Corrected

Compilation de Programmes

CCF Session 1 - **NE PAS ENLEVER LES AGRAPHES**

Durée totale : 1 heure 30

Toute communication (orale, téléphonique, par messagerie, etc.) avec les autres étudiants est interdite. Aucun document autorisé.

— Pour la partie QCM, plusieurs réponses peuvent être valides à chaque question, on souhaite avoir **toutes** les réponses valides. Chaque question admet au moins une réponse valide et au moins une réponse incorrecte. Il n'y aura pas de point négatif.

— Pour les parties rédigées, vous répondrez obligatoirement dans les parties prévues pour. Il ne sera pas donné de nouvelle copie.

Consignes :

- Utiliser un **stylo à bille noir ou bleu**.
- **Noircir ou bleuir** la/les cases, sans dépasser !
- Pour corriger (dernier recours) : effacez proprement la case.
- Dans les parties rédigées, les carrés gris (prof) sont pour la correction, merci de ne rien écrire dedans.

Numéro étudiant à coder (sans le p, 8 chiffres, il est sur votre carte étudiant !)

- Notez-le ici : .....
- Encodez-le ci-contre (chiffre des unités tout à droite, en remplaçant p par 1) : par exemple, pour un numéro $p1234567$, ie 11234567 vous grisez le 1 de la première colonne, le 1 de la deuxième, le 2 de la troisième...).

☐0	☐0	☐0	☐0	☐0	☐0	☐0	☐0
☐1	☐1	☐1	☐1	☐1	☐1	☐1	☐1
☐2	☐2	☐2	☐2	☐2	☐2	☐2	☐2
☐3	☐3	☐3	☐3	☐3	☐3	☐3	☐3
☐4	☐4	☐4	☐4	☐4	☐4	☐4	☐4
☐5	☐5	☐5	☐5	☐5	☐5	☐5	☐5
☐6	☐6	☐6	☐6	☐6	☐6	☐6	☐6
☐7	☐7	☐7	☐7	☐7	☐7	☐7	☐7
☐8	☐8	☐8	☐8	☐8	☐8	☐8	☐8
☐9	☐9	☐9	☐9	☐9	☐9	☐9	☐9

1 Questions de cours/TP

Question 1 ♣ Les étapes d'analyse et transformation réalisées par un compilateur sont :

- ■ L'analyse syntaxique.
- ☐ L'analyse p-adique.
- ☐ La coloration syntaxique.
- ☐ La transformation de Fourier.
- ☐ L'évaluation des opérations arithmétiques par un interprète.
- ■ La vérification de type.
- ☐ Le chargement du fichier binaire exécutable en mémoire.
- ■ L'allocation de registres.
- ■ La génération de code 3 adresses.
- ■ L'analyse lexicale.

Question 2 ♣

Pendant les TPs, du code généré par ANTLR est utilisé pour :

- ■ La génération de code 3 adresses.
- ■ L'analyse syntaxique de code MiniC.
- ■ Le typage.
- ☐ L'analyse syntaxique de code RiscV.

Question 3 ♣ Cochez les affirmations correctes à propos des visiteurs générés par ANTLR4 :

- ☐ Les visiteurs permettent de faire simplement un parcours en largeur d'abord de l'arbre de dérivation.
- ■ Les visiteurs permettent de faire simplement un parcours en profondeur d'abord de l'arbre de dérivation.
- ■ Il est pertinent d'utiliser un visiteur pour réaliser un interprète pour le langage source.
- ☐ Il est pertinent d'utiliser un visiteur pour réaliser le coloriage de graphe utilisé pour l'allocation de registres.
- ☐ Il est pertinent d'utiliser un visiteur pour réaliser l'analyse de vivacité.
- ■ Il est pertinent d'utiliser un visiteur pour réaliser un générateur de code à partir du langage source.
- ■ Il est pertinent d'utiliser un visiteur pour réaliser l'analyse de typage.

Question 4 ♣ Dans les TPs, le visiteur de génération de code permet de :

- ☐ S'assurer que le code MiniC en entrée est correctement typé.
- ☐ Calculer la valeur des expressions du programme d'entrée.
- ☐ Générer du code RiscV prêt à être assemblé et exécuté.
- ■ Générer du code pas tout à fait exécutable.

Question 5 ♣
Dans la phase d'allocation de registres, l'analyse de vivacité des variables ("*liveness*") :

- ☐ Calcule exactement les variables vivantes en chaque point du programme.
- ☐ Se fait sur l'arbre (de dérivation/syntaxique) du programme.
- ■ Sert à calculer des informations de conflit.
- ■ Calcule un sous-ensemble des variables vivantes en chaque point.
- ■ Calcule un sur-ensemble des variables vivantes en chaque point.

Question 6 ♣
Dans notre série de TP, les tests unitaires (infrastructure `pytest`) :

- ■ Permettent d'avoir confiance en notre compilateur.
- ■ Servent à tester le code final généré.
- ☐ Servent à tester l'absence d'erreur à l'exécution.

Question 7 ♣
Dans un compilateur C, le typage :

- ☐ Peut être statique ou dynamique.
- ☐ S'effectue après la phase de génération de code.
- ☐ Permet de s'assurer de l'absence d'erreur à l'exécution.
- ■ Permet d'implémenter le domaine d'application de certaines opérations de calcul, définies dans la spécification du langage.
- ■ Permet d'éliminer les programmes que l'on ne désire pas compiler.

Question 8 ♣
Dans un compilateur, sont NP-complets les problèmes suivants :

- ■ L'ordonnancement d'instructions à l'intérieur d'un bloc de base.
- ☐ La génération de code.
- ☐ L'analyse syntaxique.
- ■ Le coloriage du graphe de conflit avec un nombre minimal de couleurs.
- ☐ Le calcul des intervalles de durée de vie des variables temporaires.

Attention ; il y a une erreur dans le pdf QCM corrigé généré automatiquement : l'analyse des variables vivantes calcule un sur-ensemble. La correction automatique a en revanche, tenu compte de la bonne réponse.

attention dans vos copies la note calculée n'est pas la note finale. La bonne note est celle sur tomuss.

Ce sont des éléments de correction, tout n'est pas complètement rédigé ! LG Janvier 2020

Notes de barème

Pour le QCM 0, 5 ou 1 point par question. Exercice 2 deux points, Exercice 3 2 pts pour la règle, 1 point pour l'arbre, 2 points pour la règle de génération de code. Exercice 4 2 points pour la liveness, 1.5 pour le graphe d'interférence, 3 points pour le coloriage et l'allocation, 1 point pour le code final.

2. Génération de code

Voici le code généré par notre compilo : (il manque quelques commentaires).

```
python3 ../../../TP2019-20/TP04/MiniC-codegen/Main.py exam19.c --reg-alloc=none
```

```
##Automatically generated RISCV code, MIF08 & CAP 2019
##non executable 3-Address instructions version
# coupé pour la correction
        # (stat (while_stat while ( (expr (expr (atom x)) > (expr (atom 3))) ) (stat_block { (block (stat
    (assignment x = (expr (expr (atom x)) - (expr (atom y)))) ;)) }))))
lbl_l_while_begin_0:
        li temp_2, 3
        li temp_3, 0
        ble temp_0, temp_2, lbl_end_relational_1
        li temp_3, 1
lbl_end_relational_1:
        beq temp_3, zero, lbl_l_while_end_0
        # (stat (assignment x = (expr (expr (atom x)) - (expr (atom y)))) ;)
        sub temp_4, temp_0, temp_1
        mv temp_0, temp_4
        j lbl_l_while_begin_0
lbl_l_while_end_0:
        # Return at end of function: #CUTCUT
```

3. Expression ifthenelse

Règle de typage Sans difficulté

$$\frac{\Gamma \vdash b : \texttt{bool} \qquad \Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash b ? e_1 : e_2 : \texttt{int}}$$

Attention, on type une expression, donc le type obtenu ne peut pas être `void`.

Arbre de typage Le début

```
\Gamma(y)=int     \Gamma |- x==8:bool  \Gamma |-18: int \Gamma |- 42+x : int
----------------            -------------------------------
```

```
\Gamma  |- y:int              \Gamma |- (x==8)?18: 42+x : int
-------------------------------------------------
\Gamma |- y = (x==8)?18: 42+x : void
```

Il manquait une règle dans la feuille d'accompagnement, il fallait appliquer la règle "naturelle".

Règle de génération de code On fait attention à ne pas évaluer la deuxième expression trop tôt, en particulier calculer la valeur pour tous les cas de b est une mauvaise idée. Attention on génère un code pour les expressions, donc on calcule dans un nouveau temporaire que l'on retourne à la fin.

```
CodeGenExpr(b?e1:e2) ==
dr <- newtemp()  # pour stocker le résultat final
lmilieu,lfin <- newlabels()
tb <- CodeGenExpr(b)
code.addinstructionCONDJUMP(tb , "=" , 0, lmilieu) # b faux
t1 <- CodeGenExpr(e1)
code.addinstructionMV(dr, t1)  # ne pas oublier ceci !
code.addInstructionJUMP(lfin)
code.addLabel(lmilieu)
t2 <- CodeGenExpr(e1)
code.addinstructionMV(dr, t2)  # ne pas oublier ceci !
code.addLabel(lfin)
return dr
```

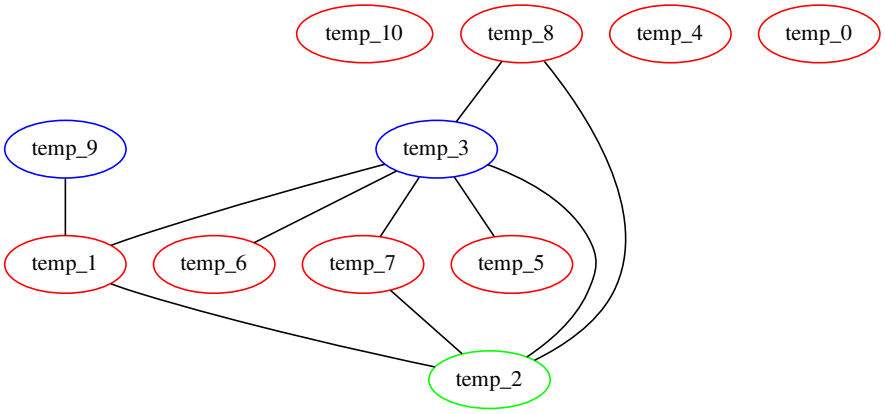
3. Liveness et génération de code

Liveness

code	temp0	temp1	temp2	temp3	temp4	temp5	temp6	temp7	temp8	temp9	temp10
`li temp4 12`											
`mv temp3 temp4`					*						
`li temp5 3`				*							
`add temp6 temp5 temp3`				*		*					
`mv temp2 temp6`				*			*				
`li temp7 4`			*	*							
`add temp8 temp7 temp2`			*	*				*			
`8:mv temp1 temp8`			*	*					*		
`9:sub temp9 temp3 temp2`		*	*	*							
`add temp10 temp9 temp1`		*								*	
`mv temp0 temp10`											*
`...`	*										

Graphe de conflits colorié La pile (fond de pile en premier) :

$$0, 4, 10, 5, 6, 9, 1, 7, 2, 3, 8$$



Allocation On attend une map dans le bon sens :

```
temp0 -> RegistrePhysique(t2), .... temp2 -> OffsetMemoire(-1)
```

Réécriture La ligne 8 devient (et un compilateur un peu malin éliminerait purement et simplement cette ligne ...) :

```
mv t2, t2
```

La ligne 9 devient, par exemple :

```
ld s2, -8(fp)  # chargement du deuxième opérande
sub t3, t3, s2
```