1. statement-by-statement code generation
2. peephole optimization
3. “global” optimization
**Peephole optimization**

**Peephole:** a short sequence of target instructions that may be replaced by a shorter/faster sequence.

One optimization may make further optimizations possible. ⇒ several passes may be required.

1. **Redundant load/store elimination:**
   
   MOV R0 a
   MOV a R0  ← delete if in same B.B.

2. **Algebraic simplification:** eliminate instructions like the following:
   
   $$x = x + 0 \quad x = x \times 1$$

3. **Strength reduction:** replace expensive operations by equivalent cheaper operations, e.g.
   
   - $$x^2 \rightarrow x \times x$$
   - fixed-point multiplication/division by power of 2  →  shift
4. Jumps:

\[
\begin{align*}
\text{goto L1} & \quad \text{if } a < b \text{ goto L1} \\
\ldots & \quad \ldots \\
L1: \quad \text{goto L2} & \quad L1: \quad \text{goto L2} \\
\downarrow & \quad \downarrow \\
\text{goto L2} & \quad \text{if } a < b \text{ goto L2} \\
\ldots & \quad \ldots \\
L1: \quad \text{goto L2} & \quad L1: \quad \text{goto L2}
\end{align*}
\]

If there are no other jumps to L1, and it is preceded by an unconditional jump, it may be eliminated.

If there is only jump to L1 and it is preceded by an unconditional goto

\[
\begin{align*}
\text{goto L1} & \quad \text{if } a < b \text{ goto L2} \\
\ldots & \quad \text{goto L3} \\
L1: \quad \text{if } a < b \text{ goto L2} & \quad \Rightarrow \quad \ldots \\
L3: & \quad L3:
\end{align*}
\]
5. Unreachable code: unlabeled instruction following an unconditional jump may be eliminated

```c
#define DEBUG 0
...
if (debug) {
    /* print stmts */
}
```

```
if 0 != 1 goto L2
/* print stmts */
L2:
```

```
if debug != 1 goto L2
/* print stmts */
L2:
```

eliminate
Example

\[ i = m-1; \ j = n; \ v = a[n]; \]

while (1) {
    do i = i+1; while (a[i] < v);
    do j = j-1; while (a[j] > v);
    if (i >= j) break;
    x = a[i]; a[i] = a[j]; a[j] = x;
}

x = a[i]; a[i] = a[n]; a[n] = x;
1  \( i = m-1 \)
2  \( j = n \)
3  \( t1 = 4*n \)
4  \( v = a[t1] \)
5  \( i = i+1 \)
6  \( t2 = 4*i \)
7  \( t3 = a[t2] \)
8  if \( t3 < v \) goto 5
9  \( j = j-1 \)
10 \( t4 = 4*j \)
11 \( t5 = a[t4] \)
12 if \( t5 > v \) goto 9
13 if \( i>=j \) goto 23
14 \( t6 = 4*i \)
15 \( x = a[t6] \)
16 \( t7 = 4*i \)
17 \( t8 = 4*j \)
18 \( t9 = a[t8] \)
19 \( a[t7] = t9 \)
20 \( t10 = 4*j \)
21 \( a[t10] = x \)
22 goto 5
23 \( t11 = 4*i \)
24 \( x = a[t11] \)
25 \( t12 = 4*i \)
26 \( t13 = 4*n \)
27 \( t14 = a[t13] \)
28 \( a[t12] = t14 \)
29 \( t15 = 4*n \)
30 \( a[t15] = x \)
1. \( i = m - 1 \)
2. \( j = n \)
3. \( t_1 = 4 \times n \)
4. \( v = a[t_1] \)
5. \( i = i + 1 \)
6. \( t_2 = 4 \times i \)
7. \( t_3 = a[t_2] \)
8. if \( t_3 < v \) goto 5
9. \( j = j - 1 \)
10. \( t_4 = 4 \times j \)
11. \( t_5 = a[t_4] \)
12. if \( t_5 > v \) goto 9
13. if \( i \geq j \) goto 23
14. \( t_6 = 4 \times i \)
15. \( x = a[t_6] \)

16. \( t_7 = 4 \times i \)
17. \( t_8 = 4 \times j \)
18. \( t_9 = a[t_8] \)
19. \( a[t_7] = t_9 \)
20. \( t_{10} = 4 \times j \)
21. \( a[t_{10}] = x \)
22. goto 5
23. \( t_{11} = 4 \times i \)
24. \( x = a[t_{11}] \)
25. \( t_{12} = 4 \times i \)
26. \( t_{13} = 4 \times n \)
27. \( t_{14} = a[t_{13}] \)
28. \( a[t_{12}] = t_{14} \)
29. \( t_{15} = 4 \times n \)
30. \( a[t_{15}] = x \)
\[
\begin{align*}
i &= m-1 \\
j &= n \\
t1 &= 4*n \\
v &= a[t1] \\
\end{align*}
\]

\[
\begin{align*}
i &= i+1 \\
t2 &= 4*i \\
t3 &= a[t2] \\
\text{if } t3 < v \text{ goto } B2 \\
\end{align*}
\]

\[
\begin{align*}
j &= j-1 \\
t4 &= 4*j \\
t5 &= a[t4] \\
\text{if } t5 > v \text{ goto } B3 \\
\end{align*}
\]

\[
\begin{align*}
\text{if } i \geq j \text{ goto } B6 \\
\end{align*}
\]
Common subexpression elimination:

Def. $E$ is a common subexpression at some point in the program if it has been previously computed and the values of variables in $E$ have not changed since the last computation.

NOTE: local vs. global C.S.E.

array expressions
Optimization methods

- **Common subexpression elimination:**
  Def. $E$ is a common subexpression at some point in the program if it has been previously computed and the values of variables in $E$ have not changed since the last computation.
  
  **NOTE:** local vs. global C.S.E. array expressions

- **Copy propagation:** after the copy statement $x = y$, use $y$ wherever possible in place of $x$.
Optimization methods

- **Common subexpression elimination:**
  **Def.** $E$ is a common subexpression at some point in the program if it has been previously computed and the values of variables in $E$ have not changed since the last computation.
  
  **NOTE:** local vs. global C.S.E.
  
  array expressions

- **Copy propagation:** after the copy statement $x = y$, use $y$ wherever possible in place of $x$

- **Dead code elimination:**
  **Dead variable:** $v$ is dead at a point if its value is not used after that point.
  **Dead code:** statements which compute values that are never used.
Loop optimizations

- **Code motion**: if a statement is independent of the number of times a loop is executed (≡ loop invariant computation), move it outside the loop
  Example:

  ```c
  while (i <= N-1) ... ⇒ t = N-1
  while (i <= t) ...
  ```

- **Induction variable elimination**: 
  **Induction variable**: variable whose value has a simple relation with no. of loop iterations

- **Strength reduction**: replacing expensive operation by cheaper one (e.g. multiplication by addition)
Data flow analysis

Motivation: collect information like live variables, common subexpressions, etc. about entire program for optimization (and code generation)

Plan:
- Structured programs
  - reaching definitions
- Flow graphs / Iterative solutions
  - reaching definitions
  - available expressions
  - live variables
Structured programs

- Control-flow changes only via `if` and `while` stmts (no arbitrary `goto`s)

- Source-level grammar:

  \[
  S \rightarrow \text{id} = E \\
  \quad \mid S ; S \\
  \quad \mid \text{if } E \text{ then } S \text{ else } S \\
  \quad \mid \text{do } S \text{ while } E
  \]
Reaching definitions

**Point:** position between 2 adjacent stmts within a BB, before the 1st stmt in a BB, and after the last stmt in a BB

**Path:** sequence of points \( p_1, p_2, \ldots, p_n \) s.t. \( p_i \) immediately precedes and \( p_{i+1} \) immediately follows an instruction, or \( p_i \) ends a block, and \( p_{i+1} \) begins a successor block

**Definition** of a variable \( x \) is a stmt that assigns or may assign a value to \( x \)

- **Unambiguous** defn. – stmt that definitely assigns a value to \( x \), e.g. direct assignments, I/O
- **Ambiguous** defn. – stmt that *may* change the value of \( x \), e.g. indirect assignment, procedure call

**Kill:** a definition of a variable is killed on a path, if there is a (unambiguous) definition of the variable along that path
Reaching Definition: A definition $d$ reaches point $p$ if there is a path from the point immediately following $d$ to $p$, and $d$ is not killed along that path.

Applications: used for constant folding, code motion, induction variable elimination, dead code elimination, etc.
Reaching definitions

Reaching Definition: A definition $d$ reaches point $p$ if there is a path from the point immediately following $d$ to $p$, and $d$ is not killed along that path.

Applications: used for constant folding, code motion, induction variable elimination, dead code elimination, etc.

- $in(S)$ — set of definitions reaching the beginning of stmt $S$
- $out(S)$ — set of definitions reaching the end of stmt $S$
- $gen(S)$ — set of definitions that reach the end of $S$, irrespective of whether they reach the beginning of $S$
- $kill(S)$ — set of definitions that never reach the end of $S$, even if they reach the beginning of $S$
\( d: a = b + c \)

\[
\begin{align*}
gen(S) &= \{ d \} \\
\text{kill}(S) &= D_a - \{ d \}
\end{align*}
\]

\[
\begin{align*}
gen(S) &= gen(S_2) \cup (gen(S_1) - \text{kill}(S_2)) \\
\text{kill}(S) &= \text{kill}(S_2) \cup (\text{kill}(S_1) - gen(S_2))
\end{align*}
\]

\[
\begin{align*}
in(S_1) &= in(S) \\
in(S_2) &= out(S_1) \\
out(S) &= out(S_2)
\end{align*}
\]
Data flow equations

\[ \text{gen}(S) = \text{gen}(S_1) \cup \text{gen}(S_2) \]
\[ \text{kill}(S) = \text{kill}(S_1) \cap \text{kill}(S_2) \]
\[ \text{in}(S_1) = \text{in}(S_2) = \text{in}(S) \]
\[ \text{out}(S) = \text{out}(S_1) \cup \text{out}(S_2) \]

\[ \text{gen}(S) = \text{gen}(S_1) \]
\[ \text{kill}(S) = \text{kill}(S_1) \]
\[ \text{in}(S_1) = \text{in}(S) \cup \text{gen}(S_1) \]
\[ \text{out}(S) = \text{out}(S_1) \]
Data flow equations

1. Compute *gen*, *kill* (synthesized attributes) bottom up from smallest stmt to largest stmt.

2. Let \( S_0 \) represent the complete program. Then \( \text{in}(S_0) = \emptyset \).

3. For \( S_1 \), a sub-statement of \( S \):
   (i) calculate \( \text{in}(S_1) \) in terms of \( \text{in}(S) \);
   (ii) calculate \( \text{out}(S_1) \) using the equation
         \[
         \text{out}(S) = \text{gen}(S) \cup (\text{in}(S) - \text{kill}(S))
         \]

4. Calculate \( \text{out}(S) \) in terms of \( \text{out}(S_1) \).
Iterative approach

Reaching definitions

Input: flow graph with $gen$, $kill$ sets computed for each BB

Output: $in$, $out$ sets for each block

Method:
1. For each block, initialize
   $out(B) \leftarrow gen(B)$
   (assume $in(B) = \emptyset$)
2. While changes occur:
   for each block $B$
   
   $in(B) \leftarrow \bigcup_P out(P)$  \hspace{1cm} where $P$ - predecessor of $B$
   $out(B) \leftarrow gen(B) \cup (in(B) - kill(B))$
Reaching definitions

- Changes are monotonic $\implies$ method converges.
- While loop (step 2) simulates all possible alternatives for control flow during the execution of the program.
- If a definition reaches a point, it can do so along a cycle-free path.
  - Longest cycle free path in the graph can cover at most all nodes, at most once.
  - $\implies$ Upper bound on # iterations $=$ # of nodes in the flow graph.
  - Avg. # of iterations for convergence for real programs $= 5$
**U-D chains**

**Definition:** for a given *use* of a variable $a$, the *ud chain* is a list of all definitions of $a$ that reach that use.

**Method:**
Given a use of variable $a$ in block $B$

1. If the use is not preceded by an unambiguous defn. of $a$ within $B$, $ud = \text{set of defns of } a \text{ in } in(B)$.

2. If there is an unambiguous defn of $a$ within $B$ prior to this use, then $ud = \{ \text{most recent unambiguous defn.} \}$

3. In addition, if there are ambiguous definitions of $a$, then add all those for which no unambiguous defn. lies between it and the current use to the *ud* chain.
Available expressions

**Definition:** $x+y$ is *available* at point $p$ if every path from the initial node to $p$ evaluates $x+y$ and there are no subsequent assignments to $x$ or $y$ after the last evaluation.

**Kill:** $x+y$ is killed if $x$ or $y$ is assigned and $x+y$ is not subsequently recomputed.

**Gen:** $x+y$ is generated if the value of $x+y$ is computed and $x, y$ are not subsequently redefined.
Calculating AEs within a block

1. Initialize $A \leftarrow \emptyset$.

2. Consider each assignment $x = y+z$ within the block in turn:
   (i) add $y+z$ to $A$
   (ii) delete any expression involving $x$ from $A$

3. At the end, $gen = A$

$kill$ = set of all expressions $y+z$ s.t. $y$ or $z$ is defined within the block and $y+z \notin A$
Available expressions

**Input:** flow graph with $e_{\text{gen}}$, $e_{\text{kill}}$ sets for each BB

**Output:** $in$, $out$ sets for each block

**Method:**

1. Initialize $in(B_1) \leftarrow \emptyset$, $out(B_1) \leftarrow e_{\text{gen}}(B_1)$ ($B_1$ - initial node)
2. For each $B \neq B_1$, initialize $out(B) \leftarrow \mathcal{U} - e_{\text{kill}}(B)$
3. While changes occur:
   
   for each $B \neq B_1$
   
   $in(B) \leftarrow \bigcap_P out(P)$ where $P$ - predecessor of $B$
   
   $out(B) \leftarrow e_{\text{gen}}(B) \cup (in(B) - e_{\text{kill}}(B))$
Live variables

\( \text{in}(B) \) – set of variables live at the initial point of \( B \)

\( \text{out}(B) \) – set of variables live at the end point of \( B \)

\( \text{def}(B) \) – set of variables definitely assigned values in \( B \)
  prior to any use in \( B \)

\( \text{use}(B) \) – set of variables whose values may be used in \( B \)
  prior to any definition of the variable
**Live variables**

**Input:** flow graph with \( \text{def}, \text{use} \) sets for each BB

**Output:** \( \text{in}, \text{out} \) sets for each block

**Method:**

1. For each \( B \), initialize
   \[
   \text{in}(B) \leftarrow \text{use}(B)
   \]

2. While changes occur:
   for each block \( B \)
   \[
   \text{out}(B) \leftarrow \bigcup_S \text{in}(S) \quad \text{where } S \text{ - successor of } B
   \]
   \[
   \text{in}(B) \leftarrow \text{use}(B) \cup (\text{out}(B) - \text{def}(B))
   \]

**D-U chain:** \( \text{du-chain} \) for a variable \( x \) at a given point \( p \) is the set of uses \( s \) of the variable s.t. there is a path from \( p \) to \( s \) that does not redefine \( x \)
Common subexpression elimination

Input: flow graph with available expression information

Output: revised flow graph

Method:
For each stmt $s$ \[ x = y + z \] s.t. $y + z$ is available at the beginning of $s$’ block and $y$, $z$ are not defined prior to $s$ within the block:
1. follow flow graph edges backwards until a block containing an evaluation \[ w = y + z \] is found
2. create a new variable $u$
3. replace \[ w = y + z \] by \[ u = y + z \quad w = u \]
4. replace \[ x = y + z \] by \[ x = u \]
Common subexpression elimination

- Should be used with copy propagation
- May need multiple passes for best effect
Copy propagation

**Principle:** Given $s: \boxed{x=y}$, $y$ can be substituted for $x$ in all uses $u$ of $x$ if

1. $s$ is the only definition of $x$ reaching $u$

2. on every path from $s$ to $u$ (including paths that go through $u$ several times but not more than once through $s$) there are no assignments to $y$
Copy propagation

\[ in(B) \quad – \quad \text{set of copies } s: \boxed{x=y} \text{ s.t. every path from initial node to beginning of } B \text{ contains } s \text{ and there are no assignments to } x, y \text{ after the last occurrence of } s \]

\[ out(B) \quad – \quad \text{as above} \]

\[ gen(B) \quad – \quad \text{all copies } s: \boxed{x=y} \text{ in } B \text{ s.t. there are no assignments to } y \text{ within } B \text{ after } s \]

\[ kill(B) \quad – \quad \text{all copies } s: \boxed{x=y} \text{ s.t. } x \text{ or } y \text{ is assigned in } B \text{ and } s \not\in B \]
Copy propagation

\[ \text{in}(B) \] — set of copies \( s: \begin{array}{c} x = y \\ \text{s.t.} \end{array} \) s.t. every path from initial node to beginning of \( B \) contains \( s \) and there are no assignments to \( x, y \) after the last occurrence of \( s \)

\[ \text{out}(B) \] — as above

\[ \text{gen}(B) \] — all copies \( s: \begin{array}{c} x = y \\ \text{s.t.} \end{array} \) in \( B \) s.t. there are no assignments to \( y \) within \( B \) after \( s \)

\[ \text{kill}(B) \] — all copies \( s: \begin{array}{c} x = y \\ \text{s.t.} \end{array} \) s.t. \( x \) or \( y \) is assigned in \( B \) and \( s \not\in B \)

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

\[
in(B_1) = \emptyset
\]

\[
in(B) = \bigcap_{P} \text{out}(P) \text{ where } P \text{- pred. of } B, \quad B \neq B_1
\]
Copy propagation

Input: flow graph with ud chains, du chains, \( \text{in}(B) \)

Output: revised flow graph

Method: For each \( s : x = y \) do:

1. Determine the uses of \( x \) reached by this definition.

2. Determine whether for every use of \( x \) in (1), \( s \in \text{in}(B) \) for the block containing the use and no definitions of \( x, y \) occur prior to this use within \( B \).

3. If \( s \) meets the conditions in (2), remove \( s \) and replace all uses of \( x \) found in (1) by \( y \).
**Loops**

**Dominator:** a node $d$ dominates node $n$ if every path from the initial node to $n$ goes through $d$

**Back edge:** an edge $a \rightarrow b$ in a flow graph is a back edge if $b$ dominates $a$

**Natural loop:** given a back edge $n \rightarrow d$, the natural loop for this edge consists of $d$ along with all nodes from which we can reach $n$ without going through $d$

**Header:** the node that dominates all other nodes in a loop

**Pre-header:** Given a loop $L$ with header $h$:
1. create an empty block $p$;
2. make $h$ the only successor of $p$;
3. all edges which entered $h$ from outside $L$ are changed to point to $p$ (edges to $h$ from inside $L$ are not changed).
**Loop-invariant computations**

**Input:** loop $L$ + ud chains for statements in the loop  
**Output:** statements that perform loop-invariant computations  

**Method:**

1. Mark “invariant” any statement whose operands are all either constants or have all their reaching definitions outside $L$.
2. Repeat until no further changes: mark “invariant” any statement that is not already marked and all of whose operands satisfy one of the following conditions:  
   (i) the operand is a constant  
   (ii) the operand has all its reaching definitions outside $L$  
   (iii) the operand has exactly one reaching definition, and that definition is a statement in $L$ that has been marked invariant
Conditions for moving \( x = y + z \)

The block containing \( s \) must dominate all exit nodes of the loop (i.e. nodes with a successor not in the loop).
There is no other assignment to \( x \) in the loop.

(usually satisfied by temporaries)
Conditions for moving $x = y + z$

No use of $x$ in the loop is reached by any definition of $x$ other than $s$.

(usually satisfied by temporaries)
**Code motion**

**Input:** loop $L$ with ud chains and dominator information  

**Output:** revised loop with a preheader  

**Method:**  
1. Find loop-invariant computations (see above).
2. For each statement $s$ defining $x$ found in (1), check whether:  
   (i) it is in a block that dominates all exits of $L$  
   (ii) $x$ is not defined elsewhere in $L$  
   (iii) all uses in $L$ of $x$ can only be reached by the definition of $x$ in statement $s$  
3. Move all stmts $s$ that satisfy (2) to the preheader in the order in which they were found in (1) provided any operands of $s$ that are defined in loop $L$ have also had their definitions moved to the preheader.