Common subexpression elimination (CSE) is a classic optimization that replaces redundantly computed expressions with a variable containing the value of the expression. It works on a general CFG. An expression is a common subexpression at a given node if it is computed on every path leading from the start to this node, and none of its operands have been changed on any path from the dominating node. Therefore, the value previously computed on the path leading to this node can be reused.

For example, in Figure 1, the expression \(a+1\) is a common subexpression at the bottom node. Therefore, it can be saved into a new temporary \(t\) in the top node, and this temporary can be used in the bottom one.

It is worth noting that CSE can make code slower, because it may increase the number of live variables, causing spilling. If there is a lot of register pressure, the reverse transformation, forward substitution, may improve performance. Forward substitution copies expressions forward when it is cheaper to recompute them than to save them in a variable.

### 1.1 Available expressions analysis

An expression is available if it has been computed in all paths leading to this node and its operands have not been redefined. The available expressions analysis finds the set of such expressions. Implicitly, each such expression is tagged with the location in the CFG that it comes from, to allow the CSE transformation to be done.

Available expressions is a forward analysis. We define \(\text{out}(n)\) to be the set of available expressions on edges leaving node \(n\). An expression is available if it was evaluated at \(n\), or was available on all edges entering \(n\), and it was not killed by \(n\):

\[
\text{in}(n) = \bigcap_{n' < n} \text{out}(n')
\]

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

Note that unlike in some previous dataflow equations, \(\text{kill}(n)\) is done after the union with \(\text{exprs}(n)\) because a node can kill its own expressions; consider the statement \(x = x + 1\), for example.

Therefore, dataflow values are sets of expressions ordered by \(\subseteq\); the meet operator is set intersection \((\cap)\), and the top value is the set of all expressions, usually implemented as a special value that acts as the identity for \(\cap\).

![Figure 1: Common subexpression elimination](image)
The expressions evaluated and killed by a node, \( \text{exprs}(n) \), are summarized in the following table. Note that we try to include memory operands as expressions subject to CSE, because replacing memory accesses with register accesses is a useful optimization.

<table>
<thead>
<tr>
<th>( n )</th>
<th>( \text{exprs}(n) )</th>
<th>( \text{kill}(n) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x = e )</td>
<td>( e ) and all subexpressions of ( e )</td>
<td>all expressions containing ( x )</td>
</tr>
<tr>
<td>( [e_1] = [e_2] )</td>
<td>( [e_2], [e_1] ), and subexpressions</td>
<td>all expressions containing ( [e'] ) that can alias ( [e_1] )</td>
</tr>
<tr>
<td>( x = f(e) )</td>
<td>( e ) and subexpressions</td>
<td>expressions containing ( x ) and expressions ( [e'] ) that could be changed by function call to ( f )</td>
</tr>
<tr>
<td>( \text{if } e )</td>
<td>( e ) and subexpressions</td>
<td>( \emptyset )</td>
</tr>
</tbody>
</table>

If a node \( n \) computes an expression \( e \) that is used and available in other nodes, the optimization proceeds as follows:

1. In the node that the available expression came from, add a computation \( t = e \), and replace the use of \( e \) with \( t \).
2. Replace expression \( e \) in other nodes where it is used and available with \( t \).

CSE can work well with copy propagation. For example, the variable \( b \) in the above code may become dead after copy propagation. However, CSE plus copy propagation can enable more CSE, because CSE only recognizes syntactically identical expressions as the same. Copy propagation can make semantically identical expressions look the same through its renaming of variables. It is possible to generalize Available Expressions to keep track of equalities more semantically, though this makes the analysis much more complex.

## 2 Cascaded dataflow analyses

Some analyses lead to optimizations that enable more optimization. For example, CSE plus copy propagation can lead to more CSE. To avoid rerunning analyses after optimizations, we can design analyses to take into account optimizations that will be performed. Compared to CSE, local variable numbering is also a cascaded analysis, at least within the scope of an extended basic block.

A more concrete example is changing live variable analysis to take into account the removal of dead code. The intuition is that with live variable analysis as we’ve defined it, the variable \( x \) is live before the node \( y = x + 1 \). But suppose \( y \) is not live after the node. Then \( x \) is not really needed before the node, because dead code removal will eliminate this node and its use of \( x \).

We reformulate live variable analysis with the principle that a variable is “dead until proven otherwise” as follows. Recall that the old transfer function \( F_n(\ell) \) is defined as:

\[
F_n(\ell) = \text{use}(n) \cup (\ell - \text{def}(n))
\]

The modified analysis uses this definition if node \( n \) may be needed: if \( \text{def}(n) \cap \ell = \emptyset \), or if the node updates memory or calls a function that might have visible side effects. If the node \( n \) is not determined to be needed, we treat \( \text{use}(n) \) as if it were empty:

\[
F_n(\ell) = \ell - \text{def}(n)
\]

This modified transfer function is still monotonic, and it is also distributive if \( |\text{def}(n)| \leq 1 \).

## 3 Partial redundancy elimination

CSE eliminates computation of fully redundant expressions: those computed on all paths leading to a node. Partially redundant expressions are those computed at least twice along some path, but not necessarily all paths. Partial redundancy elimination (PRE) eliminates these partially redundant expressions. PRE subsumes CSE and loop-invariant code motion.

Figure 2 shows an example of PRE. The computation \( b + c \) is redundant along some paths but not others. To make it fully redundant, we place computation of \( b + c \) onto earlier edges so that it has always been computed at each point where it is needed.
3.1 Lazy code motion

The idea of lazy code motion is to eliminate all redundant computations while avoiding creating any unnecessary computation: computations are moved earlier in the CFG. Further, we want to make sure that although the computations are moved earlier in the CFG, they are postponed as long as possible, to avoid creating register pressure.

The approach is to first identify candidate locations where the partially redundant expression could have been moved in order to make it fully redundant, without creating extra computations. Then among these candidates we choose the one that comes latest along each path that needs it.

3.2 Anticipated expressions

The anticipated expressions analysis (also known as very busy expressions) find expressions that are needed along every path leaving a given node. If an expression is needed along every path leaving the node, then there can be no wasted computation if the expression is moved to that node.

This is a backward analysis, in which the dataflow values are sets of expressions and the meet operator is $\cap$.

Once we know the anticipated expressions at each node, we tentatively place computations of these expressions and use an available expressions analysis to find expressions that are fully redundant under the assumption that the anticipated expressions are computed everywhere anticipated. These fully redundant expressions are the expressions to which we can apply the PRE optimization.

3.3 Postponable expressions

At this point we know some set of nodes where the expression can be moved, and we know where it is used. We need to pick a set of edges that separate these two parts of the CFG, and put the computation of the expression on those edges. We want to postpone the computation as long as possible. The postponable expressions analysis finds expressions $e$ that are anticipated at program point $p$ but not yet used: every path from the start to $p$ contains an anticipation of $e$ and no use before $p$. This is a forward analysis with meet operator $\cap$.

Once postponable expressions have been computed, certain edges form a frontier where the expression transitions from postponable to not postponable. It is on these edges that the new node computing the expressions is placed.

![Diagram of partial redundancy elimination]

Figure 2: Partial redundancy elimination