There are two principal methods of solving data-flow problems:

1. Let gen, kill, in, out be AST attributes and the data-flow equations attribute evaluation rules. We'll look at this later.

2. Treat data-flow equations as recurrences, and iterate over the set of equations until a solution is found.

Sets are stored as bit-vectors, with one element for each possible object.

\[
\text{in}[B1] = \{d_3, d_5, d_7\} \\
\equiv \begin{array}{cccccccc}
0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\
d_1 & d_2 & d_3 & d_4 & d_5 & d_6 & d_7 & d_8
\end{array}
\]
Reaching Definitions — Equations

\[ \text{out}[B] = \text{gen}[B] \cup (\text{in}[B] - \text{kill}[B]) \]
\[ \text{in}[B] = \bigcup_{\text{preds} P \text{ of } B} \text{out}[P] \]

- A definition \( d : a := b + c \) reaches a use of \( a \) at point \( p \), if the value given to \( a \) at \( d \) could be used at \( p \).
- \( \text{gen}[B] \) is the set of definitions generated within \( B \), that reach the end of \( B \).
- \( \text{kill}[B] \) is the set of definitions outside \( B \), killed by definitions within \( B \).
- \( \text{in}[B] \) is the set of definitions valid at the entrance to \( B \), \( \text{out}[B] \) is those valid at the exit of \( B \).
- The equations for \( \text{in} \) and \( \text{out} \) are valid for each basic block.

### Iterative Algorithms I

1. Compute \( \text{gen} \) and \( \text{kill} \) for each block.
2. Set up the \( 2n \) in- and out-equations for the \( n \) basic blocks, and set \( \text{in}[B] = \text{out}[B] = \{ \} \) for each block \( B \).
3. Repeat until no more changes:
   - For each block \( B \) eval. \( \text{in}[B] \) & \( \text{out}[B] \).

   **Formal Algorithm:**

   ```plaintext
   FOR each block \( B \) DO
   \[ \text{out}[B] := \text{in}[B] := \{ \} ; \]
   END;
   WHILE any \( \text{out}[B] \) has changed DO
   FOR each block \( B \) DO
   \[ \text{in}[B] := \bigcup_{\text{preds} P \text{ of } B} \text{out}[P] \]
   \[ \text{out}[B] := \text{gen}[B] \cup (\text{in}[B] - \text{kill}[B]) \]
   END;
   END
   ```

### Iterative Alg. Example I (a)

**REPEAT**

\( d_1 : i := \ldots ; \)
\( \text{IF } \ldots \text{THEN} \)
\( d_2 : i := \ldots \)
\( \text{ELSE} \)
\( d_3 : i := \ldots \)
\( \text{ENDIF} ; \)
\( d_4 : k := \ldots \)
**UNTIL** \( \ldots ; \)

- Start by setting up the \( 2n \) in- and out-equations (slide (b)).
- Simplify the example by inlining \( \text{gen} \) and \( \text{kill} \) into the equations for \( \text{in} \) and \( \text{out} \) (slide (c)).
- Visit each block in turn (we use numerical order, \( B_1, B_2, B_3, B_4 \)) and evaluate \( \text{in} \) and \( \text{out} \) (slides (d)–(g)).

### Iterative Alg. Example I (b)

- \( \text{g}[B_2] = \{d_2\} \)
- \( \text{k}[B_2] = \{d_1, d_3\} \)
- \( \text{g}[B_3] = \{d_3\} \)
- \( \text{k}[B_3] = \{d_1, d_2\} \)
- \( \text{i}[B_2] = \text{g}[B_2] \cup (\text{i}[B_2] - \text{k}[B_2]) \)
- \( \text{o}[B_2] = \text{g}[B_2] \cup (\text{i}[B_2] - \text{k}[B_2]) \)
- \( \text{i}[B_3] = \text{g}[B_3] \cup (\text{i}[B_3] - \text{k}[B_3]) \)
- \( \text{o}[B_3] = \text{g}[B_3] \cup (\text{i}[B_3] - \text{k}[B_3]) \)
- \( \text{g}[B_4] = \{d_4\} \)
- \( \text{k}[B_4] = \{\} \)
- \( \text{i}[B_4] = \text{g}[B_4] \cup (\text{i}[B_4] - \text{k}[B_4]) \)
- \( \text{o}[B_4] = \text{g}[B_4] \cup (\text{i}[B_4] - \text{k}[B_4]) \)
**Iterative Alg. Example I (g) – 3rd Iteration**

Example II (b) – 2nd Iteration

Example II (a) – 1st Iteration

Live Variable Analysis
Live Variable Analysis I

- For each definition/use of a variable $V$, **Global Live Variable Analysis** answers the question: "Could the value of $V$ computed/used here be used further on in the program?"

- If a variable $V$ is stored in a register $R5$ and $V$ is dead at the end of the block, then we don’t have to store $R5$ back into $V$.
- Assignments to dead variables can be removed.

```
R5 := V;
R5 := R5 + 1;
```

V (stored in R5)

is incremented

V is dead here.

No further used of $V$ here.

Live Variable Analysis II

- **in**[$B$] Variables live on entrance to $B$.
- **out**[$B$] Variables live on exit from $B$.
- **def**[$B$] Variables assigned values in $B$ before the variable is used:

```
B := ... C ...;
C := ...
... := ... B ...
```

- **use**[$B$] Variables whose values are used before being assigned to:

```
B := ... C ...
C := ...
... := ... B ...
```

Live Variable Analysis III

- Data-Flow (Equations):

\[ \text{in}[B] = \text{use}[B] \cup (\text{out}[B] - \text{def}[B]) \]

- Data-Flow (English):

V is live at the entrance to $B$ if

1. it is being used before it’s defined (i.e. $V \in \text{use}[B]$)

\[ \text{in} = \{ ... C ... \} \quad \text{use} = \{ C \} \]

\[ B := ... C ...; \]

\[ C := ...; \]

C is in in[B] since its value is used before C is defined.

2. it is live at the exit of the block, and it is not defined within the block (i.e. $V \in \text{out}[B]$ and $V \not\in \text{def}[B]$)

\[ \text{in} = \{ ... C ... \} \quad \text{def} = \{ B \} \]

\[ B := ...; \]

\[ \text{out} = \{ ... C, B ... \} \]

Live Variable Analysis IV

- Data-Flow (Equations):

\[ \text{out}[B] = \bigcup \text{in}[S] \]

- Data-Flow (English):

A variable $V$ is live coming out of $B$ if it is live going into any one of $B$’s successors.
Live Variable Analysis V – Example

B := 1
D := A + B
C := 1

Note that B is live at the beginning of the initial basic block (B1). This means that B may be used before it’s defined, a common error.

**Live Variables VI – Example**

\[ \text{out}[B4] = \{ \} \] since out is the union of all of B4’s successor’s in, and B4 doesn’t have any successors.

\[ \text{in}[B4] = \{ \} \] because both A & B are live coming in to B4, i.e. their values will be used before they are assigned new values.

\[ \text{out}[B3] = \text{in}[B4] = \{ A, B \} \] because the values of A and B will be used in B3’s successor block, B4. Note that since \( C \not\in \text{out}[B3] \) C’s value is dead and the assignment \( C := 1 \) can be removed.

\[ \text{out}[B1] = \{ A \} \cup \{ A, B \} = \{ A, B \} \] since if we take the left branch (through B2) A will be used further on, and if we take the right branch (through B3) both A and B will have a future use.

\[ \text{in}[B1] = \{ B \} \] since B’s value is used but not defined in B.

**Classification — Forward vs. Backward Flow**

- Data-flow problems can be classified according to the direction of flow:
  - **Forward-flow problems**: Data flows from the initial block to the end block.
    - Out-sets are computed from In-sets within basic blocks,
    - In-sets are computed from Out-sets across basic blocks.
  - **Backward-flow problems**: Data flows from the end block to the initial block.
    - In-sets are computed from Out-sets within basic blocks,
    - Out-sets are computed from In-sets across basic blocks.

**Classifying Data-Flow Problems**

- Data-flow problems can be classified according to the direction of flow:
  - **Forward-flow problems**: Data flows from the initial block to the end block.
    - Out-sets are computed from In-sets within basic blocks,
    - In-sets are computed from Out-sets across basic blocks.
  - **Backward-flow problems**: Data flows from the end block to the initial block.
    - In-sets are computed from Out-sets within basic blocks,
    - Out-sets are computed from In-sets across basic blocks.
Forward vs. Backward Flow — Equations

<table>
<thead>
<tr>
<th></th>
<th>Forward-Flow</th>
<th>Backward-Flow</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any Path</td>
<td>( o_B = g_B \cup (i_B - k_B) )</td>
<td>( i_B = g_B \cup (o_B - k_B) )</td>
</tr>
<tr>
<td>All Paths</td>
<td>( o_B = g_B \cup (i_B - k_B) )</td>
<td>( i_B = g_B \cup (o_B - k_B) )</td>
</tr>
</tbody>
</table>

- \( P(B) = \) Predecessors of \( B \), \( S(B) = \) Successors of \( B \).
- \( i_B = in_B \), \( o_B = out_B \), \( g_B = gen_B \), \( k_B = kill_B \), \( o_b = out_b \), \( i_b = in_b \).

Classification — Any- vs. All-Path

- We classify data-flow problems by the way they combine incoming information:
  - **Any-path problems:** All values coming in to a block are valid.
    Use \( \cup \).
  - **All-path problems:** Only values coming in to a block through every path are valid. Use \( \cap \).

<table>
<thead>
<tr>
<th></th>
<th>Forward-Flow</th>
<th>Backward-Flow</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any Path</td>
<td>Reaching Definitions</td>
<td>Live Variables</td>
</tr>
<tr>
<td></td>
<td>Uninitialized Variables</td>
<td>Du-chains</td>
</tr>
<tr>
<td>All Paths</td>
<td>Available Expressions</td>
<td>Very Busy Expressions</td>
</tr>
<tr>
<td></td>
<td>Copy Propagation</td>
<td></td>
</tr>
</tbody>
</table>

Summary

- With \( B \) blocks & bit-vectors of length \( V \), iterative data-flow analysis is \( O(B^2 \times V) \) in the worst case.
Homework

Show each step of the iterative reaching definitions algorithm applied to the procedure body below:

```
K := 1; I := 2;
REPEAT
  IF I = 4 THEN
    A := K + 1;
  ELSE
    A := K + 2;
    I := I + A;
  ENDIF;
UNTIL I <= 10;
K := K + A;
```

Exam Problem I (a) [07.430 '95]

An expression $E$ is *very busy* if – regardless of which path we take through the flow graph – $E$’s value will be used before it is killed. Example ($A+3$ is *very busy*):

```
BEGIN
IF expr THEN
  V := A + 3;
  R := K + 3;
ELSE
  Z := A + 3;
  K := 5;
  L := K + 3;
END;
END
```
The data-flow equations for computing very busy expressions are:

\[
\begin{align*}
\text{in}[B] & = \text{used}[B] \cup \left( \text{out}[B] - \text{killed}[B] \right) \\
\text{out}[B] & = \bigcap_{\text{successors } S \text{ of } B} \text{in}[S]
\end{align*}
\]

Problems:

1. Give an iterative pseudo-code routine for computing \text{in} and \text{out}.
2. Is \textit{very-busy expressions} a forward-flow or a backward-flow problem?
3. Show the workings of the algorithm on the procedure body in the next slide: