td>
<td>Pos</td>
<td>AnyInt</td>
<td></td>
</tr>
<tr>
<td>Pos</td>
<td>Pos</td>
<td>AnyInt</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AnyInt</td>
<td>AnyInt</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>-a</th>
<th>Neg</th>
<th>Zero</th>
<th>Pos</th>
<th>AnyInt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Neg</td>
<td>AnyInt</td>
<td>Neg</td>
<td>Neg</td>
<td>AnyInt</td>
</tr>
<tr>
<td>Zero</td>
<td>Pos</td>
<td>Zero</td>
<td>Neg</td>
<td>AnyInt</td>
</tr>
<tr>
<td>Pos</td>
<td>Pos</td>
<td>Pos</td>
<td>AnyInt</td>
<td>AnyInt</td>
</tr>
<tr>
<td>AnyInt</td>
<td>AnyInt</td>
<td>AnyInt</td>
<td>AnyInt</td>
<td>AnyInt</td>
</tr>
</tbody>
</table>
Evaluation for Arithmetic Expressions

Let’s try this first ourselves; don’t look at next slide yet

<table>
<thead>
<tr>
<th></th>
<th>Neg</th>
<th>Zero</th>
<th>Pos</th>
<th>AnyInt</th>
</tr>
</thead>
<tbody>
<tr>
<td>*&lt;sub&gt;a&lt;/sub&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Neg</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pos</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AnyInt</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Neg</th>
<th>Zero</th>
<th>Pos</th>
<th>AnyInt</th>
</tr>
</thead>
<tbody>
<tr>
<td>/&lt;sub&gt;a&lt;/sub&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Neg</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pos</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AnyInt</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Semantics of concrete /: treat as reals, then round toward zero
### Evaluation for Arithmetic Expressions

<table>
<thead>
<tr>
<th>* _a</th>
<th>Neg</th>
<th>Zero</th>
<th>Pos</th>
<th>AnyInt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Neg</td>
<td>Pos</td>
<td>Zero</td>
<td>Neg</td>
<td>AnyInt</td>
</tr>
<tr>
<td>Zero</td>
<td>Zero</td>
<td>Zero</td>
<td>Zero</td>
<td></td>
</tr>
<tr>
<td>Pos</td>
<td></td>
<td>Pos</td>
<td>AnyInt</td>
<td></td>
</tr>
<tr>
<td>AnyInt</td>
<td></td>
<td></td>
<td>AnyInt</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>I _a</th>
<th>Neg</th>
<th>Zero</th>
<th>Pos</th>
<th>AnyInt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Neg</td>
<td>AnyInt</td>
<td>![Warning]</td>
<td>AnyInt</td>
<td>![Warning]</td>
</tr>
<tr>
<td>Zero</td>
<td>Zero</td>
<td>![Warning]</td>
<td>Zero</td>
<td>![Warning]</td>
</tr>
<tr>
<td>Pos</td>
<td>AnyInt</td>
<td>![Warning]</td>
<td>AnyInt</td>
<td>![Warning]</td>
</tr>
<tr>
<td>AnyInt</td>
<td>AnyInt</td>
<td>![Warning]</td>
<td>AnyInt</td>
<td>![Warning]</td>
</tr>
</tbody>
</table>

**Example:**
- Pos \( I_a \) Pos
  - \( 5 / 3 = 1 \)
  - \( 2 / 3 = 0 \)

To represent both outcomes we use AnyInt.

⚠️ Abstract operation is undefined: cannot guarantee the absence of run-time division-by-zero error.
# Integers vs Floats

## First Operand

<table>
<thead>
<tr>
<th>$I_a$</th>
<th>Neg</th>
<th>Zero</th>
<th>Pos</th>
<th>AnyInt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Neg</td>
<td>AnyInt</td>
<td>![⚠️]</td>
<td>AnyInt</td>
<td>![⚠️]</td>
</tr>
<tr>
<td>Zero</td>
<td>Zero</td>
<td>![⚠️]</td>
<td>Zero</td>
<td>![⚠️]</td>
</tr>
<tr>
<td>Pos</td>
<td>AnyInt</td>
<td>![⚠️]</td>
<td>AnyInt</td>
<td>![⚠️]</td>
</tr>
<tr>
<td>AnyInt</td>
<td>AnyInt</td>
<td>![⚠️]</td>
<td>AnyInt</td>
<td>![⚠️]</td>
</tr>
</tbody>
</table>

## Second Operand

<table>
<thead>
<tr>
<th>$I_a$</th>
<th>Neg</th>
<th>Zero</th>
<th>Pos</th>
<th>AnyFloat</th>
</tr>
</thead>
<tbody>
<tr>
<td>Neg</td>
<td>Pos</td>
<td>![⚠️]</td>
<td>Neg</td>
<td>![⚠️]</td>
</tr>
<tr>
<td>Zero</td>
<td>Zero</td>
<td>![⚠️]</td>
<td>Zero</td>
<td>![⚠️]</td>
</tr>
<tr>
<td>Pos</td>
<td>Neg</td>
<td>![⚠️]</td>
<td>Pos</td>
<td>![⚠️]</td>
</tr>
<tr>
<td>AnyFloat</td>
<td>AnyFloat</td>
<td>![⚠️]</td>
<td>AnyFloat</td>
<td>![⚠️]</td>
</tr>
</tbody>
</table>
Division Example

```java
int x = 3;  // [x→Pos]
int z = -x; // [x→Pos, z→Neg]
int w = x - z + 5; // [x→Pos, z→Neg, w→Pos]
w = w / x;  // [x→Pos, z→Neg, w→AnyInt]
x = x / w;  // static checking error: w may be 0 [but not really...]
```

We could choose to be less conservative: only complain if we are sure that the second operand is zero

<table>
<thead>
<tr>
<th>first operand</th>
<th>Neg</th>
<th>Zero</th>
<th>Pos</th>
<th>AnyInt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Neg</td>
<td>AnyInt</td>
<td>![Warning]</td>
<td>AnyInt</td>
<td>AnyInt</td>
</tr>
<tr>
<td>Zero</td>
<td>Zero</td>
<td>![Warning]</td>
<td>Zero</td>
<td>Zero</td>
</tr>
<tr>
<td>Pos</td>
<td>AnyInt</td>
<td>![Warning]</td>
<td>AnyInt</td>
<td>AnyInt</td>
</tr>
<tr>
<td>AnyInt</td>
<td>AnyInt</td>
<td>![Warning]</td>
<td>AnyInt</td>
<td>AnyInt</td>
</tr>
</tbody>
</table>
## Integers vs Floats: Less Conservative

<table>
<thead>
<tr>
<th>$I_a$</th>
<th>Neg</th>
<th>Zero</th>
<th>Pos</th>
<th>AnyInt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Neg</td>
<td>AnyInt</td>
<td>![caution]</td>
<td>AnyInt</td>
<td>AnyInt</td>
</tr>
<tr>
<td>Zero</td>
<td>Zero</td>
<td>![caution]</td>
<td>Zero</td>
<td>Zero</td>
</tr>
<tr>
<td>Pos</td>
<td>AnyInt</td>
<td>![caution]</td>
<td>AnyInt</td>
<td>AnyInt</td>
</tr>
<tr>
<td>AnyInt</td>
<td>AnyInt</td>
<td>![caution]</td>
<td>AnyInt</td>
<td>AnyInt</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$I_a$</th>
<th>Neg</th>
<th>Zero</th>
<th>Pos</th>
<th>AnyFloat</th>
</tr>
</thead>
<tbody>
<tr>
<td>Neg</td>
<td>Pos</td>
<td>![caution]</td>
<td>Neg</td>
<td>AnyFloat</td>
</tr>
<tr>
<td>Zero</td>
<td>Zero</td>
<td>![caution]</td>
<td>Zero</td>
<td>Zero</td>
</tr>
<tr>
<td>Pos</td>
<td>Neg</td>
<td>![caution]</td>
<td>Pos</td>
<td>AnyFloat</td>
</tr>
<tr>
<td>AnyFloat</td>
<td>AnyFloat</td>
<td>![caution]</td>
<td>AnyFloat</td>
<td>AnyFloat</td>
</tr>
</tbody>
</table>
Trade-Offs in Algorithm Design

**More conservative** version: if it *does not* report an error, we are guaranteed that every execution will *not* have div-by-zero error

**Less conservative** version: if it *does* report an error, we are guaranteed that every execution will *will* have an div-by-zero error

– This will avoid false warnings, but will also miss some programs with run-time div-by-zero errors

This is an example of a typical trade-off in the design of static checking algorithms
Evaluation for Boolean Expressions

\(<\text{cond}> ::= \text{true} \mid \text{false} \mid <\text{expr}> < <\text{expr}> \quad [\text{also } <=, >, >=, ==, !=] \mid <\text{cond}> && <\text{cond}> \mid <\text{cond}> || <\text{cond}> \mid ! <\text{cond}> \mid ( <\text{cond}> )\)

Concrete: \(<\text{be}, \sigma> \rightarrow v\)

\(v\) is a value from \(\{\text{true}, \text{false}\}\)

Abstract: \(<\text{be}, \sigma_a> \rightarrow \text{AnyBool}\)

For now, keep it simple: statically, assume that at run time, both \text{true} and \text{false} are possible. Do not look inside these expressions and do not check. We will revisit this later.
Statements: \( <s, \sigma_a> \rightarrow \sigma'_a \)

\[
<\text{skip}, \sigma_a> \rightarrow \sigma_a
\]

\[
<\text{ae}, \sigma_a> \rightarrow \nu_a
\]

\[
<\text{id}=\text{ae}, \sigma_a> \rightarrow \sigma_a [\text{id} \mapsto \nu_a]
\]

\[
<s_1, \sigma_a> \rightarrow \sigma_{a1} \quad <s_2, \sigma_a> \rightarrow \sigma_{a2}
\]

\[
<\text{if (be) } s_1 \text{ else } s_2, \sigma_a> \rightarrow \sigma'_a
\]

\[
\sigma'_a = \text{merge}(\sigma_{a1}, \sigma_{a2})
\]

\[
<s_1, \sigma_a> \rightarrow \sigma_{a1}
\]

\[
<\text{if (be) } s_1, \sigma_a> \rightarrow \sigma'_a
\]

\[
\sigma'_a = \text{merge}(\sigma_a, \sigma_{a1})
\]
Merging of Abstract States

Do it variable-by-variable:
1) If the variable is defined in both abstract states: use this table

<table>
<thead>
<tr>
<th>merge</th>
<th>Neg</th>
<th>Zero</th>
<th>Pos</th>
<th>AnyInt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Neg</td>
<td>Neg</td>
<td>AnyInt</td>
<td>AnyInt</td>
<td>AnyInt</td>
</tr>
<tr>
<td>Zero</td>
<td>Zero</td>
<td>AnyInt</td>
<td>AnyInt</td>
<td>AnyInt</td>
</tr>
<tr>
<td>Pos</td>
<td></td>
<td></td>
<td>Pos</td>
<td>AnyInt</td>
</tr>
<tr>
<td>AnyInt</td>
<td></td>
<td></td>
<td></td>
<td>AnyInt</td>
</tr>
</tbody>
</table>

2) If the variable is defined in only one abstract state: undefined in the merged state [this will allow us to catch uninitialized variables; details later]

3) If the variable is undefined in both abstract states: remains undefined in the merged state
Example of Merging

<table>
<thead>
<tr>
<th>merge</th>
<th>Neg</th>
<th>Zero</th>
<th>Pos</th>
<th>AnyInt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Neg</td>
<td>Neg</td>
<td>AnyInt</td>
<td>AnyInt</td>
<td>AnyInt</td>
</tr>
<tr>
<td>Zero</td>
<td>Zero</td>
<td>AnyInt</td>
<td>AnyInt</td>
<td>AnyInt</td>
</tr>
<tr>
<td>Pos</td>
<td></td>
<td>Pos</td>
<td>AnyInt</td>
<td>AnyInt</td>
</tr>
<tr>
<td>AnyInt</td>
<td></td>
<td></td>
<td></td>
<td>AnyInt</td>
</tr>
</tbody>
</table>

resulting state:

```plaintext
x = 1; [x↦Pos]
y = -2;  [x↦Pos, y↦Neg]
if (...)
   z = x+1; [x↦Pos, y↦Neg, z↦Pos]
else
   z = x-y; [x↦Pos, y↦Neg, z↦Pos]
```
Example of Merging

<table>
<thead>
<tr>
<th>merge</th>
<th>Neg</th>
<th>Zero</th>
<th>Pos</th>
<th>AnyInt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Neg</td>
<td>Neg</td>
<td>AnyInt</td>
<td>AnyInt</td>
<td>AnyInt</td>
</tr>
<tr>
<td>Zero</td>
<td></td>
<td>Zero</td>
<td>AnyInt</td>
<td>AnyInt</td>
</tr>
<tr>
<td>Pos</td>
<td></td>
<td></td>
<td>Pos</td>
<td>AnyInt</td>
</tr>
<tr>
<td>AnyInt</td>
<td></td>
<td></td>
<td></td>
<td>AnyInt</td>
</tr>
</tbody>
</table>

resulting state:

\[
\begin{align*}
x &= 1; \\
y &= -2; \\
\text{if} \ (\ldots) \\
\quad z &= x+1; \\
\text{else} \\
\quad z &= x+y;
\end{align*}
\]

\[
\begin{align*}
[x\mapsto Pos] \\
[x\mapsto Pos, \ y\mapsto Neg] \\
[x\mapsto Pos, \ y\mapsto Neg, \ z\mapsto Pos] \\
[x\mapsto Pos, \ y\mapsto Neg, \ z\mapsto AnyInt] \\
[x\mapsto Pos, \ y\mapsto Neg, \ z\mapsto AnyInt]
\end{align*}
\]
Loops: $< \text{while (be) s, } \sigma_a \text{ } \rightarrow \text{ } \sigma'_a$

We abstract the loop condition as “don’t know; could be true or false”; need to consider all possible executions of the loop

0 iterations: $\sigma'_a = \sigma_a$

1 iteration: if $<s, \sigma_a> \rightarrow \sigma_{a1}$, then $\sigma'_a = \sigma_{a1}$

2 iterations: if $<s, \sigma_{a1}> \rightarrow \sigma_{a2}$, then $\sigma'_a = \sigma_{a2}$ and so on

$\sigma'_a = \text{merge}(\sigma_a, \sigma_{a1}, \sigma_{a2}, \sigma_{a3}, ...)$: infinite number of $\sigma_{ak}$

But: $\sigma'_a$ can be computed in a finite number of steps

$\sigma'_{a0} = \sigma_a$

$\sigma'_{a1} = \text{merge}(\sigma'_{a0}, \sigma_{a1})$

$\sigma'_{a2} = \text{merge}(\sigma'_{a1}, \sigma_{a2})$

$\sigma'_{a3} = \text{merge}(\sigma'_{a2}, \sigma_{a3})$ and so on

This converges: after a while we have $\sigma'_{a(k+1)} = \sigma'_{ak}$ [details in CSE 5343]
Interpreter for the Abstract Semantics

If we implement an interpreter, we get a static checker for division-by-zero errors and use-before-initialization errors.

**Code implementation** (e.g., for the programming projects)

AbstractValue abs_eval(TreeNode n, AbstractState as) { ...
    if (n is a plus expression) return
        abs_plus(abs_eval(left subexpr, as), abs_eval(right subexpr, as));
}

or, in a more object-oriented style

class BinaryExpr {
    AbstractValue abs_eval(AbstractState as) { ...
        if (this is a plus expression)
            return abs_plus(expr1.abs_eval(as), expr2.abs_eval(as));
    }
}
Interpreter for the Abstract Semantics

Code implementation for if-then-else
abs_exec(TreeNode n, AbstractState as) {
    AbstractState as2 = as.copy();
    abs_exec(then part, as); // updates as
    abs_exec(else part, as2); // updates as2
    abs_merge(as, as2); // merge as2 into as; changes as
}

Code implementation for while loop
AbstractState abs_exec(TreeNode n, AbstractState as) {
    // in a loop, abs_exec(body, current state) and
    // merge the current state into the result state. stop after
    // convergence is seen and then return the result state
}
Project 4

Goal: modify Project 3 to do checking for possible division by zero and use of uninitialized vars [but not inside conditional expressions; will do in Project 5]

Code changes will be minimal: if you have code to do real interpretation, it is not hard to change it to do abstract interpretation

– State now contains abstract values
– Expressions are evaluated using the abstract operators
– If-then, if-then-else, and while-loops are changed as described in the last few slides

Checking: second operand is Zero, AnyInt, AnyFloat
Uninitialized Variables

Example 1:

```c
int x; int y = x;
```
when we try abs_eval(x) we will not find x in state

Example 2:

```c
int x; if (...) { x = 1; } else { x = -2; } int y = x;
```
state after true branch \([x \mapsto \text{Pos}]\), state after false branch \([x \mapsto \text{Neg}]\), state after merge \([x \mapsto \text{AnyInt}]\), checking is fine for \(\text{int y = x;}\)

Example 3:

```c
int x; int z = 2; if (...) { x = 1; } else { ... } int y = x;
```
state after true branch \([x \mapsto \text{Pos}, z \mapsto \text{Pos}]\), state after false branch \([z \mapsto \text{Pos}]\), state after merge \([z \mapsto \text{Pos}]\), error for \(\text{int y = x;}\)

Todo: at home, try an example with a while-loop
Project 5: Boolean Expressions Refined

\[ <\text{cond}> ::= \text{true} | \text{false} | <\text{expr}> < <\text{expr}> \quad [\text{also } \leq, >, \geq, ==, !=] \]

\[ | <\text{cond}> \&\& <\text{cond}> | <\text{cond}> | <\text{cond}> \]

\[ | ! <\text{cond}> | ( <\text{cond}> ) \]

Abstract: \( \langle \text{be}, \sigma_a \rangle \rightarrow v_a \) where \( v_a \in \{ \text{True}, \text{False}, \text{AnyBool} \} \)

<table>
<thead>
<tr>
<th>&amp;&amp;_a</th>
<th>True</th>
<th>False</th>
<th>AnyBool</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>True</td>
<td>False</td>
<td>AnyBool</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>False</td>
<td></td>
</tr>
<tr>
<td>AnyBool</td>
<td>AnyBool</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Similarly for || and !
Comparisons

\[ \text{\textless cond\textgreater} ::= \ldots \mid \text{\textless expr\textgreater} \text{\textless} \text{\textless expr\textgreater} \] [also \textless=, \textgreater, \textless=, \textgreater=, !=]

<table>
<thead>
<tr>
<th>\textless_a \</th>
<th>Neg</th>
<th>Zero</th>
<th>Pos</th>
<th>AnyInt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Neg</td>
<td>AnyBool</td>
<td>True</td>
<td>True</td>
<td>AnyBool</td>
</tr>
<tr>
<td>Zero</td>
<td>False</td>
<td>False</td>
<td>True</td>
<td>AnyBool</td>
</tr>
<tr>
<td>Pos</td>
<td>False</td>
<td>False</td>
<td>AnyBool</td>
<td>AnyBool</td>
</tr>
<tr>
<td>AnyInt</td>
<td>AnyBool</td>
<td>AnyBool</td>
<td>AnyBool</td>
<td>AnyBool</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>=_a</th>
<th>Neg</th>
<th>Zero</th>
<th>Pos</th>
<th>AnyInt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Neg</td>
<td>AnyBool</td>
<td>False</td>
<td>False</td>
<td>AnyBool</td>
</tr>
<tr>
<td>Zero</td>
<td>True</td>
<td>False</td>
<td>AnyBool</td>
<td></td>
</tr>
<tr>
<td>Pos</td>
<td>AnyBool</td>
<td>AnyBool</td>
<td>AnyBool</td>
<td></td>
</tr>
<tr>
<td>AnyInt</td>
<td>AnyBool</td>
<td>AnyBool</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>


Code implementation for if-then-else
abs_exec(TreeNode n, AbstractState as) {
    AbstractBool cond = abs_eval(condition, as);
    // case 1: statically guaranteed to be true; else part is dead code.
    if (cond == True) { abs_exec(then part, as); print warning; return; }
    // case 2: statically guaranteed to be false; then part is dead code
    if (cond == False) { abs_exec(else part, as); print warning; return; }
    // case 3: don’t know statically because cond == AnyBool
    AbstractState as2 = as.copy();
    abs_exec(then part, as); // updates as
    abs_exec(else part, as2); // updates as2
    abs_merge(as, as2); // merge as2 into as; changes as
}
Example

<cond> ::= ... | <expr> < <expr>  [also <=, >, >=, ==, !=]

<table>
<thead>
<tr>
<th></th>
<th>Neg</th>
<th>Zero</th>
<th>Pos</th>
<th>AnyInt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Neg</td>
<td>AnyBool</td>
<td>True</td>
<td>True</td>
<td>AnyBool</td>
</tr>
<tr>
<td>Zero</td>
<td>False</td>
<td>False</td>
<td>True</td>
<td>AnyBool</td>
</tr>
<tr>
<td>Pos</td>
<td>False</td>
<td>False</td>
<td>AnyBool</td>
<td>AnyBool</td>
</tr>
<tr>
<td>AnyInt</td>
<td>AnyBool</td>
<td>AnyBool</td>
<td>AnyBool</td>
<td>AnyInt</td>
</tr>
</tbody>
</table>

earlier approach: this approach:

x = 1;  [x↦Pos]
y = -2;  [x↦Pos, y↦Neg]
if (y < x)
  z = x+1;  [x↦Pos, y↦Neg, z↦Pos]
else
  z = y;  [x↦Pos, y↦Neg, z↦Neg]
  ignore the else branch
  [x↦Pos, y↦Neg, z↦AnyInt]
  [x↦Pos, y↦Neg, z↦Pos]
Uninitialized Variables

Example 1:
```c
int x; int y = x;
```
when we try abs_eval(x) we will not find x in state

Example 2:
```c
int x = readint; int z; if (x > 0) { ... } else { z = 1; } int y = z;
```
error reported for uninitialized z

Example 3:
```c
int p = 1; int q = 2; int z; if (p<q) { z = 3; } else { ... } int y = z;
```
error reported for uninitialized z, but at run time there is no error

Example 4:
```c
int p = readint; int q = 1; int z; while (q<p) { q=q+1; r = q*q; } int y = q;
```
Uninitialized Variables in Java

Example 1: `int x; int y = x;`
**Result:** both our checker and javac complain

Example 2:

```java
int x = readInt; int z;
if (x > 0) { x = 0; } else { z = 1; }
int y = z;
vs.
int x = (new Scanner(System.in)).nextInt();
int z;
if (x > 0) { x = 0; } else { z = 1; }
int y = z;
```
**Result:** both our checker and javac complain
Uninitialized Variables in Java

Example 3:
```java
int x = 1; int y = 2; int z = x + y;
int w;
if (z > 0) { w = 3; }
int v = w;
```

**Result:** javac incorrectly complains; our checker correctly accepts

Example 4: can add “final” (write-once) in the Java code
```java
final int x = 1; final int y = 2; final int z = x + y;
int w;
if (z > 0) { w = 3; }
int v = w;
```

**Result:** javac correctly accepts; our checker correctly accepts
Uninitialized Variables in Java

Example 5:
```java
final int x = 1; final int y = 2; int z = x + y; int w;
if (z > 0) { w = 3; }
int v = w;
int p; z = -z;
if (z > 0) { p = 4; }
int q = p;
```

Result: javac complains about \texttt{w} and \texttt{p}; our checker complains about \texttt{p}

Example 6:
```java
final int x = 1; final int y = 2; int w;
if (x < y) { w = 3; }
int v = w;
```

Result: javac correctly accepts; our checker incorrectly complains