Similarly, in Java:
\[
a[1] = 1; \\
b[1] = 0; \\
\text{print}(a[1]);
\]
is 1 or 0 printed?

Points-to analysis aims to determine what variables or heap objects a pointer or reference may access. Exact analysis is impossible (why?). But fast and reasonably accurate analyses are known.

### Review of Compiler Optimizations

1. **Redundant Expression Elimination**
   (Common Subexpression Removal)
   
   Use an address or value that has been previously computed. Consider control and data dependencies.

2. **Partially Redundant Expression (PRE) Elimination**
   
   A variant of Redundant Expression Elimination. If a value or address is redundant along some execution paths, add computations to other paths to create a fully redundant expression (which is then removed).
   
   Example:
   
   ```java
   if (i > j)
   a[i] = a[j];
   a[i] = a[i] * 2;
   ```

3. **Constant Propagation**
   
   If a variable is known to contain a particular constant value at a particular point in the program, replace references to the variable at that point with the constant value.

4. **Copy Propagation**
   
   After the assignment of one variable to another, a reference to one variable may be replaced with the value of the other variable (until one or the other of the variables is reassigned).
   (This may also “set up” dead code elimination. Why?)

5. **Constant Folding**
   
   An expression involving constant (literal) values may be evaluated and simplified to a constant result value. Particularly useful when constant propagation is performed.

6. **Dead Code Elimination**
   
   Expressions or statements whose values or effects are unused may be eliminated.

7. **Loop Invariant Code Motion**
   
   An expression that is invariant in a loop may be moved to the loop’s header, evaluated once, and reused within the loop.
   
   Safety and profitability issues may be involved.
8. Scalarization (Scalar Replacement)
A field of a structure or an element of an array that is repeatedly read or written may be copied to a local variable, accessed using the local, and later (if necessary) copied back.
This optimization allows the local variable (and in effect the field or array component) to be allocated to a register.

9. Local Register Allocation
Within a basic block (a straight line sequence of code) track register contents and reuse variables and constants from registers.

10. Global Register Allocation
Within a subprogram, frequently accessed variables and constants are allocated to registers. Usually there are many more register candidates than available registers.

11. Interprocedural Register Allocation
Variables and constants accessed by more than one subprogram are allocated to registers. This can greatly reduce call/return overhead.

12. Register Targeting
Compute values directly into the intended target register.

13. Interprocedural Code Motion
Move instructions across subprogram boundaries.

14. Call Inlining
At the site of a call, insert the body of a subprogram, with actual parameters initializing formal parameters.

15. Code Hoisting and Sinking
If the same code sequence appears in two or more alternative execution paths, the code may be hoisted to a common ancestor or sunk to a common successor. (This reduces code size, but does not reduce instruction count.)

16. Loop Unrolling
Replace a loop body executed N times with an expanded loop body consisting of M copies of the loop body. This expanded loop body is executed N/M times, reducing loop overhead and increasing optimization possibilities within the expanded loop body.

17. Software Pipelining
A value needed in iteration $i$ of a loop is computed during iteration $i-1$ (or $i-2$, ...). This allows long latency operations (floating point divides and square roots, low hit-ratio loads) to execute in parallel with other operations. Software pipelining is sometimes called symbolic loop unrolling.

18. Strength Reduction
Replace an expensive instruction with an equivalent but cheaper alternative. For example a division may be replaced by multiplication of a reciprocal, or a list append may be replaced by cons operations.
19. Data Cache Optimizations

- Locality Optimizations
  Cluster accesses of data values both spatially (within a cache line) and temporally (for repeated use). Loop interchange and loop tiling improve temporal locality.

- Conflict Optimizations
  Adjust data locations so that data used consecutively and repeatedly don’t share the same cache location.

20. Instruction Cache Optimizations

Instructions that are repeatedly executed should be accessed from the instruction cache rather than the secondary cache or memory. Loops and “hot” instruction sequences should fit within the cache. Temporally close instruction sequences should not map to conflicting cache.

Control Flow Graphs

A Control Flow Graph (CFG) models possible execution paths through a program. Nodes are basic blocks and arcs are potential transfers of control.

For example,

```
if (a > 0)
    b = 1;
else
    b = 2;
a = c + b;
```

Basic Blocks

A basic block is a linear sequence of instructions containing no branches except at the very end. A basic block is always executed sequentially as a unit.

For a Basic Block b:

Let \( \text{Preds}(b) \) = the set of basic blocks that are Immediate Predecessors of b in the CFG.

Let \( \text{Succ}(b) \) = the set of basic blocks that are Immediate Successors to b in the CFG.
### Data Flow Problems

A data flow problem is a program analysis computed on a control flow graph.

A data flow problem may be **forward** (following a program’s control flow) or **reverse** (opposite a program’s control flow).

Informally, forward analyses “remember the past” while reverse analyses “predict the future.”

Some analyses determine that an event may have occurred, while others determine that an event must have occurred.

Some analyses compute a set of values, while others are Boolean-valued.

### Two important data flow problems are **Reaching Definitions** and **Liveness**.

For a given use of a variable $v$, reaching definitions tell us which assignments to $v$ may reach (affect) the current value of $v$. Reaching definition analysis is useful in both optimization and debugging.

Liveness analysis tells us at a particular point in a program whether the current value of variable $v$ will ever be used. A variable that is not live is dead. A dead value need not be kept in memory, or perhaps even be computed.

### Reaching Definitions

For a Basic Block $b$ and Variable $V$:

Let $\text{DefsIn}(b) = \text{the set of basic blocks that contain definitions of } V \text{ that reach (may be used in) the beginning of Basic Block } b$.

Let $\text{DefsOut}(b) = \text{the set of basic blocks that contain definitions of } V \text{ that reach (may be used in) the end of Basic Block } b$.

The sets $\text{Preds}$ and $\text{Succ}$ are derived from the structure of the CFG.

They are given as part of the definition of the CFG.

### DefsIn and DefsOut must be computed, using the following rules:

1. If Basic Block $b$ contains a definition of $V$ then
   \[ \text{DefsOut}(b) = \{b\} \]

2. If there is no definition to $V$ in $b$ then
   \[ \text{DefsOut}(b) = \text{DefsIn}(b) \]

3. For the First Basic Block, $b_0$:
   \[ \text{DefsIn}(b_0) = \phi \]

4. For all Other Basic Blocks
   \[ \text{DefsIn}(b) = \bigcup_{p \in \text{Preds}(b)} \text{DefsOut}(p) \]
Liveness Analysis

For a Basic Block \( b \) and Variable \( V \):

\[
\text{LiveIn}(b) = \text{true if } V \text{ is Live (will be used before it is redefined) at the beginning of } b.
\]

\[
\text{LiveOut}(b) = \text{true if } V \text{ is Live (will be used before it is redefined) at the end of } b.
\]

LiveIn and LiveOut are computed, using the following rules:

1. If Basic Block \( b \) has no successors then
   \[ \text{LiveOut}(b) = \text{false} \]

2. For all Other Basic Blocks
   \[ \text{LiveOut}(b) = \bigvee_{s \in \text{Succ}(b)} \text{LiveIn}(s) \]

3. LiveIn(b) =
   If \( V \) is used before it is defined in Basic Block \( b \)
   Then true
   Elsif \( V \) is defined before it is used in Basic Block \( b \)
   Then false
   Else \( \text{LiveOut}(b) \)

Example

[Diagram showing the liveness analysis process with variables and blocks labeled accordingly]
Reading Assignment
• Section 14.3 - 14.4 of CaC

Data Flow Frameworks
• Data Flow Graph:
  Nodes of the graph are basic blocks
  or individual instructions.
  Arcs represent flow of control.

  Forward Analysis:
  Information flow is the same
direction as control flow.

  Backward Analysis:
  Information flow is the opposite
direction as control flow.

  Bi-directional Analysis:
  Information flow is in both
directions. (Not too common.)

• Meet Lattice
  Represents solution space for the
data flow analysis.

• Meet operation
  (And, Or, Union, Intersection, etc.)
  Combines solutions from
  predecessors or successors in the
  control flow graph.

• Transfer Function
  Maps a solution at the top of a node
to a solution at the end of the node
(forward flow)
or
  Maps a solution at the end of a node
to a solution at the top of the node
(backward flow).
Example: Available Expressions

This data flow analysis determines whether an expression that has been previously computed may be reused.

Available expression analysis is a forward flow problem—computed expression values flow forward to points of possible reuse.

The best solution is True—the expression may be reused.

The worst solution is False—the expression may not be reused.

The Meet Lattice is:

T (Expression is Available)
F (Expression is Not Available)

As initial values, at the top of the start node, nothing is available. Hence, for a given expression,

\[ \text{AvailIn}(b_0) = F \]

We choose an expression, and consider all the variables that contribute to its evaluation.

Thus for \( e_1 = a + b - c \), \( a, b \) and \( c \) are \( e_1 \)'s operands.

The transfer function for \( e_1 \) in block \( b \) is defined as:

If \( e_1 \) is computed in \( b \) after any assignments to \( e_1 \)'s operands in \( b \)
Then  \( \text{AvailOut}(b) = T \)
Elsif any of \( e_1 \)'s operands are changed after the last computation of \( e_1 \) or \( e_1 \)'s operands are changed without any computation of \( e_1 \)
Then  \( \text{AvailOut}(b) = F \)
Else  \( \text{AvailOut}(b) = \text{AvailIn}(b) \)

The meet operation (to combine solutions) is:

\[ \text{AvailIn}(b) = \text{AND} \text{ AvailOut}(p) \]

\[ p \in \text{Pred}(b) \]

Example: \( e_1 = v + w \)
Circularities Require Care

Since data flow values can depend on themselves (because of loops), care is required in assigning initial “guesses” to unknown values.

Consider
If the flow value on the loop backedge is initially set to false, it can never become true. (Why?)
Instead we should use True, the identity for the AND operation.

```
z = v + w
```

```
v = 9
y = v + w
w = 5
x = v + w
z = v + w
```

```
v = 2
stop
```

```