Function Information Pass

This pass collects information about the functions in a module. This pass mostly uses the member functions of the **Function** class to gather different information about the functions. This way we get both the name and the number of arguments for each function.

The information about the number of call sites for each function uses the member function of the **Instruction** class. The number of call sites are determined at the Initialization function for the module under consideration. During the initialization, we count the number of occurrences of a function.

The instruction count and the basic block counts of a function are populated in a runOnFunction().

The member functions used for collecting different information are marked in the source code with comments

Local Optimization Pass

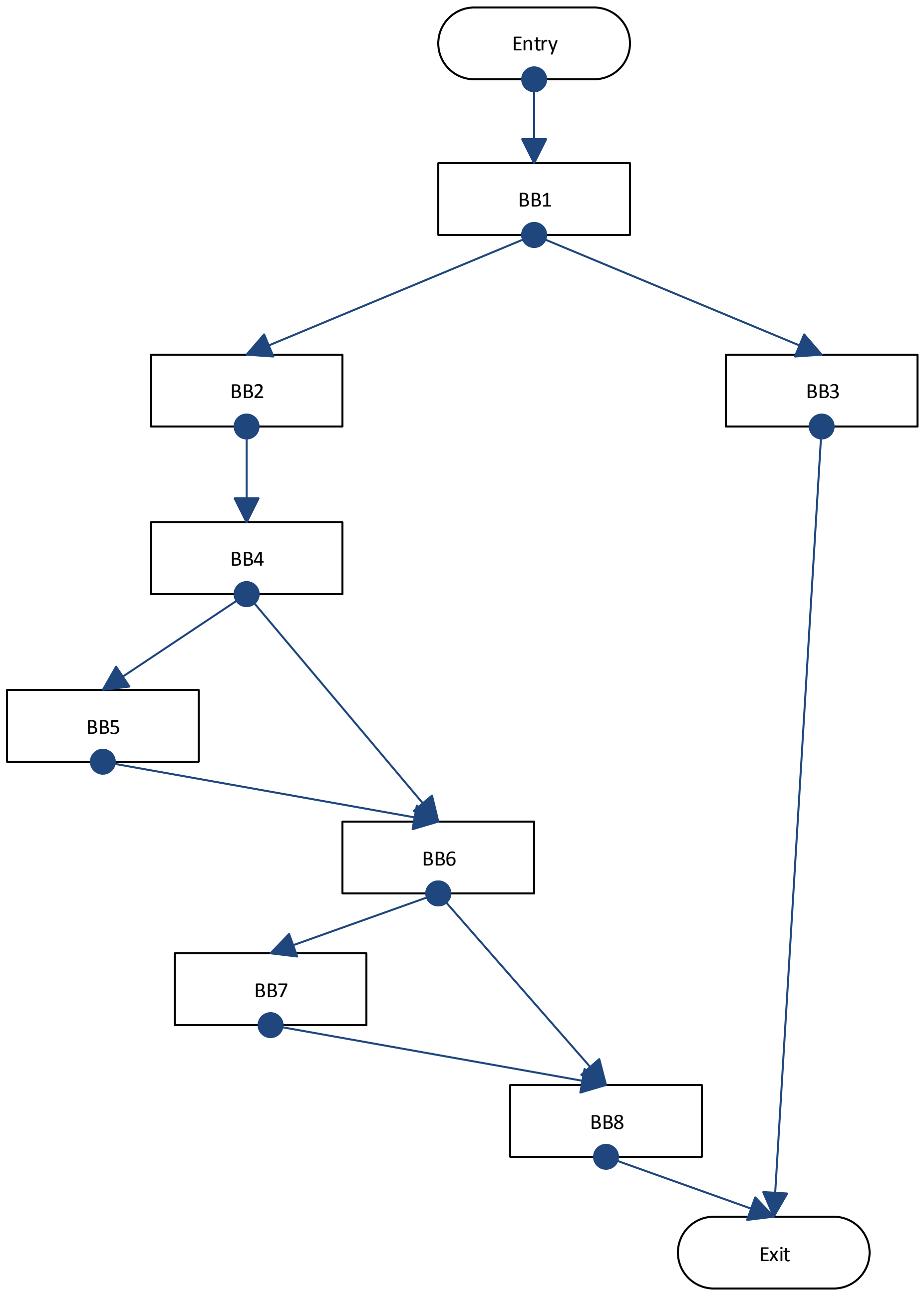
For the LocalOpts pass, it is implemented as a BasicBlockPass. For each block given it scans through all the instructions forward and decides that if a constant fold, algebraic identity or strength reduction can be done on that instruction respectively. If the first two can be done, then the instruction would be removed from the instruction list and be replaced by a constant value or operand value while for strength reduction the instruction would be replaced by another instruction of simpler strength. The iterator method used here is a self written one that first gets the next element in line and then process the current element so that in case the current element got removed the list can still be further traversed. All the optimizations are of instruction level and are only processed upon binary operations with two operands. The algebraic identity optimization is performed when one of the following are met:
1)Adding a variable with 0
2)Subtracting a variable by 0 or by itself
3)Multiplying by 0 or 1
4)Dividing by 1 or itself or dividing 0 (Including remainder operations)   5)Shifting by 0

6)Anding and Xoring with 0 or by itself
7)Oring with all ones or itself

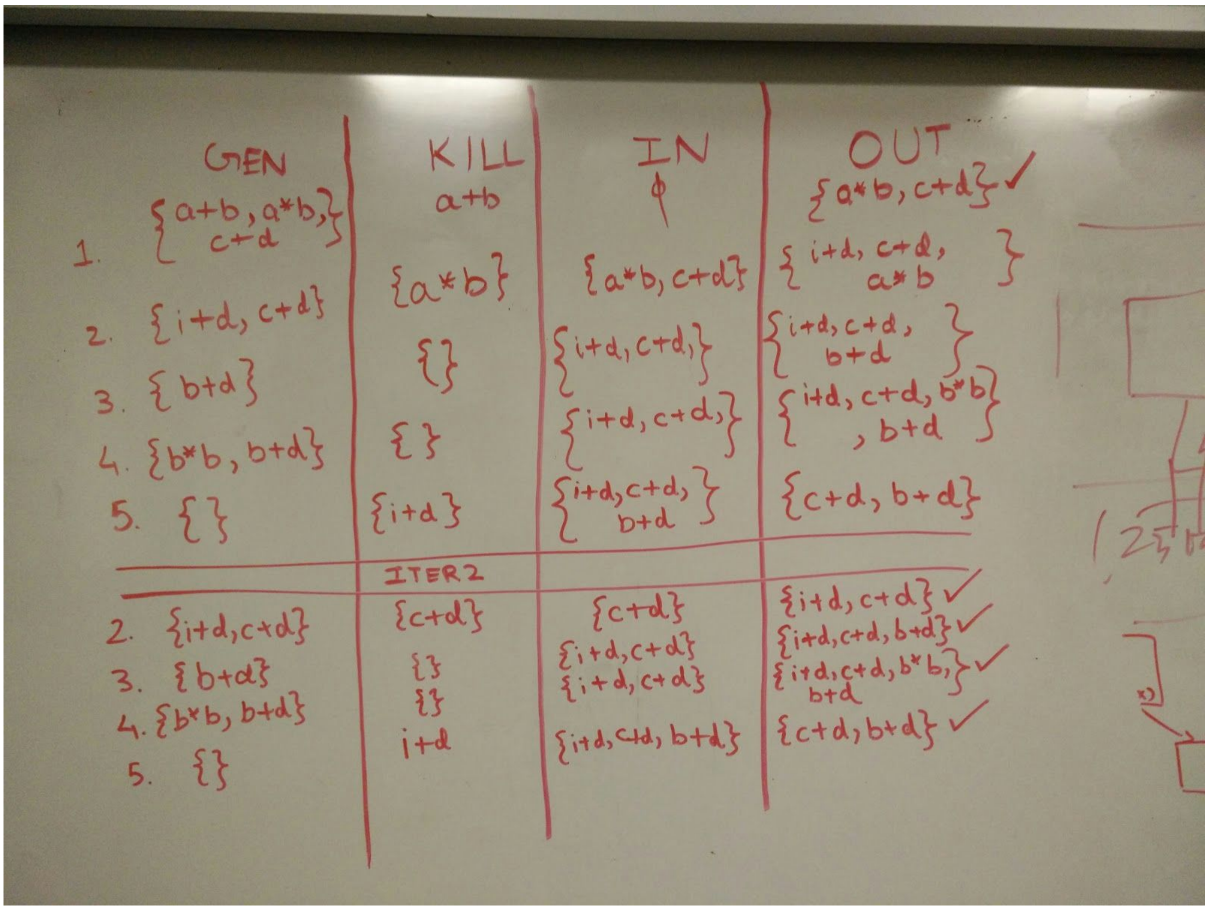
The constant folding optimization is performed when both of the operands of the instruction are constant and that the instruction is not a floating point instruction. The Strength reduction optimization is performed when multiplying or dividing by a polynomial of two.

3.1 CFG Basics

```
BB1            x = 5
               y = 8
               z = 234
               if ( x < z ) { goto L1 )
BB2            y = 89
               goto L2
BB3      L1:   z = 65
               return z
BB4      L2:   y = x + 1
               if ( z < x ) { goto L3 }
BB5            x = 25
BB6      L3:   y = x + z
               switch (y) { 334: goto L4 | default: goto L5 }
BB7      L4:   print("failure")
BB8      L5:   y = 65
               return y
```



Q 3.2)



	GEN	KILL	IN	OUT
1.	{a+b, a*b, c+d}	a+b	ϕ	{a*b, c+d} ✓
2.	{i+d, c+d}	{a*b}	{a*b, c+d}	{i+d, c+d, a*b}
3.	{b+d}	{}	{i+d, c+d,}	{i+d, c+d, b+d}
4.	{b*b, b+d}	{}	{i+d, c+d,}	{i+d, c+d, b*b , b+d}
5.	{}	{i+d}	{i+d, c+d, b+d}	{c+d, b+d}
		ITER2		
2.	{i+d, c+d}	{c+d}	{c+d}	{i+d, c+d} ✓
3.	{b+d}	{}	{i+d, c+d}	{i+d, c+d, b+d} ✓
4.	{b*b, b+d}	{}	{i+d, c+d}	{i+d, c+d, b*b, b+d} ✓
5.	{}	i+d	{i+d, c+d, b+d}	{c+d, b+d} ✓

The OUT variables with the tick mark beside them are the final values of the basic blocks

Q 3.3) Faint Analysis

1) We call the set of elements for this analysis as Used Variables (UV). A variable will be termed as a used variable if either of the following conditions hold:
   - The variable must be used in a routine that obviously interacts with the outside world (eg. printing to console), is used to determine the control flow (eg. loop variables) or if it is certain that the value will be used at the end of the code segment (eg. returning a value)
   - The variable is used to define a variable that is a Used Variable

2) The direction of the analysis will be backwards. This will allow faster convergence (explained later)

3) The Transfer function is defined as:

   In[b] = {Uses*} U Out[b]

   Where Members of the Uses set are a member of the Uses set iff the variable defined by the use is a member of the UV set.*
   Additionally, uses as mentioned in the first point of section 1

   Example) In the expression,

   z = x + y

   x and y will be in the Uses set if z is in the UV set*

   In the expression,

   incrementGlobalCounterByX(x)

   x will be in the Uses set*

   In the expression,

   if(x>53)

   x will be in the Uses set*

   In the expression,

   return z

   z will be in the Uses set*

   NOTE: The above

4) The meet operator is the Union operator because we need to propagate the useful variable that comes from any path

5) The ENTRY and EXIT are both initialized to the empty set which is the Bottom element for our meet operator

6) The IN and OUT are also initialized to the empty set

7) There will be no impact of direction on correctness because the definition of useful variables will still be the same. The effect of forward analysis would be on time taken to converge. The impact of used variables (ie. generating more used variables) towards the end of the code section will take many iterations to propagate upwards. Also, the switch of direction will result in a change in the direction of the transfer function and the meet operator.

8) The algorithm converges because the analysis is monotonic. The Used Variable set grows towards the bottom element of the lattice (Case when all variables are used)

9)
```
FOR each variable definition:
        IF defined variable NOT in UV:
                (expression) = FAINT
```