CSC410 Assignment 2 Question 2

Liang Chen 1005735126
Joshua Han 1005109669
Yiqi Zhi 1004892725

2.1. Solution:

Recall the definition of available expression analysis:
An expression e is available at point p if every path leading to p contains a prior definition (i.e. its value is computed) of e and e is not killed (i.e. its operands are not redefined) between that definition and point b.

We used a forward must analysis and gained the largest fixed point output:

Data flow facts P(exp): power set of all expressions in this program
Lattice L: (P(exp), ∩)
Direction: forward
Partial order: $\sqsubseteq = \subseteq$
Initialization: OUT[B] = set of all expressions in program (except for entry node)
Boundary: OUT[Entry] = Ø
Transfer function F: P(exp) -> P(exp)
(assuming one basic block B contains a single statement)

OUT[B] = F(IN[B])
OUT[B] = (IN[B] – Kill[B]) ∪ Gen[B]
IN[B] = $\cap_{P\ predecessors\ of\ B}\ OUT[P]$

Where

IN[B]: available expression at the entry of block B
OUT[B]: available expression at the exit of block B
Used[B]: expressions used in the block statement
Written[B]: variable written in the block's assignment statement
Kill[B]: All expressions containing written[B]
Gen[B]: expressions used in the block and not containing the written variable
OUT[P]: available expression at the exit of block B's predecessors.

(Atomic values will not be taken into consideration in our analysis)
Self-designed test case 1:

```
1  l1:  z = a + b;
2  l2:  y = a * b;
3  l3:     while (((y > (a + b) ) && 1) && (!1))
4     {
5  l4:    a = a + 1;
6  l5:    x = a + b;
7  l6:    b = 1;
8  l7:    m = m + (a + b);
9  }
```

IN[Entry] = none, OUT[Entry] = none
IN[L1] = {}; OUT[L1] = {a+b}
IN[L2] = {a+b}; OUT[L2] = {a+b; a*b}
IN[L3] = {a+b}; OUT[L3] = {a+b; y>a+b; !1; (y>a+b)&&1; ((y>a+b)&&1)&&!1}
IN[L4] = {a+b; y>a+b; !1; (y>a+b)&&1; ((y>a+b)&&1)&&!1}; OUT[L4] ={!1}
IN[L5] = {!1}; OUT[L5] = {a+b;!1}
IN[L6] = {a+b;!1}; OUT[L6] = {!1}
IN[L7] = {!1}; OUT[L7] = {!1, a+b}
IN[Exit] = {a+b; y>a+b; !1; (y>a+b)&&1; ((y>a+b)&&1)&&!1};
OUT[Exit] = {a+b; y>a+b; !1; (y>a+b)&&1; ((y>a+b)&&1)&&!1};

Self-designed test case 2:

```
1   l1: a = 0+3;
2   l2: if (a != b) {
3   l3:      x = b - a;
4   l4:      y = a - b;
5     } else {
6   l5:     y = b - a;
7   l6:        a = 0;
8   l7:        x = a - b;
9     }
10  l8: while(x < 0) {
11  l9: x = y - 1;
12    }
```

Result:

```
set()
set()
set()
{<BinaryExpression: 0 + 3>}
{<BinaryExpression: 0 + 3>}
{<BinaryExpression: a != b>, <BinaryExpression: 0 + 3>}
{<BinaryExpression: a != b>, <BinaryExpression: 0 + 3>}
{<BinaryExpression: a != b>, <BinaryExpression: b - a>, <BinaryExpression: 0 + 3>}
{<BinaryExpression: a != b>, <BinaryExpression: b - a>, <BinaryExpression: 0 + 3>}
{<BinaryExpression: a != b>, <BinaryExpression: a - b>, <BinaryExpression: b - a>, <BinaryExpression: 0 + 3>}
{<BinaryExpression: a != b>, <BinaryExpression: 0 + 3>}
{<BinaryExpression: a != b>, <BinaryExpression: b - a>, <BinaryExpression: 0 + 3>}
{<BinaryExpression: a != b>, <BinaryExpression: b - a>, <BinaryExpression: 0 + 3>}
{<BinaryExpression: 0 + 3>}
{<BinaryExpression: 0 + 3>}
{<BinaryExpression: a - b>, <BinaryExpression: 0 + 3>}
{<BinaryExpression: a - b>, <BinaryExpression: 0 + 3>}
{<BinaryExpression: a - b>, <BinaryExpression: x < 0>, <BinaryExpression: 0 + 3>}
{<BinaryExpression: a - b>, <BinaryExpression: x < 0>, <BinaryExpression: 0 + 3>}
{<BinaryExpression: y - 1>, <BinaryExpression: a - b>, <BinaryExpression: 0 + 3>}
{<BinaryExpression: a - b>, <BinaryExpression: x < 0>, <BinaryExpression: 0 + 3>}
{<BinaryExpression: a - b>, <BinaryExpression: x < 0>, <BinaryExpression: 0 + 3>}
```

2.2. Solution:

Definition: an expression is partially available at a program point ℓ if there is at least one control flow path to location ℓ along which the expression is computed, and none of its operands have been overwritten since. (may forward analysis with the least fixed point gained, which is overapproximation)

Data flow facts P(exp): power set of all expressions in this program
Lattice L: (P(exp), ∪)
Direction: forward
Partial order: $\sqsubseteq = \supseteq$
Initialization: OUT[B] = ∅
Boundary: OUT[Entry] = ∅
Transfer function F: P(exp) -> P(exp)
(assuming one basic block B contains a single statement)

OUT[B] = F(IN[B])
OUT[B] = (IN[B] – Kill[B]) ∪ Gen[B]
IN[B] = $\cup_{P\ predecessors\ of\ B}\ OUT[P]$

Where

IN[B]: Partially available expression at the entry of block B
OUT[B]: Partially available expression at the exit of block B
Used[B]: expressions used in the block statement
Written[B]: variable written in the block's assignment statement
Kill[B]: All expressions containing written[B]
Gen[B]: expressions used in the block and not containing the written variable
OUT[P]: Partially available expression at the exit of block B's predecessors.

Self-designed test case 1:

```
1  l1:  z = a + b;
2  l2:  y = a * b;
3  l3:     while (((y > (a + b) ) && 1) && (!1))
4     {
5  l4:    a = a + 1;
6  l5:    x = a + b;
7  l6:    b = 1;
8  l7:    m = m + (a + b);
9  }
```

IN[Entry] = none, OUT[Entry] = none
IN[L1] = {}; OUT[L1] = {a+b}
IN[L2] = {a+b}; OUT[L2] = {a+b; a*b}
IN[L3] = {a+b; a*b; !1}; OUT[L3] = {a+b; a*b y>a+b; !1; (y>a+b)&&1; ((y>a+b)&&1)&&!1}
IN[L4] = {a+b; a*b; y>a+b; !1; (y>a+b)&&1; ((y>a+b)&&1)&&!1}; OUT[L4] ={!1}
IN[L5] = {!1}; OUT[L5] = {a+b;!1}

IN[L6] = {a+b;!1}; OUT[L6] = {!1}
IN[L7] = {!1}; OUT[L7] = {!1, a+b}
IN[Exit] = {a+b; y>a+b; !1; (y>a+b)&&1; ((y>a+b)&&1)&&!1};
OUT[Exit] = {a+b; y>a+b; !1; (y>a+b)&&1; ((y>a+b)&&1)&&!1};

Self-designed test case 2:

```
1   l1: a = 0+3;
2   l2: if (a != b) {
3   l3:      x = b - a;
4   l4:      y = a - b;
5     } else {
6   l5:     y = b - a;
7   l6:        a = 0;
8   l7:        x = a - b;
9     }
10  l8: while(x < 0) {
11  l9: x = y - 1;
12    }
```

Result:

```
set()
set()
set()
{<BinaryExpression: 0 + 3>}
{<BinaryExpression: 0 + 3>}
{<BinaryExpression: 0 + 3>, <BinaryExpression: a != b>}
{<BinaryExpression: 0 + 3>, <BinaryExpression: a != b>}
{<BinaryExpression: 0 + 3>, <BinaryExpression: b - a>, <BinaryExpression: a != b>}
{<BinaryExpression: 0 + 3>, <BinaryExpression: b - a>, <BinaryExpression: a != b>}
{<BinaryExpression: 0 + 3>, <BinaryExpression: b - a>, <BinaryExpression: a - b>, <BinaryExpression: a != b>}
{<BinaryExpression: 0 + 3>, <BinaryExpression: a != b>}
{<BinaryExpression: 0 + 3>, <BinaryExpression: b - a>, <BinaryExpression: a != b>}
{<BinaryExpression: 0 + 3>, <BinaryExpression: b - a>, <BinaryExpression: a != b>}
{<BinaryExpression: 0 + 3>}
{<BinaryExpression: 0 + 3>}
{<BinaryExpression: 0 + 3>, <BinaryExpression: a - b>}
{<BinaryExpression: y - 1>, <BinaryExpression: b - a>, <BinaryExpression: a != b>, <BinaryExpression: a - b>, <BinaryExpression: 0 + 3>}
{<BinaryExpression: y - 1>, <BinaryExpression: b - a>, <BinaryExpression: x < 0>, <BinaryExpression: a - b>, <BinaryExpression: a != b>, <BinaryExpression: 0 + 3>}
{<BinaryExpression: y - 1>, <BinaryExpression: b - a>, <BinaryExpression: x < 0>, <BinaryExpression: a - b>, <BinaryExpression: a != b>, <BinaryExpression: 0 + 3>}
{<BinaryExpression: y - 1>, <BinaryExpression: b - a>, <BinaryExpression: a - b>, <BinaryExpression: a != b>, <BinaryExpression: 0 + 3>}
{<BinaryExpression: y - 1>, <BinaryExpression: b - a>, <BinaryExpression: x < 0>, <BinaryExpression: a - b>, <BinaryExpression: a != b>, <BinaryExpression: 0 + 3>}
{<BinaryExpression: y - 1>, <BinaryExpression: b - a>, <BinaryExpression: x < 0>, <BinaryExpression: a - b>, <BinaryExpression: a != b>, <BinaryExpression: 0 + 3>}
```

2.3 Solution

Definition: An expression can be *potentially placed* after line ℓ″ if it is partially available at ℓ′ and there is a control flow path from ℓ″ to ℓ′.

To create a dataflow analysis that could give us the potential placements of each expression used in a program, we define a lattice L such that:

> Data flow facts: $\mathcal{P}$(Exp) = power set of all expressions in this program
> Lattice L: ($\mathcal{P}$(Exp), ∩, ∪) with finite height
> Direction: Forward and Backward
> Partial Order: ⊆ and ⊇
> Initialization: IN[B] = ∅ for each node B in the control flow graph

The transfer function of this lattice is defined with the following equations for the *out* and *in* sets of this Potential Placement Analysis, where block B represents a node in the control flow graph:

$$OUT[B] = (\cap_{S \in succ(B)} AvailableExp_{IN}[S]) \cup (\cap_{S \in succ(B)} IN[S])$$
$$IN[B] = ([OUT[B] \cap (PAvailableExp_{IN}[B] \cup PAvailableExp_{OUT}[B])] - Kill[B]) \cup Gen[B]$$

where

- $AvailableExp_{IN}[X]$ is the *in* set at block X from the Available Expressions Analysis.
- $PAvailableExp_{IN}[X]$ is the *in* set at node X from the Partially Available Expressions Analysis.
- $PAvailableExp_{OUT}[X]$ is the *out* set at node X from the Partially Available Expressions Analysis.
- $Gen[X]$ is the set of expressions in the statement at node X (which does not contain the variable written to at this node.
- $Kill[X]$ is the set of expressions that contain the variable written to at this node.

The intuition behind this transfer function is that we check where expressions are computed and we move those expressions up if those expressions are still partially available at that location. We stop moving expressions up the control flow graph if

we reach a region of the control flow graph where those expressions are not partially available.

An expression belongs to the *out* set at a node X if we can add a new computation of that expression to a fresh variable directly after the statement in node X. An expression belongs to the *in* set at a node X if we can add a new computation of that expression to a fresh variable directly before the statement in node X.

If an expression E can be placed at the end of an arbitrary node X (the expression is in the *out* set of X), we can move E before X if:

- $E \in PAvailableExp_{IN}[X]$ and $E \in PAvailableExp_{OUT}[X]$: At this location E is partially available, which means there is a path from this location to a computation of E where the value of E is not modified.
- $E \notin PAvailableExp_{IN}[X]$ and $E \in PAvailableExp_{OUT}[X]$ and $E \in Gen[X]$: At this location E is partially available only at the end. This means that either this location is the first computation of E or the statement at X modified the value of E. We can place E before X if and only if E is computed at this location ($E \in Gen[X]$).

For the other cases:

- $E \in PAvailableExp_{IN}[X]$ and $E \notin PAvailableExp_{OUT}[X]$: We do not move E before X since the expression E was modified in the statement at X ($E \in Kill[X]$).
- $E \notin PAvailableExp_{IN}[X]$ and $E \notin PAvailableExp_{OUT}[X]$: We do not move E before X since at this point we reached a region where E is not partially available.

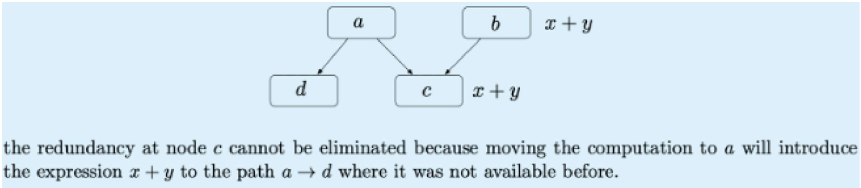
This is why the $IN[X]$ function uses this term:
$OUT[B] \cap (PAvailableExp_{IN}[B] \cup PAvailableExp_{OUT}[B])$.
We take the union with $Gen[X]$ to move the expression up in the second case.
We take the difference with $Kill[X]$ to remove the expression from the *in* set in the third case.

If we have decided that an expression E can be placed at the beginning of a node X, we determine which predecessors to move E to the end of by using Available Expression Analysis. In the $OUT[X]$ function, we take the intersection of the *in* set of Available Expression Analysis for each successor of X to prevent our Potential

Placement Analysis from doing harm. For each successor S of X, if $E \in AvailableExp_{IN}[S]$ then every control flow path to S has computed E and the value of E has not been modified before reaching S. This way, we can ensure that when moving E up from S to X, E will not become known to a path where E was previously unknown. This is why we take the union of $\cap_{S \in succ(B)} AvailableExp_{IN}[S]$ with $\cap_{S \in succ(B)} IN[S]$ in the $OUT[X]$.

PS:
Generally speaking, there exists a more complicated algorithm that optimizes partial redundancy elimination, which contains four analyses and the detailed steps are shown below:

- Step 1: Find anticipated expressions at each node (i.e. very busy expression). An expression is anticipated at point p if all paths leaving p eventually compute the expression from the values of the operands that are available at p. (focus on expressions that are anticipated at the entry of each block; if it is anticipated at the exit of the block, I only know it will be busy in the entry of its successors but nothing else). This is because copies of an expression must be placed only at the point where this expression is anticipated to avoid extra operations - just as in the example



the redundancy at node c cannot be eliminated because moving the computation to a will introduce the expression $x + y$ to the path $a \to d$ where it was not available before.

Here is the table of anticipated expressions analysis facts/direction/transfer function:

Domain	Set of all expressions in program
Direction	Backward
Transfer function	$AntIn[B] = Gen[B] \cup (AntOut[B] - Kill[B])$ $AntOut[B] = \cap_{S \text{ are sucessors of } B} AntIn[S]$
Boundary	$AntIn[Exit] = \emptyset$
Initialization	$AntIn[B] = U$

- Step 2: Find available expression
  Assume we calculate expression e whenever it is anticipated and redefine available expressions here: the expression e is available at point p if expression e has been anticipated but not subsequently killed on all paths reaching p.

Since the availability of e at p depends on if expression e has been anticipated on ALL (i.e. every) path reaching to point p (i.e. predecessors), we use forward intersection iteration and construct transfer function:
Set of available expressions at the entry of a block B is the intersection of sets of available expressions at the exit of ALL of B's predecessors;
If an expression is available at the exit of block B, it is either 1) available at the entry of block B and was not killed by this block; 2) it is anticipated at the entry of block B and was not killed by this block

Domain	Set of all expressions in program
Direction	Forward
Transfer function	$AvOut[B] = (AvIn[B] \cup AntIn[B]) - Kill[B]$ $AvIn[B] = \cap_{P\ are\ predesesors\ of\ B}\ AvOUT[P]$
Boundary	$AvOut[Entry] = \emptyset$
Initialization	$AvOUT[B] = U$

We actually want to insert expression at "anticipated" but not "will be available" blocks, which is the frontier of e's anticipated places:

$$Earliest[B] = AntIn[B] - AvaIn[B]$$

- Step 3: find the postponable expressions
An expression is postponable at a program point p if all paths leading to p have seen the earliest placement of e but not a subsequent use before reaching point p

Domain	Set of all expressions in program
Direction	Forward
Transfer function	$PostOut[B] = (Earliest[B] \cup PostIn[B]) - Use[B]$ $PostIn[B] = \cap_{P\ are\ predesesors\ of\ B}\ PostOut[P]$
Boundary	$PostOut[Entry] = \emptyset$
Initialization	$PostOUT[B] = U$

$$Latest[B] = (Earliest[B] \cup PostIn[B]) \cap (Use[B] \cap \neg(\cap_{S\ are\ Succesors\ of\ B}\ (Earliest[S] \cup PostIn[S])))$$

- Step 4: find the used expressions.
We also have to verify if a temporary variable introduced is used anywhere else except for the block it is in. We would say an expression is used in point p if there exists a path from p that used this expression before it is recalculated.
We use backward analysis to get UsedOut[B]

Domain	Set of all expressions in program
Direction	Backward
Transfer function	$UsedIn[B] = (Use[B] \cup UsedOut[B]) - latest[B]$ $UsedOut[B] = \cup_{s\ are\ successors\ of\ B}\ UsedIn[S]$
Boundary	$UsedIn[Exit] = \emptyset$
Initialization	$UsedIn[B] = \emptyset$

In summary, in the optimized PRE analysis we follow the steps :

(1) create a temporary variable for each expression in the program;

(2) Execute all four analyses mentioned above and get the fixed point results;

(3) For each basic block, if an expression e is in Latest[B] ∩ Used[B], it should be added to the entry of block B;

(4) If an expression is in use[B] ∩ (negation of Latest[B] ∪ UsedOut[B]), we use temporarily variable of this expression to replace the expression in this block

2.4 Solution

Let $insert_\ell$ and $delete_\ell$ be sets defined by these equations for each node X in the control flow graph:

$$insert_\ell(X) = PotentialPlacement_{OUT}(X) - PotentialPlacement_{IN}(X)$$

$$delete_\ell(X) = Used[X] - [PotentialPlacement_{OUT}(X) - PotentialPlacement_{IN}(X)]$$

where

- $Used[X]$ is the set of expressions computed at node X.
- $PotentialPlacement_{IN}(X)$ is the set of expressions that are in the *in* set of the Potential Placement Analysis at node X.
- $PotentialPlacement_{OUT}(X)$ is the set of expressions that are in the *out* set of the Potential Placement Analysis at node X.

If an expression E can be placed at both the beginning and end of the statement at node X then it means that there is a better location to place E if we move E up. We want to insert E at a node X where it can only be potentially placed at the end because at these locations there must be a successor of X that computes E. This is why we take

difference between the *out* set and the *in* set of the Potential Placement Analysis at node X when computing the *insert* set.

The *delete* set is defined so that we remove any expressions computed at node X if they are in the *insert* set. Otherwise, we remove that expression from node X since it has been inserted somewhere else.