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 vars to abstract values
A summarization of many possible concrete states

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

$\text{Vars}$ 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*}
<\texttt{const}, \sigma_a> & \rightarrow Pos & \text{if const.lexval is a positive integer; similarly for Zero and Neg} \\
<\texttt{id}, \sigma_a> & \rightarrow \sigma_a(id) & \text{static error if } \sigma_a(id) \text{ is undefined; use of uninitialized variable} \\
<\texttt{ae}_1, \sigma_a> & \rightarrow v_{a1} & <\texttt{ae}_2, \sigma_a> \rightarrow v_{a2} \\
\underline{<\texttt{ae}_1+\texttt{ae}_2, \sigma_a> \rightarrow v_a} & \quad & v_a = v_{a1} +_a v_{a2}
\end{align*}

Here we use abstract addition operator $+_a$ working on abstract values
### Evaluation for Arithmetic Expressions

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

#### second operand

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

**first operand**

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></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>* (_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>

Semantics of concrete `/`: treat as reals, then round toward zero

<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>
Evaluation for Arithmetic Expressions

<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>Neg</td>
<td>Zero</td>
<td>Pos</td>
<td>AnyInt</td>
</tr>
<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>Pos</td>
<td>AnyInt</td>
<td>AnyInt</td>
<td>AnyInt</td>
</tr>
<tr>
<td>AnyInt</td>
<td>AnyInt</td>
<td></td>
<td>AnyInt</td>
<td>AnyInt</td>
</tr>
</tbody>
</table>

First operand

<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>Neg</td>
<td>Zero</td>
<td>Pos</td>
<td>AnyInt</td>
</tr>
<tr>
<td>Neg</td>
<td>AnyInt</td>
<td></td>
<td>AnyInt</td>
<td>AnyInt</td>
</tr>
<tr>
<td>Zero</td>
<td>Zero</td>
<td></td>
<td>Zero</td>
<td>AnyInt</td>
</tr>
<tr>
<td>Pos</td>
<td>AnyInt</td>
<td></td>
<td>AnyInt</td>
<td>AnyInt</td>
</tr>
<tr>
<td>AnyInt</td>
<td>AnyInt</td>
<td></td>
<td>AnyInt</td>
<td>AnyInt</td>
</tr>
</tbody>
</table>

Abstract operation is undefined: cannot guarantee the absence of run-time division-by-zero error

Example:
Pos /<sub>a</sub> Pos
5 / 3 = 1
2 / 3 = 0

To represent both outcomes we use AnyInt
## 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>![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>

### 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>![Warning]</td>
<td>Neg</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>Neg</td>
<td>![Warning]</td>
<td>Pos</td>
<td>![Warning]</td>
</tr>
<tr>
<td>AnyFloat</td>
<td>AnyFloat</td>
<td>![Warning]</td>
<td>AnyFloat</td>
<td>![Warning]</td>
</tr>
</tbody>
</table>
Division Example

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

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

<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>![error]</td>
<td>AnyInt</td>
<td>AnyInt</td>
</tr>
<tr>
<td>Zero</td>
<td>Zero</td>
<td>![error]</td>
<td>Zero</td>
<td>Zero</td>
</tr>
<tr>
<td>Pos</td>
<td>AnyInt</td>
<td>![error]</td>
<td>AnyInt</td>
<td>AnyInt</td>
</tr>
<tr>
<td>AnyInt</td>
<td>AnyInt</td>
<td>![error]</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>![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>

<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>![Warning]</td>
<td>Neg</td>
<td>AnyFloat</td>
</tr>
<tr>
<td>Zero</td>
<td>Zero</td>
<td>![Warning]</td>
<td>Zero</td>
<td>Zero</td>
</tr>
<tr>
<td>Pos</td>
<td>Neg</td>
<td>![Warning]</td>
<td>Pos</td>
<td>AnyFloat</td>
</tr>
<tr>
<td>AnyFloat</td>
<td>AnyFloat</td>
<td>![Warning]</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 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

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

\[ \mid \langle \text{cond} \rangle \&\& \langle \text{cond} \rangle \mid \langle \text{cond} \rangle | | \langle \text{cond} \rangle \]

\[ \mid ! \langle \text{cond} \rangle \mid ( \langle \text{cond} \rangle ) \]

Concrete: \(\langle \text{be}, \sigma \rangle \rightarrow v\)

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

Abstract: \(\langle \text{be}, \sigma_a \rangle \rightarrow \text{AnyBool}\)

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

$<\text{skip}, \sigma_a> \rightarrow \sigma_a$

$<\text{ae}, \sigma_a> \rightarrow \nu_a$

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

$<\text{s}_1, \sigma_a> \rightarrow \sigma_{a1}$  $<\text{s}_2, \sigma_a> \rightarrow \sigma_{a2}$

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

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

$<\text{s}_1, \sigma_a> \rightarrow \sigma_{a1}$

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

$\sigma'_a = \text{merge}(\sigma_a, \sigma_{a1})$
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>AnyInt</td>
<td>Zero</td>
<td>AnyInt</td>
<td>AnyInt</td>
</tr>
<tr>
<td>Pos</td>
<td>AnyInt</td>
<td>AnyInt</td>
<td>Pos</td>
<td>AnyInt</td>
</tr>
<tr>
<td>AnyInt</td>
<td>AnyInt</td>
<td>AnyInt</td>
<td>AnyInt</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>AnyInt</td>
<td>Pos</td>
<td>AnyInt</td>
</tr>
<tr>
<td>AnyInt</td>
<td></td>
<td></td>
<td>AnyInt</td>
<td>AnyInt</td>
</tr>
</tbody>
</table>

resulting state:

\[
x = 1; \quad \text{[}x\mapsto\text{Pos}\text{]}
\]
\[
y = -2; \quad \text{[}x\mapsto\text{Pos}, y\mapsto\text{Neg}\text{]}
\]
\[
\text{if (...) }
\]
\[
z = x+1; \quad \text{[}x\mapsto\text{Pos}, y\mapsto\text{Neg}, z\mapsto\text{Pos}\text{]}
\]
\[
\text{else}
\]
\[
z = x-y; \quad \text{[}x\mapsto\text{Pos}, y\mapsto\text{Neg}, z\mapsto\text{Pos}\text{]}
\]
\[
\text{[}x\mapsto\text{Pos}, y\mapsto\text{Neg}, z\mapsto\text{Pos}\text{]}
\]

17
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:

x = 1;
[y↦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↦AnyInt]
    [x↦Pos, y↦Neg, z↦AnyInt]
Loops: < while (be) s, σ_a >→ σ'_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: σ'_a = σ_a

1 iteration: if <s, σ_a> → σ_a1, then σ'_a = σ_a1

2 iterations: if <s, σ_a1> → σ_a2, then σ'_a = σ_a2 and so on

σ'_a = merge(σ_a, σ_a1, σ_a2, σ_a3, ...): infinite number of σ_ak

But: σ'_a can be computed in a finite number of steps

σ'_a0 = σ_a

σ'_a1 = merge(σ'_a0, σ_a1)

σ'_a2 = merge(σ'_a1, σ_a2)

σ'_a3 = merge(σ'_a2, σ_a3) and so on

This converges: after a while we have σ'_ak = σ'_a(k-1) [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 s) { ...
    if (n is a plus expression) return
        abs_plus(abs_eval(left subexpr, s), abs_eval(right subexpr, s));
}

or, in a more object-oriented style

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

Code implementation for if-then-else

```java
abs_exec(TreeNode n, AbstractState s) {
    AbstractState s2 = s.copy(); // create a new table and copy all data
    abs_exec(then part, s); // updates s
    abs_exec(else part, s2); // updates s2
    abs_merge(s, s2); // merge s2 into s; changes s
}
```

Code implementation for while loop

```java
AbstractState abs_exec(TreeNode n, AbstractState as) {
    // in a loop, abs_exec(body, current state) and
    // merge the current state $\sigma_{ak}$ into the result state $\sigma'_{ak}$. stop after
    // convergence is seen with $\sigma'_{ak} = \sigma'_{a(k-1)}$ and then return $\sigma'_{ak}$
}
```
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
Project 4

Implementation detail: Integers vs Floats

Use more refined versions of the abstract values: set
\{ NegInt, ZeroInt, PosInt, AnyInt, NegFloat, ZeroFloat, PosFloat, AnyFloat \}

– Easier to handle division (has different semantics for ints vs floats)
– Printing for testing/debugging/grading: print one of those six strings

Printing:
– For statement \texttt{print expr;} abstractly evaluate the expression and then println its abstract value
– Do not print the program
– Do not print the abstract state
Check 1: division by zero – report error if the second operand of division is ZeroInt, AnyInt, ZeroFloat, AnyFloat
[This is the “more conservative” approach from earlier; could result in false warnings a.k.a. false positives]

Check 2: use of uninitialized variable – error if a variable is used in an expression but there is no value for the variable in the abstract state
Uninitialized Variables

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

Example 2:
```
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 int y = x;

Example 3:
```
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 int y = x;

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

<cond> ::= true | false | <expr> < <expr> [also <=, >, >=, ==, !=] 
| <cond> && <cond> | <cond> || <cond> 
| ! <cond> | ( <cond> )

Abstract: <be, σ_a> → v_a where v_a ∈ { True, False, 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>False</td>
</tr>
<tr>
<td>AnyBool</td>
<td></td>
<td></td>
<td>AnyBool</td>
</tr>
</tbody>
</table>

Similarly for || and !
Short-Circuit Evaluation

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

Our abstract evaluation should “simulate” what happens in concrete evaluations. For example, consider `&&`

**Case 1**: first operand evaluates to `True` [i.e., in all concrete executions, the first operand will evaluate to true and the second operand will definitely be evaluated]. So, in Project 5, evaluate the second operand and use its value as the result of `&&`

**Case 2**: first operand evaluates to `False` [i.e., in all concrete executions, the first operand will evaluate to false and the second operand will definitely not be evaluated]. So, in Project 5, do not evaluate the second operand and just produce `False`
Short-Circuit Evaluation

Case 3: first operand evaluates to AnyBool [i.e., in some concrete executions, the first operand could possibly evaluate to true and in those cases the second operand will be evaluated]. So, in Project 5, evaluate abstractly the second operand and then produce $\text{AnyBool} \&\& a$ that value

Operator $| |$: do something similar, but suitable for OR
In reality, will have comparisons for \{ \texttt{NegInt}, \texttt{ZeroInt}, \texttt{PosInt}, \texttt{AnyInt} \} and separately for \{ \texttt{NegFloat}, \texttt{ZeroFloat}, \texttt{PosFloat}, \texttt{AnyFloat} \}, since we assume that the input program successfully passed typechecking.
If-Then-Else with Dead Code Errors

Code implementation for if-then-else

```
abs_exec(TreeNode n, AbstractState s) {
    AbstractBool cond = abs_eval(condition, s);
    // case 1: statically guaranteed to be true; else part is dead code
    if (cond == True) { terminate with static error (dead code) }
    // case 2: statically guaranteed to be false; then part is dead code
    if (cond == False) { terminate with static error (dead code) }
    // case 3: do not know statically because cond == AnyBool
    AbstractState s2 = s.copy();
    abs_exec(then part, s); // updates s
    abs_exec(else part, s2); // updates s2
    abs_merge(s, s2); // merge s2 into s; changes s
}
```
If-Then with Dead Code Error

Code implementation for if-then
abs_exec(TreeNode n, AbstractState s) {
    AbstractBool cond = abs_eval(condition, s);
    // case 1: statically guaranteed to be false; then part is dead code
    if (cond == False) { terminate with static error (dead code) }
    // case 2: statically guaranteed to be true; no merging needed
    if (cond == True) { abs_exec(then part, s); return; }
    // case 3: do not know statically because cond == AnyBool
    AbstractState s2 = s.copy();
    abs_exec(then part, s2); // updates s2
    abs_merge(s, s2); // merge s2 into s; changes s
}
While-Do with Dead Code Error

Code implementation for while-do loop

```java
abs_exec(TreeNode n, AbstractState s) {
    AbstractBool cond = abs_eval(condition, s);
    // case 1: statically guaranteed to be false; loop body is dead code
    if (cond == False) { terminate with static error (dead code) }
    // case 2: statically guaranteed to be true; at least one iteration
    if (cond == True) { abs_exec(loop body, s); }
    // general case: process exactly how you did in Project 4;
    // do not re-evaluate the loop condition
    ...
}
```
Why Case 2?

Example:

```c
int x; int y = 0;
while (y < 100) { x = y; y = y+1; }
print x;
```

Project 3: successful completion, x is 99
Project 4: x is uninitialized in the abstract state immediately before the loop; the analysis thinks that there could be zero iterations of the loop and complains about uninitialized x at the print
Project 5: since \textit{ZeroInt} \leq_a \textit{PosInt}, case 2 applies. The state is changed to include an initial value for x. Then the loop is processed as usual, starting from that modified state. The final value for x is \textit{AnyInt}.
Project 5 without case 2: same as Project 4.
Food for thought: what would happen if we had \textbf{int} \textit{y} = 1; instead?
Uninitialized Variables

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

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

Example 3:
```
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:
```
int p = readint; int q = 1; int r; while(q<p) { q = q+1; r = q*q; } int y = r;
```
Uninitialized Variables in Java

Example 1: \texttt{int \ x; int \ y = x;}

\textbf{Result:} both our checker and javac complain

Example 2:

\texttt{int \ x = readint; int \ z;}
\texttt{if \ (x > 0) \{ \ x = 0; \} \ else \{ \ z = 1; \}}
\texttt{int \ y = z;}

\textbf{vs.}

\texttt{int \ x = (new \ Scanner(System.in)).nextInt();}
\texttt{int \ z;}
\texttt{if \ (x > 0) \{ \ x = 0; \} \ else \{ \ z = 1; \}}
\texttt{int \ y = z;}

\textbf{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: let us 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