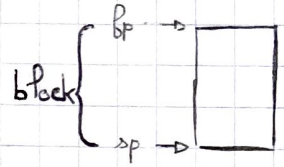
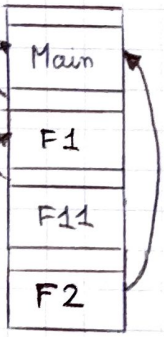
Fiche CMP2

HIR/LIR = High/Low Intermediate Representation
- ↳ Insertion/Suppression de nouveaux langages
- ↳ Correction debugs
- ↳ Evite de faire un grand pas



bp = frame pointer
sp = stack pointer

Variable temporaire = stocke les étapes d'un calcul
Block d'activation = stocke les variables, registre @ de retour → Une instance d'exec
Non-local variable = utilise des static links du block à son parent



Calling conventions (HIR) = preserve fp, sp, allocate the frame, handle static link, receive args

Traduction des instructions = Linéariser programme → Checker si 2 arbres sont commutables pour limiter sauvegarde des temporaires. Sous arbres "pur" si variables const ou read-only

Basic block: Commence par un label / Finit par un jump

Avoir moins de jump possible: Réorganise les blocks

Algo:
- Parcours les blocks en basic blocks
- "Ajoute" le block de départ/prochain block
- Si fini par un jump
  - Si block dst dispo, on ajoute
  - Sinon on en prend un autre
- Si fini par un cjump
  - Si block false dispo, on ajoute
  - Si block false non-dispo mais true oui, on flip la condition et on ajoute (true)
  - Sinon on crée un label temp * où le cjump va aller dessus

* label temp
jump false

CISC : très haut niveau mais instructions ont un coût variable

RISC : simple, peut d'instruction mais pipeline peut causer du délai

IF = Instruction Fetch → **ID** = Instruction Decode → **EX** = Execute → **MA** = Memory Access → **WB** = Write Back

Liveness: Transforme le texte en un graphe et on va colorier les arcs pour chaque variable vivante

$in[n] = use[n] \cup (out[n] \setminus def[n])$

$out[n] = \bigcup_{s \in succ[n]} in[s]$

Astuces: - ds in il y a tjrs le use → parfait pour démarrer
- démarre par la fin est plus rapide mais ne pas suivre la technique du cours

	use	def	in	out	in	Label
a := 0		a	c	a,c	c	a,c
b := a+1	a	b	a,c	b,c	a,c	b,c
c := c+b	b,c	c	b,c	b,c	b,c	b,c
a := b*2	b	a	b,c	a,c	b,c	a,c
a < N	a		a,c	c	a,c	a,c
return c	c		c		c	

2 étapes ⇒ Colorie les arcs

Data flow analysis :

gen = statement qui détermine (par les jumps, goto...)
kills = statement qui invalide le gen

$begin[n] = \bigcup end[p]$ $p \in pred[n]$
$end[n] = gen[n] \cup (begin[n] \setminus kills[n])$

	gen	kills	begin	end	begin	end	begin	end
a := 5	1	6		1		1		1
c := 1	2	4,7	1	1,2	1	1,2	1	1,2
L1 : if b < a goto L2			1,2	1,2	1,2,4	1,2,4	1,2,4	1,2,4
c := c+c	4	2,7	1,2	4,1	1,2,4	1,4	1,2,4	1,4
goto L1			4,1	4,1	1,4	1,4	1,4	1,4
L2 : a := c - a	6	1	4,1	6,4	1,4	4,6	1,2,4	2,4,6
c := 0	7	2,4	6,4	6,7	4,6	6,7	2,4,6	6,7

Coloriage du graphe : - Tu relies un noeud à (registre-1) arêtes
- Tu peux merge 2 noeuds coalescents qui relie à (registre-1) arêtes safety

- Quand on ne peut pas décider on fait le tableau de spill parity

use+def hors boucle	use+def in boucle	degree	spill parity

spill parity = (hors boucle + 10 × in boucle) / degree

On sacrifie la variable avec le spill parity le plus petit sur la stack

- On remplace variable par son registre
- On relie les lignes inutiles

Graphe de dépendance : - Une instruction = 1 noeud
- 1 dépendance = 1 arc
  - 0 = WAW, WAR
  - 1 = RAW
  - 2 = RAR

} Construction

W = write
R = read

Donne l'ordre d'exec
- On "lie" en premier les noeuds sans prédécesseurs avec le + gros poids
- Si 2 noeuds même poids, on prend celui plus éloignés de ses prédécesseurs