3.2.2 Liveness with fixpoint!

Let us recall the notations here: A variable at the left-hand side of an assignement is *killed* by the block. A variable whose value is used in this bloc (before any assignment) is *generated*.

$$LV_{exit}(\ell) = \begin{cases} \emptyset & \text{if } \ell = \text{final} \\ \bigcup\{LV_{entry}(\ell') | (\ell, \ell') \in flow(G)\} \end{cases}$$

$$LV_{entry}(\ell) = \left(LV_{exit}(\ell) \backslash kill_{LV}(\ell)\right) \cup gen_{LV}(\ell)$$

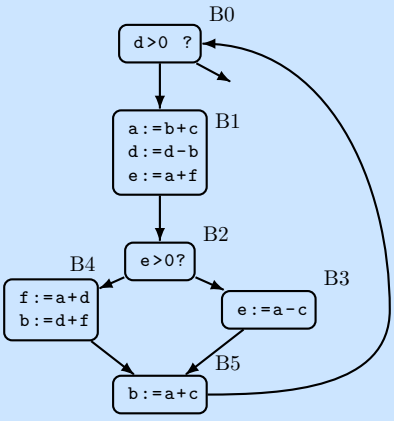
The sets are initialised to $\emptyset$ and computed iteratively, until reaching a fixpoint.

EXERCISE #5 ▸ **Live variables**
Generate the CFG for the following program:

```
while d>0 then {
    a:=b+c;
    d:=d-b;
    e:=a+f;
    if e>0 then {
        f:=a+d;
        b:=d+f;
    }
    else{
        e:=a-c;
        }
    b:=a+c;
}
```

On this CFG:
- Compute Gen, $Kill$ for each block ℓ
- Compute $In(\ell) = LV_{entry}(\ell)$ and $Out(\ell) = LV_{exit}(\ell)$ iteratively.
- Suppress the dead code.

Solution. On obtient pour graphe de flot de contrôle (faire attention aux noms des blocs, histoire de bien gérer les calculs):



Attention, la convention choisie ici est différente de celle du cours : on fait un graphe de flot de contrôle avec des blocs de bases de plusieurs instructions, alors que dans le cours on est à la granularité d'une instruction.

Cela a un impact sur le calcul de gen et kill. En particulier, on peut aussi considérer qu'une variable n'est dans $kill$ que si elle n'est pas générée avant, mais ça ne change pas l'analyse vu qu'on va appliquer un $\cup gen_{LV}$ après le $\backslash kill_{LV}$.

On a mis le test "e > 0" à l'extérieur du bloc B_1, mais on peut le mettre dans le bloc B1. Les calculs

seraient alors un peu modifiés. Dans la suite des calculs, on met en couleur (jaune) des ensembles qui sont modifiés d'une itération à une autre.

Initialisation:

- Variables générées, gen(ℓ) : leur présence comme opérandes sources dans le bloc génère une information : celle qu'elles doivent être vivantes en entrée du bloc. gen(ℓ) est l'ensemble des variables qui apparaîssent comme opérandes sources dans ℓ, mais qui ne sont pas affectées avant dans ℓ.
- Variables tuées, kill(ℓ) : celles dont on sait qu'elle n'ont pas besoin d'être présentes à l'entrée du bloc, car on va "les fabriquer" dans le bloc. Ce sont celles qui sont affectées dans le bloc.

On obtient alors les deux premières colonnes du tableau ci dessous. Attention, un bloc "génère" une variable seulement si il n'y a pas de kill **avant** dans le même bloc (c'est le cas pour a dans B_1 et f dans B_4.)

Les ensembles gen(ℓ) et kill(ℓ) sont des informations locales, mais on veut obtenir des informations globales de vivacité (qui assurent que la sémantique du code, telle que voulue par le programmeur, est respectée). On veut calculer, pour chaque bloc ℓ :

- $\mathrm{LV}_{\mathrm{entry}}(\ell)$: l'ensemble des variables qui doivent être vivantes à l'entrée du bloc ℓ; c'est l'ensemble des variables qui doivent être rendues vivantes par les blocs situés en amont du bloc ℓ dans le CFG.
- $\mathrm{LV}_{\mathrm{out}}(\ell)$: l'ensemble des variables qui doivent être vivantes en sortie du bloc ℓ ; c'est une information qui remonte des blocs situés en aval du bloc ℓ dans le CFG.

En prenant quelques exemples de CFG qui contiennent des structures conditionnelles ou des boucles, on se convainc assez facilement que l'on ne peut pas calculer $\mathrm{LV}_{\mathrm{entry}}(\ell)$ et $\mathrm{LV}_{\mathrm{out}}(\ell)$ localement, en une seule étape, d'après gen(ℓ) et kill(ℓ). On va donc calculer $\mathrm{LV}_{\mathrm{entry}}(\ell)$ et $\mathrm{LV}_{\mathrm{out}}(\ell)$ en mettant itérativement à jour deux ensembles :

- In(ℓ), que l'on va faire converger vers $\mathrm{LV}_{\mathrm{entry}}(\ell)$,
- Out(ℓ), que l'on va faire converger vers $\mathrm{LV}_{\mathrm{out}}(\ell)$.

Pour cela, on va effectuer les mises à jour suivantes. Pour chaque bloc ℓ, In(ℓ) reçoit l'ensemble gen(ℓ) des variables qui doivent être vivantes localement, unies avec celles Out(ℓ) qui doivent être vivantes en sortie moins celles qui sont fabriquées localement kill(ℓ) :

$$\mathrm{In}(\ell) \leftarrow \mathrm{gen}(\ell) \cup \big(\mathrm{Out}(\ell) \setminus \mathrm{kill}(\ell)\big).$$

Out(ℓ) reçoit l'ensemble des variables qui doivent être présentes en sortie du bloc. Cette information doit provenir de l'ensemble des blocs ℓ' situés en aval de ℓ dans le CFG :

$$\mathrm{Out}(\ell) \leftarrow \bigcup_{\ell' \text{ en aval de } \ell} \mathrm{In}(\ell').$$

Notez que Out(ℓ) peut être vide si ℓ est un bloc final dans le CFG. Pour utiliser ces formules, on calcule à chaque itération :

- d'abord les In(ℓ) pour tout ℓ,
- ensuite les Out(ℓ) pour tout ℓ, en utilisant les In(ℓ) calculés juste avant.

A priori, on ne sait pas ce que doivent êtres les In(ℓ) initiaux : on peut partir avec In(ℓ) = gen(ℓ) pour tout ℓ ; ils seront mis à jour par la suite. On peut aussi partir de $\emptyset$, ce qui sera le cas dans la suite.

Comme les ensembles de variables ne peuvent que grossir au fur et à mesure, et que au pire ils vaudront l'ensemble de toutes les variables du programme, qui est un ensemble fini, le processus de mise à jour des In(ℓ) et Out(ℓ) finit par converger : à une certaine itération, on retrouve les mêmes ensembles qu'à l'itération précédente. Alors, on a obtenu $\mathrm{In}(\ell) = \mathrm{LV}_{\mathrm{entry}}(\ell)$ et $\mathrm{Out}(\ell) = \mathrm{LV}_{\mathrm{out}}(\ell)$, pour tout ℓ.

On démarre avec $\emptyset$ partout, donc au premier coup on calcule $In = \emptyset \backslash kill \cup gen = gen$, puis $Out = \cup_{successeurs} In$, ...

B_i	Gen	$Kill$	$In1$	$Out1$	$In2$	$Out2$	$In3$	$Out3$	$In4$	$Out4$	$In5$	$Out5$	$In6$	$Out6$
B_0	d	$\emptyset$	d	$bcdf$	$bcdf$	$bcdf$	$bcdf$	$bcdf$	$bcdf$	$bcdf$	$bcdf$	$bcdf$	$bcdf$	$bcdf$
B_1	$bcdf$	ade	$bcdf$	e	$bcdf$	$acde$	$bcdf$	$acde$	$bcdf$	$acde$	$bcdf$	$acde$	$bcdf$	$acdef$
B_2	e	$\emptyset$	e	acd	$acde$	acd	$acde$	acd	$acde$	$acdf$	$acdef$	$acdf$	$acdef$	$acdf$
B_3	ac	e	ac	ac	ac	acd	acd	$acdf$	$acdf$	$acdf$	$acdf$	$acdf$	$acdf$	$acdf$
B_4	ad	bf	ad	ac	acd	acd	acd	$acdf$	acd	$acdf$	acd	$acdf$	acd	$acdf$
B_5	ac	b	ac	d	acd	$bcdf$	$acdf$	$bcdf$	$acdf$	$bcdf$	$acdf$	$bcdf$	$acdf$	$bcdf$

Quelques remarques:

- Il faut du temps pour d'une information soit propagée du bas vers le haut, les ensembles croissent doucement.
- On peut ensuite supprimer du code mort:
  - b est affectée en B_4 mais n'est pas vivante en sortie de B_4 ($Out(B_4) = \{a, c, d, f\}$), on peut donc enlever cette affectation.
  - De même, dans le bloc 3, e n'est pas vivante en sortie, donc on peut supprimer l'affectation à e dans ce bloc. En fait, e n'est vivante qu'entre B_1 et B_2.

  On peut noter que f est vivante partout sauf en entrée de B_4 (logique vu qu'elle est ré-affecté en début de bloc). On voit bien l'info de liveness de f remonter depuis B_0 au fil des itérations.

□