

# hw11实验过程记录

## 参考资料

1. 实验文档
2. 虎书第19章
3. bitmap的实现参考 <https://www.cse.psu.edu/~deh25/cmpsc473/assignments/HW1/bitmap.c>
4. [支配树 - OI Wiki \(oi-wiki.org\)](https://oi-wiki.org/)

## 实验过程

### 数据结构

#### bitmap

本次实验的算法中涉及很多集合操作 (大多是关于CFG节点的), 故用bitmap作为集合表示, 速度较快。

实现放在 `utils/dsa/bitmap.h/c` 文件中。

#### 节点(基本块)信息

本次实验的算法中用到很多以基本块为索引的信息, 所以用 `SSA_block_info` 结构体存储关于基本块的信息。

```
1  typedef struct SSA_block_info_ *SSA_block_info;
2  struct SSA_block_info_ {
3      G_node mynode;
4      bitmap doms;
5      int idom;
6      blockIdList dom_tree_children;
7      blockIdList dom_frontiers;
8      Temp_tempList orig_vars;
9      Temp_tempList phi_vars;
10     instrInfoList instrInfos;
11     Temp_tempList blockIn;
12     Temp_tempList blockOut;
13 };
```

其中:

- mynode: bg中对应的节点
- doms: 存储必经节点
- idom: 直接必经节点
- dom\_tree\_children: 存储必经节点树的子节点
- dom\_frontiers: 存储必经节点边界

- `orig_vars`: 存储未进行SSA之前的块中所有被def的变量集合
- `phi_vars`: 存储被插入该块的所有  $\phi$  函数对应的原始变量集合
- `instrInfos`: 存储该块最终的指令序列，对每条指令还记录了对应的原始的 `orig_def` (在变量重命名时用到)
- `blockIn/blockOut`: 存储该块对应的Live-In/Live-Out变量集合

由于一拿到CFG就知道了基本块 (节点) 数量，故可以直接用数组存储所有节点的信息。

```
1 static SSA_block_info *blockInfoEnv;
2
3 static void init_blockInfoEnv(G_nodeList bg) {
4     blockInfoEnv =
5         (SSA_block_info *)checked_malloc(num_bg_nodes * sizeof *blockInfoEnv);
6     for (G_nodeList p = bg; p; p = p->tail) {
7         blockInfoEnv[p->head->mykey] = SSA_block_info_init(p->head, num_bg_nodes);
8     }
9 }
```

## 变量信息

在插入  $\phi$  函数时，需要为每个变量初始化 *defsites*(*a*)；变量重命名时，需要为每个变量维护栈存储最新版本。故用 `TAB_table` 存储变量到这些信息的映射关系：

```
1 typedef struct SSA_var_info_ *SSA_var_info;
2 struct SSA_var_info_ {
3     bitmap defsites;
4     var_stack stack;
5 };
6
7 static TAB_table varInfoEnv;
```

## 初始化

在初始化阶段，可以根据传入的bg和lg，记录每个基本块对应的blockIn/blockOut和最初的指令序列，并在表中记录所有的变量。

## 计算必经节点边界

### 计算CFG的逆后序

利用DFS得到逆后序的算法实现如下：

```

1 static void dfs_bg(int i) {
2     // perform a depth first search on bg
3     if (marked[i]) {
4         return;
5     }
6
7     marked[i] = TRUE;
8     for (G_nodeList l = G_succ(blockInfoEnv[i]->mynode); l; l = l->tail) {
9         dfs_bg(l->head->mykey);
10    }
11
12    bg_RPO[dfs_N--] = i;
13 }

```

## 计算必经节点

给定CFG和起点  $s_0$ , 根据方程  $D[u] = \{u\} \cup (\bigcap_{v \in \text{pred}[u]} D[v])$ , where  $D[s_0] = \{s_0\}$  可以通过数据流迭代法计算出任意节点  $u$  的必经节点集合  $D[u]$  (即不断进行迭代直到达到不动点)。

另外, 在迭代开始前先利用DFS排序, 得到CFG的逆后序 (RPO), 按该顺序进行迭代。这是因为希望每轮迭代时,  $u$  的所有前驱节点都尽可能执行完这次迭代, 而RPO恰有这样的性质: 若存在有向边  $u \rightarrow v$ , 则  $u$  出现在  $v$  之前, 尽管这只在CFG无环时严格成立, 但仍然能减少迭代次数。

在初始化阶段, 应该令  $D[s_0] = \{s_0\}$ , 而其他所有节点的支配集都初始化为所有节点。

算法实现如下:

```

1 static void compute_bg_doms() {
2     bitmap_set(blockInfoEnv[0]->doms, 0);
3     for (int i = 1; i < num_bg_nodes; ++i) {
4         bitmap_set_all(blockInfoEnv[i]->doms);
5     }
6
7     bool changed = TRUE;
8     int num_iters = 0;
9     bitmap tmp = Bitmap(num_bg_nodes);
10    while (changed) {
11        changed = FALSE;
12        num_iters++;
13        for (int i = 0; i < num_bg_nodes; ++i) {
14            int u = bg_RPO[i];
15            if (u == 0) {
16                continue;
17            }
18
19            bitmap_set_all(tmp);
20
21            // D[u] = {u} U (intersection of D[p] for all p in preds[u])
22            for (G_nodeList p = G_pred(blockInfoEnv[u]->mynode); p; p = p->tail) {

```

```

23     int v = p->head->mykey;
24     bitmap_intersection_into(tmp, blockInfoEnv[v]->doms);
25 }
26 bitmap_set(tmp, u);
27
28 if (!bitmap_equal(tmp, blockInfoEnv[u]->doms)) {
29     changed = TRUE;
30     bitmap_copy(blockInfoEnv[u]->doms, tmp);
31 }
32 }
33 }
34
35 compute_doms_iter = num_iters;
36 }

```

## 求解必经节点树

求解必经节点树，实际上就是求解所有节点  $u$  的直接必经节点  $idom(u)$  (除了起点  $s_0$  没有直接必经节点)。

直接必经节点的定义如下：设  $u (u \neq s_0)$  的支配集为  $D[u]$ ，若存在  $v \in D[u] (v \neq u)$ ，使得  $\forall w \in D[u] \setminus \{u, v\}$ ， $v$  不是  $w$  的必经节点，则  $idom(u) = v$ 。

首先给出引理：若  $u \neq v \neq w$ ， $v \text{ dom } u$  且  $w \text{ dom } u$ ，则必有  $v \text{ dom } w$  或  $w \text{ dom } v$ 。

引理证明：反设  $v$  不支配  $w$  且  $w$  不支配  $v$ ，则一定存在一条不经过  $v$  的从  $s_0$  到  $w$  的路径  $s_0 \rightarrow \dots \rightarrow w$ ，再由  $w \text{ dom } u$  知一定存在一条不经过  $v$  的从  $s_0$  到  $u$  的路径  $s_0 \rightarrow \dots \rightarrow w \rightarrow \dots \rightarrow u$ ，这与  $v \text{ dom } u$  矛盾！

由引理可以得到  $idom(u) = v$  等价于存在  $v \in D[u] (v \neq u)$ ，使得  $\forall w \in D[u] \setminus \{u, v\}$ ， $w \text{ dom } v$ 。

进一步， $idom(u) = v$  等价于存在  $v \in D[u] (v \neq u)$ ，使得  $D[u] \setminus D[v] = \{u\}$ 。

证明：显然

$D[u] \setminus D[v] = \{u\} \cup (D[u] \setminus \{u, v\}) \setminus D[v] = \{u\} \cup \{w : w \in D[u] \setminus \{u, v\}, \text{ but } w \text{ not dom } v\}$ ，故等价性得证。

故可以很轻松地根据最后的式子得到每个点的直接必经节点，时间复杂度为  $O(\frac{n^2}{32})$ 。

算法实现如下：

```

1  static void compute_bg_idoms() {
2      blockInfoEnv[0]->idom = -1;
3
4      bitmap u_mask = Bitmap(num_bg_nodes);
5      for (int u = 1; u < num_bg_nodes; ++u) {
6          bitmap_clear_all(u_mask);
7          bitmap_set(u_mask, u);
8          for (int v = 0; v < num_bg_nodes; ++v) {
9              // idom[u] != u
10             // idom[u] dominates u
11             if (v == u || !bitmap_read(blockInfoEnv[u]->doms, v)) {

```

```

12         continue;
13     }
14     // idom[u] is the closest dominator of u
15     bitmap flag = bitmap_difference(blockInfoEnv[u]->doms, blockInfoEnv[v]->doms);
16     if (bitmap_equal(flag, u_mask)) {
17         blockInfoEnv[u]->idom = v;
18         break;
19     }
20 }
21 }
22 }

```

在得到直接必经节点后，可以直接构建必经节点树：

```

1 static void construct_bg_dom_tree() {
2     for (int i = 1; i < num_bg_nodes; ++i) {
3         int idom = blockInfoEnv[i]->idom;
4         if (idom == -1) {
5             continue;
6         }
7
8         blockInfoEnv[idom]->dom_tree_children = BlockIdlist_Splice(blockInfoEnv[idom]-
9 >dom_tree_children, BlockIdList(i, NULL));
10    }
11 }

```

## 求解必经节点边界

支配边界的定义如下：若  $x$  是  $w$  的某个前驱（记为  $p$ ）的支配点，但  $x$  不严格支配  $w$ ，称  $w$  是  $x$  的支配边界。

虎书给出了一个高效的计算公式： $DF[x] = (DF_{\text{local}}[x]) \cup \bigcup_{c \in \text{children}[x]} DF_{\text{up}}[c]$ （children是指必经节点树的子节点），其中

- $DF_{\text{local}}[x]$ ：  $x$  的后继中不以  $x$  为严格必经节点的那些点
- $DF_{\text{up}}[c]$ ：在  $DF[c]$  中不以  $\text{idom}(c) = x$  作为严格必经节点的那些点

对等价性的证明：

先证必要性：若  $w$  是  $x$  的支配边界，则由定义， $x$  是  $w$  的某个前驱（记为  $p$ ）的支配点，但  $x$  不严格支配  $w$ （即  $x = w$  或  $x$  不支配  $w$ ）。

对  $p$  进行分类：

1. 若  $p = x$ ，则显然  $w$  为  $x$  的后继，但不以  $x$  为严格支配点，故  $w \in DF_{\text{local}}[x]$ 。
2. 若  $p \neq x$ ，则在支配树中一定有一条从  $x$  到  $p$  的路径  $P$ ，记  $P$  中紧接着  $x$  的节点为  $c$ （即  $\text{idom}(c) = x$ ，其中  $c$  可以等于  $p$ ），自然有  $c$  支配  $p$ ；而由  $x$  不严格支配  $w$  知  $c$  也不严格支配  $w$ （反设  $c$  严格支配  $w$ ，则由  $x$  支配  $c$  知  $x$  支配  $w$ ，再由  $c \neq w$  知  $x \neq w$ ，否则若  $x = w$  则  $x$  和  $c$  互相支配，得出  $x = c$  的矛盾，这样最终有  $x$  严格支配  $w$ ，矛盾！），故  $w$  是  $c$  的支配边界。而由定义  $x$  不严格支配  $w$ ，故  $w \in DF_{\text{up}}[c], \exists c \in \text{children}[x]$ 。

再证必要性：记等式右边的集合为  $W$ ，任取  $w \in W$ ，下证  $w \in DF[x]$ 。

首先，若  $w \in DF_{\text{local}}[x]$ ，结论显然成立。

对  $w \in \bigcup_{c \in \text{children}[x]} DF_{\text{up}}[c]$ ，一定存在某个  $\hat{c} \in \text{children}[x]$ ，使得  $w \in DF_{\text{up}}[\hat{c}]$ ，即  $\hat{c}$  是  $w$  的某个前驱

(记为  $\hat{p}$ ) 的支配点，但  $\hat{c}$  不严格支配  $w$ ，且  $x$  不严格支配  $w$ 。

显然由  $x$  支配  $\hat{c}$ ， $\hat{c}$  支配  $\hat{p}$  知  $x$  支配  $\hat{p}$ ，且  $x$  不严格支配  $w$ ，故  $w \in DF[x]$ 。

显然在算法中，对每个节点  $x$ ，可以先计算  $DF_{\text{local}}[x]$ ，然后递归计算  $DF[c]$ ，从而获取  $DF_{\text{up}}[c]$ ，最终将两部分合并得到  $DF[x]$ 。

在实现过程中，可以先用bitmap表示这些集合进行运算，最后将结果从bitmap转成DF的形式。

算法实现如下：

```
1  static void compute_bg_df_recur(int u) {
2      bitmap tmp = Bitmap(num_bg_nodes);
3
4      // compute DF_local[u]
5      for (G_nodeList s = G_succ(blockInfoEnv[u]->mynode); s; s = s->tail) {
6          int v = s->head->mykey;
7          if (blockInfoEnv[v]->idom != u) {
8              bitmap_set(tmp, v);
9          }
10     }
11
12     // compute DF_up[u]
13     for (blockIdList p = blockInfoEnv[u]->dom_tree_children; p; p = p->tail) {
14         int w = p->blockid;
15         compute_bg_df_recur(w);
16         for (blockIdList q = blockInfoEnv[w]->dom_frontiers; q; q = q->tail) {
17             int x = q->blockid;
18             if (!bitmap_read(blockInfoEnv[x]->doms, u) || x == u) {
19                 bitmap_set(tmp, x);
20             }
21         }
22     }
23
24     // store DF[u]
25     for (int i = 0; i < num_bg_nodes; ++i) {
26         if (bitmap_read(tmp, i)) {
27             blockInfoEnv[u]->dom_frontiers = BlockIdlist_Splice(blockInfoEnv[u]-
28 >dom_frontiers, BlockIdList(i, NULL));
29         }
30     }
```

## 插入 $\phi$ 函数

算法完全参考虎书19.1.3。

算法的整体思想是对变量 $a$ ，所有包含对 $a$ 的定义的基本块，应该在它们的必经节点边界处插入  $\phi$  函数，并且对被插入  $\phi$  函数的必经节点边界依然要继续递归地考虑。

值得注意的是，可以对虎书上的算法进行一个小改进：如果该变量 $a$ 在必经节点边界处不是Live-In的，就不用插入对应的  $\phi$  函数了。

如果不做这个改进，会遇到一个大问题：算法会把每次条件跳转时，`cmp+br` 中用到的 `i1` 变量在该块的后继中插入  $\phi$  函数，但首先该变量在别的块中根本不会 def/use，其次还要单独对这个 `i1` 类型区分，从而导致很多问题，故需要做这个“改进”。

以下是算法的基本步骤 (针对每个变量 $a$ ):

1. 计算  $defsites(a)$ : 即所有包含对 $a$ 的定义的基本块，可以根据  $A_{orig}[n]$  计算得到，其中
  - $A_{orig}[n]$ : 表示在基本块  $n$  中有定义语句的所有变量集合，可以直接通过  $n$  中所有语句的 `FG_Def` 的并得到
2. 初始化:  $W = defsites(a)$ , 所有基本块的  $A_\phi[Y] = \emptyset$ , 其中
  - $W$ : 表示包含对 $a$ 的定义的所有基本块的集合，注意到在插入  $\phi$  函数后要更新  $W$
  - $A_\phi[Y]$ : 表示基本块  $Y$  中所有已经插入  $\phi$  函数的变量集合
3. 每次从  $W$  中取出一个基本块  $n$  处理，直到  $W$  为空集:
  1. 检查  $DF[n]$  中的每个基本块  $Y$ :
    - 如果还没插入  $\phi$  函数 (即  $a \notin A_\phi[Y]$ ), 且  $a \in LiveIn(Y)$ , 就在块  $Y$  的顶端插入  $\phi$  函数, 并更新  $A_\phi[Y] \leftarrow A_\phi[Y] \cup \{a\}$
    - 在插入  $\phi$  函数后, 如果  $Y$  还没有考虑过 (即  $Y$  中原来没有对 $a$ 的定义, 亦即  $a \notin A_{orig}[Y]$ ), 将  $Y$  加入  $W$  中继续考虑

算法实现如下:

```
1 static void compute_phi_functions(G_nodeList lg) {
2     // step 1: record defsites of each variable
3     G_nodeList p = lg;
4     for (int i = 0; i < num_bg_nodes; ++i) {
5         for (Temp_tempList t1 = blockInfoEnv[i]->orig_vars; t1; t1 = t1->tail) {
6             Temp_temp var = t1->head;
7             SSA_var_info info = (SSA_var_info)TAB_look(varInfoEnv, (void *)var);
8             if (!info) {
9                 fprintf(stderr, "Error: var not found in varInfoEnv\n");
10                exit(1);
11            }
12            bitmap_set(info->defsites, i);
13        }
14    }
15
16    // step2: place phi func
```

```

17  bitmap w = Bitmap(num_bg_nodes);
18  Temp_temp top = varInfoEnv->top;
19  binder b;
20  while (top) {
21      b = TAB_getBinder(varInfoEnv, (void *)top);
22      Temp_temp var = (Temp_temp)b->key;
23      SSA_var_info info = (SSA_var_info)b->value;
24
25      // w = defsites(var)
26      bitmap_copy(w, info->defsites);
27
28      while (!bitmap_empty(w)) {
29          // remove a node u from w
30          int u = bitmap_get_first(w);
31          bitmap_clear(w, u);
32
33          // for each node v in DF[u]
34          for (blockIdList p = blockInfoEnv[u]->dom_frontiers; p; p = p->tail) {
35              int v = p->blockid;
36              // if var not in phi_vars(v), then place phi function for var
37              if (!Temp_TempInTempList(var, blockInfoEnv[v]->phi_vars) &&
38                  Temp_TempInTempList(var, blockInfoEnv[v]->blockIn)) {
39                  place_phi_func(var, v);
40                  blockInfoEnv[v]->phi_vars =
41                      Temp_TempList(var, blockInfoEnv[v]->phi_vars);
42                  // need to reconsider v
43                  if (!Temp_TempInTempList(var, blockInfoEnv[v]->orig_vars)) {
44                      bitmap_set(w, v);
45                  }
46              }
47          }
48      }
49
50      top = (Temp_temp)b->prevtop;
51  }
52 }

```

- 在实际插入  $\phi$  函数时 (即 `place_phi_func` 函数), 需要插入在block的第一条指令后面 (因为第一条指令就是开头的label)。

## 变量重命名

算法完全参考虎书19.1.4。

算法的主要思想为在遍历必经节点树的过程中, 为每个变量维护一个栈用于记录该变量的最新版本。

在初始化阶段, 对每个变量  $a$ ,  $stack[a] \leftarrow \{a_0\}$  (即压入最开始的版本)。

以下是算法的基本步骤 (从起始块开始):

1. 遍历基本块  $n$  中的每个语句  $S$ :



1. 若  $S$  不是  $\phi$  函数, 则  $S$  中每个被use的变量  $x$  都要用最新版本  $\text{top}(\text{Stack}[x])$  替换
2. 不管  $S$  是不是  $\phi$  函数, 对  $S$  中每个被def的变量  $a$ , 都要新申请一个temp作为新的版本  $a_i$  压入  $\text{Stack}[a]$  中, 并且在  $S$  中用  $a_i$  替换  $a$ 
  - 这里需要记录  $S$  的原始被def的变量  $a$ , 以便在递归返回后恢复栈中版本
2. 对基本块  $n$  的每个后继  $Y$  (首先通过 `get_pred_num` 函数获取  $n$  是  $Y$  前驱中的顺序, 记  $n$  是  $Y$  的第  $j$  个前驱):
  1. 对  $Y$  中的每一个  $\phi$  函数, 设第  $j$  个操作数为  $a$ , 则要用最新版本  $\text{top}(\text{Stack}[a])$  替换  $a$
3. 按照必经节点树的顺序, 对  $n$  的每个子节点  $X$  递归调用 **Rename(X)**
4. 还原栈: 对  $n$  中每个语句  $S$  中每个被def的变量  $a$ , 从  $\text{Stack}[a]$  中弹出栈顶版本

算法实现如下:

```

1  static void rename_vars() {
2      // step 1: initialize the stack
3      Temp_temp top = varInfoEnv->top;
4      binder b;
5      while (top) {
6          b = TAB_getBinder(varInfoEnv, (void *)top);
7          Temp_temp var = (Temp_temp)b->key;
8          SSA_var_info info = (SSA_var_info)b->value;
9          var_stack_push(info, var);
10         top = (Temp_temp)b->prevtop;
11     }
12
13     // step 2: rename variables
14     rename_vars_recur(0);
15 }
16
17 static void rename_vars_recur(int u) {
18     #ifdef RENAME_DEBUG
19         fprintf(out, "rename_vars_recur: %d\n", u);
20     #endif
21     // rename variables in block u
22     for (instrInfoList p = blockInfoEnv[u]->instrInfos; p; p = p->tail) {
23         AS_instr s = p->instr;
24         if (!is_phi_func(s)) {
25             Temp_tempList use_x = NULL;
26             switch (s->kind) {
27                 case I_OPER: {
28                     use_x = s->u.OPER.src;
29                     break;
30                 }
31                 case I_MOVE: {
32                     use_x = s->u.MOVE.src;
33                     break;
34                 }
35                 default:

```

```

36         break;
37     }
38     while (use_x) {
39         use_x->head = Var_Stack_top((SSA_var_info)TAB_look(varInfoEnv,
Temp_namedtemp(use_x->head->num, use_x->head->type)));
40         use_x = use_x->tail;
41     }
42 }
43
44 Temp_tempList def_x = NULL;
45 switch (S->kind) {
46     case I_OPER: {
47         def_x = S->u.OPER.dst;
48         break;
49     }
50     case I_MOVE: {
51         def_x = S->u.MOVE.dst;
52         break;
53     }
54     default:
55         break;
56 }
57
58 // record the origin def
59 p->origin_def = NULL;
60 for (Temp_tempList q = def_x; q; q = q->tail) {
61     p->origin_def =
62         Temp_TempListSplice(p->origin_def, Temp_TempList(q->head, NULL));
63 }
64
65 while (def_x) {
66     Temp_temp new_x = Temp_newtemp(def_x->head->type);
67     Var_Stack_push((SSA_var_info)TAB_look(varInfoEnv, def_x->head), new_x);
68     def_x->head = new_x;
69     def_x = def_x->tail;
70 }
71 }
72
73 for (G_nodeList Y = G_succ(blockInfoEnv[u]->mynode); Y; Y = Y->tail) {
74     int cnt = get_pred_num(G_pred(Y->head), u);
75     for (instrInfoList p = blockInfoEnv[Y->head->mykey]->instrInfos; p;
76         p = p->tail) {
77         if (!is_phi_func(p->instr)) {
78             continue;
79         }
80
81         int op_cnt = 0;
82         for (Temp_tempList q = p->instr->u.OPER.src; q; q = q->tail, ++op_cnt) {
83             if (op_cnt == cnt) {
84                 q->head = Var_Stack_top((SSA_var_info)TAB_look(varInfoEnv, q->head));

```

```
85     }
86   }
87 }
88 }
89
90 // rename variables in the children of u
91 for (blockIdList p = blockInfoEnv[u]->dom_tree_children; p; p = p->tail) {
92   rename_vars_recur(p->blockid);
93 }
94
95 // pop the stack
96 for (instrInfoList p = blockInfoEnv[u]->instrInfos; p; p = p->tail) {
97   Temp_tempList def_x = p->origin_def;
98   while (def_x) {
99     Var_stack_pop((SSA_var_info)TAB_look(varInfoEnv, def_x->head));
100    def_x = def_x->tail;
101  }
102 }
103 }
```

## 测试结果



















---

运行 `make test-run`，结果如下：

```
● zqwh@LAPTOP-HDCBVNK7:~/compiler/2024/hw11$ make test-run
[bubblesort]
0.000000 1.000000 2.000000 3.000000 5.000000 6.000000 9.000000
0
[example]
5
5: 2 3 2 5 2
[example01]
[example02]
0
[example03]
0
[example04]
[example05]
[example06]
[example07]
[example08]
0
[example09]
0
[example10]
[example11]
[example12]
0
[example13]
0
[example14]
[fibonacci]
Enter the number of term:14
0 1 1 2 3 5 8 13 21 34 55 89 144 233
0
[float]
0
[hw8test00]
3
3: 2 3 2
[intbubblesort]
0 1 2 3 5 6 9
0
[onefib]
```

# 开发过程

git提交记录如下：

	 <b>hw11</b> hw11: fix sort rpo bug	16 May 2024 08:36
	hw11: fix place phi func bug	15 May 2024 11:09
	hw11: add runone	14 May 2024 23:58
	hw11: merge master	14 May 2024 23:58
	hw11: refactor	13 May 2024 22:56
	hw11: complete renane and fix place phi func bug	13 May 2024 12:22
	hw11: merge master	12 May 2024 21:11
	hw11: add rename (not completed)	12 May 2024 18:32
	hw11: add phi func instr placement	12 May 2024 16:58
	hw11: add phi func computation	12 May 2024 14:09
	hw11: add Stringf in util.h/c	12 May 2024 14:08
	hw11: add some utils in bitmap	12 May 2024 14:08
	hw11: fix dom tree construction and add dom frontier computation	12 May 2024 00:11
	hw11: add compute dom tree	11 May 2024 21:37
	hw11: add compute doms	11 May 2024 21:02
	hw11: add bitmap and checked_calloc	11 May 2024 21:02
	hw11: add sort bg in rpo	11 May 2024 18:06