CS 473: COMPILER DESIGN
Compilation in a Nutshell

**Source Code**
(Character stream)

```c
if (b == 0) { a = 1; }
```

**Token stream:**

```c
if ( b == 0 ) { a = 0 ; }
```

**Abstract Syntax Tree:**

```
  If
    Eq
    Assn
      b
      0
      a
      1
    None
```

**Intermediate code:**

```assembly
l1:
  %cnd = icmp eq i64 %b, 0
  br i1 %cnd, label %l2, label %l3
l2:
  store i64* %a, 1
  br label %l3
l3:
```

**Assembly Code**

```assembly
l1:
  cmpq %eax, $0
  jeq 12
  jmp 13
l2:
  ...
```
**Compiler Goals**

• The most important thing for a compiler to do is to translate source programs into target programs that *correctly* implement the source.

• The second most important thing is to translate source programs into target programs that *efficiently* implement the source:
  – small number of instructions
  – faster instructions instead of slower (e.g., registers instead of memory accesses)
  – use less stack space
  – don’t recompute the same thing repeatedly
  – (without making compilation take too long)
Compilation in a Nutshell

Source Code
(Character stream)

```c
if (b == 0) { a = 1; }
```

Token stream:

```c
if ( b == 0 ) { a = 0 ; }
```

Abstract Syntax Tree:

```
If
  Eq
   b
   0
  Assn
   a
  None
   1
```

Intermediate code:

```
11: %cnd = icmp eq i64 %b, 0
    br i1 %cnd, label %l2, label %l3
12: store i64* %a, 1
    br label %l3
13: ...
```

Assembly Code

```
11:
    cmpq %eax, $0
    jeq 12
    jmp 13
12:
    ...
```
MORE DATAFLOW ANALYSES
Generalizing Dataflow Analyses

- Optimization = Analysis + Transformation

- The iterative constraint solving algorithm we used for liveness analysis applies to other kinds of analyses as well:
  - Reaching definitions: which variables are defined here?
  - Available expressions: which values have we already computed here?
  - Alias analysis: which variables could point to the same address here?

- For each of them, we can set up equations for in and out sets, and then use the equations to propagate information through the program’s CFG until we reach a fixpoint
REACHING DEFINITIONS
Reaching Definition Analysis

• Optimization = Analysis + Transformation

• Question: which variable definitions are in effect at each point in the program?
  – The opposite of liveness analysis, which asked “which variables have values that we’re still going to use?”
  – In liveness, we cared about which variables were live, but not what their values were!

• Useful for constant propagation & copy propagation
  – Constant propagation: if a variable use will always have the same value, replace it with that value
  – Copy propagation: if we have a statement $x = y$, replace uses of $x$ with $y$

• Input: Control flow graph for a program
• Output: $\text{in}[n]$ (resp. $\text{out}[n]$) contains the set of variable-defining nodes whose definitions may reach the beginning (resp. end) of node $n$
Example of Reaching Definitions

- Reaching definitions for a simple CFG:

```
1
  b = a + 2
  out[1]: {1}
  in[2]: {1}

2
  c = b * b
  out[2]: {1,2}
  in[3]: {1,2}

3
  b = c + 1
  out[3]: {2,3}
  in[4]: {2,3}

4
  return b * a
```

- Note that in and out sets contain *node numbers*, not variable names, in this analysis
Questions

Top
Reaching Definitions: Basic Sets

• Let `defs[a]` be the set of nodes that define the variable `a`  
• A node that defines a variable `a` both *generates* a definition of `a` and *overwrites* ("kills") any other definition of `a`  

• Define `gen[n]` and `kill[n]` as follows:

<table>
<thead>
<tr>
<th>Statement in n:</th>
<th><code>gen[n]</code></th>
<th><code>kill[n]</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a = b op c</code></td>
<td><code>{n}</code></td>
<td><code>defs[a] - {n}</code></td>
</tr>
<tr>
<td><code>a = load b</code></td>
<td><code>{n}</code></td>
<td><code>defs[a] - {n}</code></td>
</tr>
<tr>
<td><code>store b, a</code></td>
<td><code>Ø</code></td>
<td><code>Ø</code></td>
</tr>
<tr>
<td><code>a = f(b_1,...,b_n)</code></td>
<td><code>{n}</code></td>
<td><code>defs[a] - {n}</code></td>
</tr>
<tr>
<td><code>br L</code></td>
<td><code>Ø</code></td>
<td><code>Ø</code></td>
</tr>
<tr>
<td><code>br a L1 L2</code></td>
<td><code>Ø</code></td>
<td><code>Ø</code></td>
</tr>
<tr>
<td><code>return a</code></td>
<td><code>Ø</code></td>
<td><code>Ø</code></td>
</tr>
</tbody>
</table>

• Exercise: How can we compute `in[n]` and `out[n]` from `gen[n]` and `kill[n]`? In other words, how are the definitions available after a node affected by the definitions that node generates and overwrites?
Reaching Definitions: Basic Sets

• Let $\text{defs}[a]$ be the set of nodes that define the variable $a$

• A node that defines a variable $a$ both generates a definition of $a$ and overwrites (“kills”) any other definition of $a$

• Exercise: How can we compute $\text{in}[n]$ and $\text{out}[n]$ from $\text{gen}[n]$ and $\text{kill}[n]$? In other words, how are the definitions available after a node affected by the definitions that node generates and overwrites?

• $\text{in}[n] = \{a = 1, \ b = 2\}$

• $\text{statement}[n] = a = b + c$
  
  – $\text{gen}[n] = a = b + c$
  
  – $\text{kill}[n] = \text{every other definition of } a$

• $\text{out}[n] = \{a = b + c\} + (\text{in}[n] - \{a = 1\})$
Define the constraints that a reaching definitions solution must satisfy:

- \( \text{gen}[n] \subseteq \text{out}[n] \): if \( n \) generates a definition, the definition reaches the end of \( n \)

- \( \text{in}[n] \setminus \text{kill}[n] \subseteq \text{out}[n] \): if a definition reaches the start of \( n \), and isn’t overwritten by \( n \), then it reaches the end of \( n \)

- \( \text{in}[n] \supseteq \text{out}[n'] \) for all \( n' \) in \( \text{pred}[n] \): if a definition reaches the end of any predecessor of \( n \), then it reaches the start of \( n \)
• Convert constraints to update equations:

\[
\begin{align*}
in[n] &:= \bigcup_{n' \in \text{pred}[n]} \text{out}[n'] \\
\text{out}[n] &:= \text{gen}[n] \cup (\text{in}[n] - \text{kill}[n])
\end{align*}
\]

• Algorithm: initialize \(\text{in}[n]\) and \(\text{out}[n]\) to \(\emptyset\), iterate the updates until a fixed point is reached

• The algorithm terminates because \(\text{in}[n]\) and \(\text{out}[n]\) increase \textit{monotonically}, and at most include all definitions in the program

• Now we can transform the graph! For instance, constant propagation: If a use of \(x\) is only reached by one definition of \(x\), and that definition is of the form \(x = c\), then we can replace the use of \(x\) with the constant \(c\)
  
  — What if there’s only one reaching definition of \(x\), but it isn’t constant?
Dataflow Analysis: Example

\[
\begin{align*}
\text{in}[n] & := \bigcup_{n' \in \text{pred}[n]} \text{out}[n'] \\
\text{out}[n] & := \text{gen}[n] \cup (\text{in}[n] - \text{kill}[n])
\end{align*}
\]

Node 1:

\[
\begin{align*}
\text{in}[1] & = \emptyset \\
\text{out}[1] & = \text{gen}[1]
\end{align*}
\]

Node 1 generates the definition

\[
i = 0
\]
Apply the equations

1. Data flows from out to in
2. Data flows through $n$ if it is not in $\text{kill}[n]$

\[
\text{in}[n] := \bigcup_{n' \in \text{pred}[n]} \text{out}[n']
\]
\[
\text{out}[n] := \text{gen}[n] \cup (\text{in}[n] - \text{kill}[n])
\]
Dataflow Analysis: Example

\[
\text{in}[n] := \bigcup_{n' \in \text{pred}[n]} \text{out}[n'] \\
\text{out}[n] := \text{gen}[n] \cup (\text{in}[n] - \text{kill}[n])
\]

Apply the equations

1. Data flows from out to in
2. Data flows through \( n \) if it is not in \( \text{kill}[n] \)
Dataflow Analysis: Example

\[ \text{in}[n] := \bigcup_{n' \in \text{pred}[n]} \text{out}[n'] \]

\[ \text{out}[n] := \text{gen}[n] \cup (\text{in}[n] - \text{kill}[n]) \]

Apply the equations

1. Data flows from out to in
2. Data flows through \( n \) if it is not in \( \text{kill}[n] \)
Dataflow Analysis: Example

\[
\begin{align*}
    \text{in}[n] & := \bigcup_{n' \in \text{pred}[n]} \text{out}[n'] \\
    \text{out}[n] & := \text{gen}[n] \cup (\text{in}[n] - \text{kill}[n])
\end{align*}
\]

Apply the equations

1. Data flows from out to in
2. Data flows through \( n \) if it is not in \( \text{kill}[n] \)
Dataflow Analysis: Example

\[ \text{in}[n] := \bigcup_{n' \in \text{pred}[n]} \text{out}[n'] \]

\[ \text{out}[n] := \text{gen}[n] \cup (\text{in}[n] - \text{kill}[n]) \]

Apply the equations

1. Data flows from out to in
2. Data flows through \( n \) if it is not in \( \text{kill}[n] \)

Exercise: What should \( \text{out}[5] \) be?
Dataflow Analysis: Example

\[
\begin{align*}
\text{in}[n] & := \bigcup_{n' \in \text{pred}[n]} \text{out}[n'] \\
\text{out}[n] & := \text{gen}[n] \cup (\text{in}[n] - \text{kill}[n])
\end{align*}
\]

Apply the equations

1. Data flows from out to in
2. Data flows through \( n \) if it is not in \( \text{kill}[n] \)

Keep repeating until nothing changes
Dataflow Analysis: Example

\[ \text{in}[n] := \bigcup_{n' \in \text{pred}[n]} \text{out}[n'] \]
\[ \text{out}[n] := \text{gen}[n] \cup (\text{in}[n] - \text{kill}[n]) \]

Apply the equations

1. Data flows from out to in
2. Data flows through \( n \) if it is not in \( \text{kill}[n] \)

Keep repeating until nothing changes
Dataflow Analysis: Example

\[\text{in}[n] := \bigcup_{n' \in \text{pred}[n]} \text{out}[n']\]
\[\text{out}[n] := \text{gen}[n] \cup (\text{in}[n] - \text{kill}[n])\]

Apply the equations

1. Data flows from out to in
2. Data flows through \(n\) if it is not in \(\text{kill}[n]\)

Keep repeating until nothing changes
Dataflow Analysis: Example

\[
\text{in}[n] := \bigcup_{n' \in \text{pred}[n]} \text{out}[n'] \\
\text{out}[n] := \text{gen}[n] \cup (\text{in}[n] - \text{kill}[n])
\]

Apply the equations

1. Data flows from out to in
2. Data flows through \( n \) if it is not in \( \text{kill}[n] \)

Keep repeating until nothing changes

Can we propagate any constants?
No, because every use of \( i \) is reached by both 1 \((i = 0)\) and 5 \((i = i + 1)\)

3 is another definition of \( t \), so it is in \( \text{kill}[6] \) and won’t flow through 6

1 is another definition of \( i \), so it is in \( \text{kill}[5] \) and won’t flow through 5
Reaching Definitions: Analysis

• Convert constraints to update equations:

\[ \text{in}[n] := \bigcup_{n' \in \text{pred}[n]} \text{out}[n'] \]
\[ \text{out}[n] := \text{gen}[n] \cup (\text{in}[n] - \text{kill}[n]) \]

• Algorithm: initialize \( \text{in}[n] \) and \( \text{out}[n] \) to \( \emptyset \), iterate the updates until a fixed point is reached

• The algorithm terminates because \( \text{in}[n] \) and \( \text{out}[n] \) increase monotonically, and at most include all definitions in the program

• Now we can transform the graph! For instance, constant propagation: If a use of \( x \) is only reached by one definition of \( x \), and that definition is of the form \( x = c \), then we can replace the use of \( x \) with the constant \( c \)
Questions

Top
AVAILABLE EXPRESSIONS
Available Expressions

• Goal: don’t recompute what we’ve already computed (“common subexpression elimination”)
  
  \[ a = x + 1 \]
  \[ a = x + 1 \]
  \[ \cdots \]
  \[ b = x + 1 \]
  \[ b = a \]

• This transformation is safe if \( x+1 \) computes the same value at both places (because \( x \) hasn’t been modified in between)
  
  – In this case, we say that \( x+1 \) is an available expression at the assignment to \( b \)

• Dataflow sets:
  
  – \[ \text{in}[n] \] = set of nodes whose RHSs are available on entry to \( n \)
  – \[ \text{out}[n] \] = set of nodes whose RHSs are available on exit of \( n \)
Available Expressions: Basic Sets

- Each statement makes some values available and changes others

- Define \( \text{gen}[n] \) and \( \text{kill}[n] \) as follows:

<table>
<thead>
<tr>
<th>Statement in ( n )</th>
<th>( \text{gen}[n] )</th>
<th>( \text{kill}[n] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a = b \text{ op } c )</td>
<td>{ n } - \text{kill}[n]</td>
<td>\text{uses}[a]</td>
</tr>
<tr>
<td>( a = \text{load } b )</td>
<td>{ n } - \text{kill}[n]</td>
<td>\text{uses}[a]</td>
</tr>
<tr>
<td>( \text{store } b, a )</td>
<td>\emptyset</td>
<td>?</td>
</tr>
<tr>
<td>( \text{br } L )</td>
<td>\emptyset</td>
<td>\emptyset</td>
</tr>
<tr>
<td>( \text{br } a \text{ L1 L2} )</td>
<td>\emptyset</td>
<td>\emptyset</td>
</tr>
<tr>
<td>( a = f(b_1, \ldots, b_n) )</td>
<td>\emptyset</td>
<td>\text{uses}[a] \cup \text{all loads}</td>
</tr>
<tr>
<td>( \text{return } a )</td>
<td>\emptyset</td>
<td>\emptyset</td>
</tr>
</tbody>
</table>

- Note: \{ n \} - \text{kill}[n] means that, e.g., \( x = x + 1 \) doesn’t make \( x + 1 \) available
Available Expressions: Basic Sets

- Each statement makes some values available and changes others

- Define \textit{gen}[n] and \textit{kill}[n] as follows:

<table>
<thead>
<tr>
<th>Statement in \textit{n}</th>
<th>gen[n]</th>
<th>kill[n]</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{a = b op c}</td>
<td>{n} - \textit{kill}[n]</td>
<td>\textit{uses}[a]</td>
</tr>
<tr>
<td>\texttt{a = load b}</td>
<td>{n} - \textit{kill}[n]</td>
<td>\textit{uses}[a]</td>
</tr>
<tr>
<td>\texttt{store b, a}</td>
<td>\emptyset</td>
<td>\textit{loads from a}?</td>
</tr>
<tr>
<td>\texttt{br L}</td>
<td>\emptyset</td>
<td>\emptyset</td>
</tr>
<tr>
<td>\texttt{br a L1 L2}</td>
<td>\emptyset</td>
<td>\emptyset</td>
</tr>
<tr>
<td>\texttt{a = f(b_1,...,b_n)}</td>
<td>\emptyset</td>
<td>\textit{uses}[a] \cup \text{all loads}</td>
</tr>
<tr>
<td>\texttt{return a}</td>
<td>\emptyset</td>
<td>\emptyset</td>
</tr>
</tbody>
</table>

(a[1] is available)

```
int *x = a[1];
*x = 5;
```

We might never mention \texttt{a}, but still load from the memory that \texttt{a} points to!
Available Expressions: Basic Sets

• Each statement makes some values available and changes others

• Define \textit{gen}[n] and \textit{kill}[n] as follows:

  • Statement in \textit{n}:
    \begin{align*}
    a = b \text{ op } c & \quad \text{gen}[n] = \{n\} - \text{kill}[n] \quad \text{kill}[n] = \text{uses}[a] \\
    a = \text{load } b & \quad \{n\} - \text{kill}[n] \quad \text{uses}[a] \\
    \text{store } b, a & \quad \emptyset \quad \text{all loads} \\
    \text{br } L & \quad \emptyset \quad \emptyset \\
    \text{br } a \text{ L1 L2} & \quad \emptyset \quad \emptyset \\
    a = f(b_1, \ldots, b_n) & \quad \emptyset \quad \text{uses}[a] \cup \text{all loads} \\
    \text{return } a & \quad \emptyset \quad \emptyset
    \end{align*}

(a[1] is available)

\begin{verbatim}
int *x = a[1];
*x = 5;
\end{verbatim}

We might never mention \textit{a}, but still load from the memory that \textit{a} points to!
Available Expressions: Basic Sets

- Each statement makes some values available and changes others

- Define `gen[n]` and `kill[n]` as follows:

  - Statement in n:
    
    | Statement               | gen[n]                | kill[n]                                      |
    |------------------------|-----------------------|----------------------------------------------|
    | `a = b op c`           | `{n} - kill[n]`       | `uses[a]`                                   |
    | `a = load b`           | `{n} - kill[n]`       | `uses[a]`                                   |
    | store b, a             | ∅                     | `all loads that might alias with a`          |
    | br L                   | ∅                     | ∅                                            |
    | br a L1 L2             | ∅                     | ∅                                            |
    | `a = f(b₁,...,bₙ)`     | ∅                     | `uses[a] U all loads`                       |
    | return a               | ∅                     | ∅                                            |

(a[1] is available)

```cpp
int *x = a[1];
x[1] = 5;
```

We might never mention `a`, but still load from the memory that `a` points to!
Available Expressions: Basic Sets

• Each statement makes some values available and changes others
• gen[n] is “expression this node computes”, kill[n] is “expressions whose value this node might change”
• Define gen[n] and kill[n] as follows:

<table>
<thead>
<tr>
<th>Statement in n:</th>
<th>gen[n]</th>
<th>kill[n]</th>
</tr>
</thead>
<tbody>
<tr>
<td>a = b op c</td>
<td>{n} - kill[n]</td>
<td>uses[a]</td>
</tr>
<tr>
<td>a = load b</td>
<td>{n} - kill[n]</td>
<td>uses[a]</td>
</tr>
<tr>
<td>store b, a</td>
<td>\emptyset</td>
<td>all loads that might alias with a</td>
</tr>
<tr>
<td>br L</td>
<td>\emptyset</td>
<td>\emptyset</td>
</tr>
<tr>
<td>br a L1 L2</td>
<td>\emptyset</td>
<td>\emptyset</td>
</tr>
<tr>
<td>a = f(b_1, ..., b_n)</td>
<td>\emptyset</td>
<td>uses[a] \cup all loads</td>
</tr>
<tr>
<td>return a</td>
<td>\emptyset</td>
<td>\emptyset</td>
</tr>
</tbody>
</table>

• How do gen[n] and kill[n] relate to the expressions available before and after n?
Available Expressions: Constraints

- Define the constraints that an available expressions solution must satisfy.

- \( \text{gen}[n] \subseteq \text{out}[n] \): the expression computed in \( n \) is available after \( n \)

- \( \text{in}[n] – \text{kill}[n] \subseteq \text{out}[n] \): expressions available before \( n \) are available after \( n \) unless their pieces might be modified by \( n \)

- \( \text{in}[n] \subseteq \text{out}[n'] \) for all \( n' \) in \( \text{pred}[n] \): if an expression is available before \( n \), it must be because it was available after every predecessor of \( n \)
  - We need to know that the expression will \textit{definitely} be available, not that it \textit{might possibly} be

Note similarities and differences with constraints for “reaching definitions”. 
Available Expressions: Analysis

• Convert constraints to iterated updates:

\[
in[n] := \bigcap_{n' \in \text{pred}[n]} \text{out}[n']
\]

\[
\text{out}[n] := \text{gen}[n] \cup (\text{in}[n] - \text{kill}[n])
\]

• Algorithm: initialize \text{in}[n] and \text{out}[n] to \{set of all nodes\}, and iterate the updates until a fixed point is reached

• We start by assuming every expression in the program is available everywhere, and then rule them out based on \text{kill}[n]

• Now we can transform the graph! Common subexpression elimination: If we compute \(e\) when \(e\) is already available from a node \(t = e\), replace \(e\) with the variable \(t\)

\[
\begin{align*}
a &= x + 1 & a &= x + 1 \\
\ldots & \quad \ldots \\
b &= x + 1 & b &= a
\end{align*}
\]
Questions

Top
Comparing Dataflow Analyses

• Look at the update equations of the analyses

• Liveness: \((\text{backward})\)
  - Let \(\text{gen}[n] = \text{use}[n]\) and \(\text{kill}[n] = \text{def}[n]\)
  - \(\text{out}[n] := \bigcup_{n' \in \text{succ}[n]} \text{in}[n']\)
  - \(\text{in}[n] := \text{gen}[n] \cup (\text{out}[n] - \text{kill}[n])\)

• Reaching Definitions: \((\text{forward})\)
  - \(\text{in}[n] := \bigcup_{n' \in \text{pred}[n]} \text{out}[n']\)
  - \(\text{out}[n] := \text{gen}[n] \cup (\text{in}[n] - \text{kill}[n])\)

• Available Expressions: \((\text{forward})\)
  - \(\text{in}[n] := \bigcap_{n' \in \text{pred}[n]} \text{out}[n']\)
  - \(\text{out}[n] := \text{gen}[n] \cup (\text{in}[n] - \text{kill}[n])\)
Common Features

- All of these analyses have a domain over which they solve constraints
  - Liveness: sets of variables
  - Reaching defns., available exprs.: sets of nodes
- Each analysis has a notion of gen[\(n\)] and kill[\(n\)]
  - Used to describe how information propagates across a node
- Each analysis propagates information either forward or backward
  - Forward: in[\(n\)] defined in terms of out[\(n'\)] for predecessors \(n'\)
  - Backward: out[\(n\)] defined in terms of in[\(n'\)] for successors \(n'\)
- Each analysis has a way of combining information
  - Liveness & reaching definitions use union (\(\cup\))
  - Union expresses a property that holds for at least one path
  - Available expressions uses intersection (\(\cap\))
  - Intersection expresses a property that holds for all paths
Questions

Top
Dataflow Analysis on Basic Blocks

- Sometimes it is helpful to think of the fall-through between sequential instructions as an edge of the control-flow graph too.
- But it can be more efficient to do one basic block at a time!
Dataflow Analysis on Basic Blocks

\[
x = y \\
z = a + b \\
C\text{jump}
\]

\[
c = -z \\
\text{Jump}
\]

generates “\(x = y\)”  
keeps other defs of \(x\)

generates “\(z = a + b\)”  
keeps other defs of \(z\)
Dataflow Analysis on Basic Blocks

[Diagram showing basic blocks with instructions: x = y, z = a + b, c = -z, Cjump, Jump. Arrows indicate flow and dependencies.

generates “x = y” generates “z = a + b”
kills other defs of x kills other defs of z]
Dataflow Analysis on Basic Blocks

• Exercise: What if the second assignment was to x instead of z? What definitions would the block generate and kill?

```
x = y
z = a + b
Cjump
```

```c
x = y
z = a + b
```
generates “x = y” and “z = a + b”
kills other defs of x and z

```
c = -z
Jump
```

Exercise: What if the second assignment was to x instead of z? What definitions would the block generate and kill?
Dataflow Analysis on Basic Blocks

\[
x = y \\
x = a + b \\
Cjump
\]

\[
c = -z \\
Jump
\]

generates “x = y” and “x = a + b” kills other defs of x
In general, the combination of nodes 1 and 2 has:

\[
\text{kill}[n] = \text{kill}[1] \cup \text{kill}[2] \quad \quad \quad \text{gen}[n] = (\text{gen}[1] - \text{kill}[2]) \cup \text{gen}[2]
\]
Questions

Top
Undefined Behavior and Optimization

• Some languages (especially C) let compilers make simplifying assumptions

```c
if(x + 1 > x) do_this();
else do_that();
```
Undefined Behavior and Optimization

- Some languages (especially C) let compilers make simplifying assumptions

```c
if(x + 1 > x) do_this();
else do_that();
```

- Is \( x + 1 \) always greater than \( x \)?
  - Not if \( x \) is MAX_INT, and the addition overflows and wraps around!

- To allow this optimization, C standard says that a program that adds 1 to MAX_INT has *undefined behavior*: it can compile into anything, including whatever’s most convenient in the other cases

From *A Guide to Undefined Behavior in C and C++, Part 1*, John Regehr
Undefined Behavior and Optimization

• Some languages (especially C) let compilers make simplifying assumptions
  
  ```
  if(x + 1 > x) do_this(x);    do_this(x);
  else error("x is too big");
  ```

• Now we’ve optimized a program with a safety check into a program without a safety check!

• And C says this is allowed, because we did the undefined operation (adding 1 to MAX_INT) before getting to the error message

• At some point, the Linux kernel devs made a special GCC flag to tell the compiler not to optimize away null pointer checks like this

• If you’re writing a compiler, maybe give a warning and compile safely instead of doing the technically correct but dangerous optimization!

From A Guide to Undefined Behavior in C and C++, Part 1, John Regehr