Final Exam
Compilation and Program Analysis (CAP)
February, 8th, 2017
Duration: 3 Hours

Only the CAP companion sheet is autorized.

Instructions :

1. We give some typing/operational/code generation rules as a companion.
2. Explain your results !
3. We give indicative timing.
4. Vous avez le droit de répondre en Français.

1 Operational semantics and code generation - 40 min

We recall the (abstract syntax) of the tiny imperative language seen in the course (no bloc, no procedure) :

$$
\begin{array}{rcll}
S(Smt) & ::= & x := e & \textit{assignment} \\
& | & \texttt{skip} & \textit{do nothing} \\
& | & S_1; S_2 & \textit{sequence} \\
& | & \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2 & \textit{test} \\
& | & \texttt{while } b \texttt{ do } S \texttt{ done} & \textit{loop}
\end{array}
$$

(e is a numerical expression, and b a boolean expression). In this exercice, we add the new statement $\texttt{exit}(b)$, where b is a boolean expression, whose intuitive semantics is the following :

— When b evaluates to false, $\texttt{exit}(b)$ is like `skip`.

— When b evaluates to true *and the flow is inside a loop*, then $\texttt{exit}(b)$ ends the loop and transfers the control at the end of the loop.

— When b evaluates to true and the flow is not inside a loop, then its behavior is the same as `skip`.

The natural semantics of the mini-language is in appendix.

Question #1

Compute the semantics of $S_0 = x := 2; \texttt{while } 0 < x \texttt{ do } x := x - 1 \texttt{ done}$ with the initial empty environment.

Question #2

Complete the LC3 code generated by the rules for S_0. (see the appendix).

Solution:

```
;;Automatically generated code
;;( stat  (assignment x = (expr (atom 10)) ;))
AND temp_1 , temp_1 , 0
ADD temp_1 , temp_1 , 10
;;( stat  (while_stat while (expr (atom ( (expr (expr (atom 0)) < (expr (atom i))) )) (
    stat_block { (block  (stat  (assignment x = (expr (expr (atom x)) - (expr (atom 1))) ;))
    ) }))
l_while_begin_1:
AND temp_2 , temp_2 , 0
ADD temp_2 , temp_2 , 0
NOT temp_4 , None
ADD temp_4 , temp_4 , 1
ADD temp_3 , temp_2 , temp_4
BRzp l_cond_neg_2
AND temp_5 , temp_5 , 0
ADD temp_5 , temp_5 , 1
BR l_cond_end_2
l_cond_neg_2:
AND temp_5 , temp_5 , 0
l_cond_end_2:
```

```
BRz l_while_end_1
;;( stat  (assignment x = (expr (expr (atom x)) − (expr (atom 1))) ;))
AND temp_6 , temp_6 , 0
ADD temp_6 , temp_6 , 1
NOT temp_8 , temp_6
ADD temp_8 , temp_8 , 1
ADD temp_7 , temp_1 , temp_8
ADD temp_1 , temp_7 , 0
BR l_while_begin_1
l_while_end_1:
```

Question #3

Extend the natural semantics of the language in order to take the new command exit() into account. Explain your rules. *You may have to modify the states. If necessary, to prove your rules, you can make the assumption that if a loop contains an exit, the exit is reached in its first execution.*

Question #4

Compute (properly) the semantics of the following programs with the initial environment σ where $\sigma(x) = 1$:

$$S_1 \ : \ \text{while true do exit(true) done} \ ; y := x$$
$$S_2 \ : \ \text{while } x > 0 \ \text{ do } x := x + 1; \text{exit}(x > 0) \text{ done} \ ; y := x$$
$$S_3 \ : \ \text{while } x > 0 \ \text{ do exit}(x > 0); x := x + 1 \text{ done} \ ; y := x$$

Question #5

Give a code generation rule for exit(), following the companion sheet style.

Solution: Dans le codage des états, il faut un moyen de se rappeler si on est à l'intérieur d'une boucle ou pas, et quel était le contrôle au début de la boucle. Plus précisément, on va coder dans l'état deux booléens, un premier qui dit que l'on est à l'intérieur d'(au moins une) boucle, et un deuxième qui dit que l'on est en train d'exécuter des instructions qui sont derrière un `exit(b)`, donc que l'on est en train d'avancer sans regarder l'effet de ces instructions sur l'état (la valeur des variables).

Cela donne UNE solution :

$$(\ell_1, \ell_2, \texttt{skip}, \sigma) \to (\ell_1, \ell_2, \sigma)$$

Pour l'affectation, elle est sautée si on est dans une boucle et derrière un `exit` ($\ell_2 = tt$ et $\ell_1 = tt$) et prise en compte sinon : (on peut fusionner les deux dernières règles en mettant l_2 explicitement)

$$(tt, tt, x := a, \sigma) \to (tt, tt, \sigma);$$

$$(ff, tt, x := a, \sigma) \to (ff, tt, \sigma[x \mapsto \mathcal{A}[a]\sigma]);$$

$$(tt, ff, x := a, \sigma) \to (tt, ff, \sigma[x \mapsto \mathcal{A}[a]\sigma]).$$

$$(ff, ff, x := a, \sigma) \to (ff, ff, \sigma[x \mapsto \mathcal{A}[a]\sigma]).$$

Règles pour le `exit(b)`, soit on n'est pas à l'intérieur d'une boucle et on se fiche de savoir si on a auparavant fait un exit ou pas, ou bien on est à l'intérieur d'une boucle et alors, soit on a déjà fait un exit, soit pas (avec la condition vraie ou la condition fausse), donc 4 règles :

$$(ff, \ell_2, \texttt{exit(b)}, \sigma) \to (ff, \ell_2, \sigma);$$

$$(tt, tt, \texttt{exit(b)}, \sigma) \to (tt, tt, \sigma);$$

$$(tt, ff, \texttt{exit(b)}, \sigma) \to (tt, tt, \sigma) \text{ si } \mathcal{B}[b]\sigma = tt;$$

$$(tt, ff, \texttt{exit(b)}, \sigma) \to (tt, ff, \sigma) \text{ si } \mathcal{B}[b]\sigma = ff$$

Passons maintenant au While, il s'agit de mettre le premier booléen à vrai si la condition est vraie, et le reste se passe au niveau des sous-instructions :

Si la condition du `while` est vérifiée :

$$\frac{(tt, \ell_2, S', \sigma) \to (\ell_1', \ell_2', \sigma') \quad (\ell_1, \ell_2', \texttt{ while } (b_1) \texttt{ do } S', \sigma') \to (\ell_1", \ell_2", \sigma")}{(\ell_1, \ell_2, \texttt{ while } (b_1) \texttt{ do } S', \sigma) \to (\ell_1, \ell_2, \sigma")} \text{ si } \mathcal{B}[b_1]\sigma = tt$$

- Supposons que cette boucle apparaisse dernière un exit, alors nécessairement $\ell_2 = tt$, information qui va être transmise à l'intérieur de S', donc $\ell_2' = tt$, et ce qui va permettre la non prise en compte des autres. Par ailleurs, on pourrait dire que $\ell_2" = tt$ aussi.
- Si la boucle n'apparaît pas derrière un exit, il y a deux cas :
  - pas de exit dans la boucle, on exécute une première fois l'instruction du corps de la boucle, puis le reste. Toutes les variables ℓ_2^i sont à ff.
  - il y a un exit dans la boucle (avec condition vraie). Sans perdre de généralité on peut supposer que cet exit a été effectué dans la première exécution de S'. Dans ce cas, $l_2' = tt$, ce qui va permettre la non-exécution du deuxième tour de boucle.

Dans ces deux cas, la variable ℓ_1 est mise à tt lors du passage à l'intérieur du corps de la boucle, et remise à sa valeur initiale lors de la sortie de la boucle. La règle résume tous ces cas.

Si la condition de la boucle `while` est fausse, alors il n'y a pas de problème :

$$(\ell_1, \ell_2, \texttt{ while } (b_1) \texttt{ do } S', \sigma) \to (\ell_1, \ell_2, \sigma) \text{ si } \mathcal{B}[b_1]\sigma = ff$$

Passons à la séquence, tout le travail étant fait lors de l'affectation et de la rencontre de la commande `exit` :

$$\frac{(\ell_1, \ell_2, S_1, \sigma) \to (\ell_1', \ell_2', \sigma') \quad (\ell_1', \ell_2', S_2, \sigma') \to (\ell_1", \ell_2", \sigma")}{(\ell_1, \ell_2, S_1; S_2, \sigma) \to (\ell_1", \ell_2", \sigma')}$$

2 Register Allocation - 30 min

We consider the following 3-address LC3 code (where we invent a new instruction `sub t1 t2 t3` that performs $t_1 \leftarrow t_2 - t_3$). t_is are temporaries to be allocated (registers, memory).

```
LEA R6 spinit
[ ... ]
ld  t1,label1
ld  t2,label2
sub t3,t1,t2
ld  t4,label3
ld  t5,label4
sub t6,t4,t5
add t7,t6,0
add t8,t3,t7
st  t8,label5
RET
```

```
;;data
label1  : .FILL #2
label2  : .FILL #3
label3  : .FILL #-1
label4  : .FILL 7
label5  : .BLZW

;;stack
spinit :  .FILL
stackend: .BLKW #15 ;

.END
```

Question #1

What is the expression computed by the code ? At which address will it be stored ?

Solution: Ce code calcule la valeur $(2-3)+(-1-7)=-9$ puis la range à l'adresse "label5".

Question #2

Fill the array in appendix with stars each time a given temporary is alive at the entry of the instruction. After the `st`, all temporaries are considered to be dead.

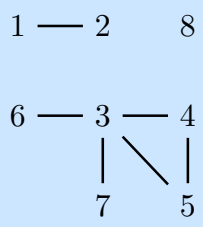
Solution:

code	t1	t2	t3	t4	t5	t6	t7	t8
`ld t1, label1`								
`ld t2, label2`	*							
`sub t3,t1,t2`	*	*						
`ld t4, label3`			*					
`ld t5, label4`			*	*				
`sub t6,t4,t5`			*	*	*			
`add t7,t6,0`			*			*		
`add t8,t3,t7`			*				*	
`st t8, label5`								*

Question #3

Draw the interference graph (8 nodes).

Solution: Le noeud i désigne le temporaire t_i :

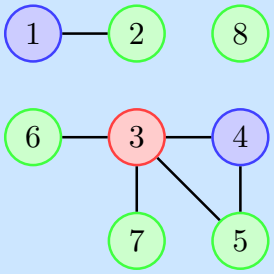


Question #4

Color this graph with the course's algorithm with 3 colors (green, blue, red, in this order). When

there is a choice to do, always choose the node with the lesser number. Draw the stack. (we recall that we push the nodes with lesser degree first).

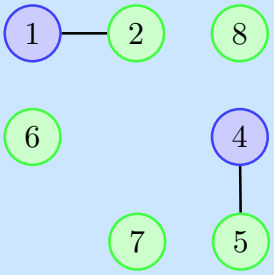
Solution: On empile donc les numéros dans cet ordre : 8,1,2,6,7,3,4,5 (la pile est à lire dans l'ordre inverse, 5 est en haut de pile). Cela nous donne le coloriage suivant :



Question #5

t_3 will be spilled. Re color the rest of the graph with 2 colors (green, blue).

Solution: La pile obtenue est 5,4,2,1,8,7,6 (5 en haut de pile) :



Question #6

Generate the final code with two registers (R1,R2) for the temporaries, $R6$ stack pointers, $R5$ to handle the spilled variable t_3.

Solution: Pour les variables allouées en registre, on remplace "vert" par R1, "bleu" par R2. On donne le décalage 0 pour t_3 car c'est la seule variable spillée.

```
LEA R6 spinit
[...]
ld  R2,label1
ld  R1,label2
sub R5,R2,R1  #calcul ds R5
STR R5 R6 1   #stockage en mémoire
ld  R2,label3
ld  R1,label4
sub R1,R2,R1
add R2,R1,0
LDR R5 R6 0    #récupération
add R1,R5,R1   #calcul
st R1,label5
RET
```

3 Dataflow - 20 min

source : F. Pereira. Consider the following program :

```
x:=1;
y:=read(); // random value for y
while y>0  {x:=x+1; y:=y-1;}
x:=2
print(x);
```

Question #1

Construct the CFG with one instruction per bloc.

Question #2

Perform the liveness analysis for this program (initialisation, computation, final result). Use the empty array in appendix.

Now consider the following program :

```
x:=1;
x:=x-1;
x:=2;
print(x);
```

where variable x is dead at the exit of the second instruction. On the other hand, x is alive at the exit of the second instruction, even though its value is only used to compute a dead variable. Under such circumstances, we say that x is a quasi-dead variable. In other words, a variable is quasi-dead either if it is dead, or if it is only used to compute a dead variable. In our example, x is quasi-dead in the OUT set of each one of the instructions. If a variable is not quasi-dead, then we shall call it *vigorous*.

Question #3

Design a new analysis to detect vigorous variables. *F*irst give your main ideas, then properly define the initialisation of your sets, and the equations you use to propagate this information. Perform it on the second example.

Solution: todo

4 Abstract Interpretation : modulo - 20 min

source P. Cousot

A multiplication $m \times n$ can be checked by summing the digits of integer m modulo 9, summing the digits of n modulo 9, and checking that their product modulo 9 is equal to the sum of the digits of the result r modulo 9. For example,

$$\begin{array}{rrcrcl} & 1234 & \rightarrow & 10 \bmod 9 & = & 1 \\ \times & 5678 & \rightarrow & 26 \bmod 9 & = & 8 \\ \hline = & 7006652 & \rightarrow & 26 \bmod 9 & = & 8 \end{array}$$

is a success, but

$$\begin{array}{rrcl} & 1234 & \rightarrow & 10 \bmod 9 = 1 \\ \times & 5678 & \rightarrow & 26 \bmod 9 = 8 \\ \hline = & 7006651 & \rightarrow & 27 \bmod 9 = 7 \neq 1 \times 8 \bmod 9 \end{array}$$

fails.

Question #1

Is a success a proof? Is a failure a proof?

Question #2

Show that this is an abstraction. Which property of modular arithmetic do you use?

Question #3

Generalise this idea and construct an abstract domain $\mathcal{A}$ of congruences modulo 9. *A partial informal solution will also give points.*

Question #4

Give an example of a program and an assertion that will be proved non reachable with your new abstract domain.

Solution:

- Casting out nines is a sound method of checking equations because of a property of mod arithmetic. Specifically, if n and n' (respectively, m and m') have the same remainder modu then so do $n \times m$ and $n' \times m'$. To compute n modulo 9, one observes that the sum of the digi the decimal writing of an integer has the same remainder, modulo 9, as this integer. Of co all 9 digits in n and m can be cast out.
- This is an example of abstraction by over-approximation because the number is replaced by set of all numbers which have the same remainder modulo 9. Of course a failure is a proof $n \times m \neq r$.
- However a success is not a proof of correctness of the multiplication since changing m, n a modulo 9 (e.g. by exchanging digits in the decimal representation or adding 9's) yields the s false positive result.

The same reasoning is valid for other operations $+$, $-$, $/$, etc. A sound generalization of the isthe congruence abstraction used in program analysis.

5 Functional languages - 70 min

Adapted from J.C Filliâtre

We extend the expressions of mini-ml with `iszero` (test to 0) and `print`. The expression `print e` prints and returns the value of e. A program is a set of function definitions (possibility mutually recursives, possibility defined more than once, ...) followed by a main expression.

Here is an example of a program :

```
def mult (x,y) = ifzero x then 0 else y + mult (x - 1, y)
print mult(2,4)
```

Question #1

Give a big steps operational semantics for *expressions*, under the form of a judgment $e \Rightarrow v, L$ where v is the final value of the expression e and L the list of all values printed by `print` in order (7 rules).

Solution: La correction aussi est une adaptation de celle de JC Filliâtre.

```
n --> n,[]

e1 --> n1,L1   e2 --> n2,L2
-------------------------
 e1+e2 --> n1+n2, L1L2

e1 --> v1,L1   e2[x<-v1] --> v2,L2
--------------------------------
    let x = e1 in e2 --> v2, L1L2

e1 --> 0,L1   e2 --> v,L2                e1 --> n, L1   n<>0   e3 --> v,L3
-----------------------------------      -----------------------------------
ifzero e1 then e2 else e3 --> v, L1L2    ifzero e1 then e2 else e3 --> v, L1L3

 e --> v,L
-----------------
print e --> v,Lv

 e1 --> v1,L1 ... en --> vn,Ln   f(x1,...,xn)=e   e[x1<-v1,...,xn<-vn] --> v,L
-------------------------------------------------------------------------------
  f(e1,...,en) --> v, L1L2...LnL
```

Question #2

Give an example of a program for which the e-main expression is such that $\nexists v, L, e \Rightarrow v, L$.

Solution:

```
def boucle(x)=boucle(x)
boucle(0)
```

Question #3

To efficiently compile programs we propose to eliminate common subexpressions e, *only when possible*, with the help of a `let x = e in ...`.

Question #4

Rewrite the following two programs (explain !)

```
def double(x) = x + x
def f(x) = print (double(x)+double(x))
print( f(1+2) + double(2) + double(2) + f(1+2) )
```

and

```
def boucle(x) = boucle(x)
ifzero 1 then boucle(0)
         else ifzero 0 then 1 else boucle(0)
```

Solution: Le premier programme (attention on ne partage pas `f(x)` car `f` fait un affichage).

```
def double(x) = x+x
def f(x) = print (let v = double(x) in v+v)
print (let x = 1+2 in
       let y = double(2) in
       f(x) + y + y + f(x))
```

Le deuxième ne doit pas être réécrit qu'autrement que lui même, sinon on modifie la sémantique (terminaison)

We give the following ANTLR grammar for programs and expressions :

```
grammar Fun;

prog: deflist expr #progdef;

deflist: deff+ ;
deff: 'def' VAR '(' VAR (',' VAR)* ')' '=' expr #deffun ;

expr:  INT     #cst
    |  VAR     #var
    | expr '+' expr #plus
    | 'let' VAR '=' expr 'in' expr #defi
    | 'ifzero' expr 'then' expr 'else' expr #testz
    | 'print' expr      #print
    | VAR '(' expr (','expr)* ')' #funcall
    ;

INT: [0-9]+;
VAR: [a-z]+;
WS   :   (' '|'\t'|'\n')+   -> skip;
```

Question #5

We say that a given expression is *pure* if its evaluation terminates and never calls `print`. Complete the ANTLR visitor in appendix (only the expression part) to be able to say if a given expression is pure.

Question #6

Complete the rest of the visitor that determines for all function declarations, if their body is pure, and stores the result in a global dictionary. Explain your algorithm.

Question #7

(Difficult, and long) We suppose that for each expression we can decide if it is pure or not : `if ispure(e)`. Write another visitor that computes for each expression its optimised version. *First explain your pseudo algorith, then implement in Python.*

Question #8

Give another example of code optimisation that can be done exploiting the purity information.

Fill and paste section

```
;;Automatically generated code
;;(stat (assignment x = (expr (atom 2)) ;)) ;;2 lines (x->temp_1)

;;(stat (while_stat while (expr (atom ( (expr (expr (atom 0)) < (expr (atom i))) ))) (
    stat_block { (block (stat (assignment x = (expr (expr (atom x)) - (expr (atom 1))) ;))) }))
    )
l_while_begin_1: ;; computation of 0-i (5 lines)

BRzp l_cond_neg_2
AND temp_5 , temp_5 , 0
ADD temp_5 , temp_5 , 1
BR l_cond_end_2
l_cond_neg_2:
AND temp_5 , temp_5 , 0
l_cond_end_2:
BRz l_while_end_1
;;(stat (assignment x = (expr (expr (atom x)) - (expr (atom 1))) ;)) ;;5/6lines

;;  1 branching instruction here

l_while_end_1:
;;  the end
```

code	t1	t2	t3	t4	t5	t6	t7	t8
`ld t1, label1`								
`ld t2, label2`								
`sub t3,t1,t2`								
`ld t4, label3`								
`ld t5, label4`								
`sub t6,t4,t5`								
`add t7, t6, 0`								
`add t8,t3,t7`								
`st t8, label5`								

			Step		Step		Step		Step	
ℓ	$kill(\ell)$	$gen(\ell)$	$In(\ell)$	$Out(\ell)$	$In(\ell)$	$Out(\ell)$	$In(\ell)$	$Out(\ell)$	$In(\ell)$	$Out(\ell)$

			Step		Step		Step		Step	
ℓ	$kill(\ell)$	$gen(\ell)$	$In(\ell)$	$Out(\ell)$	$In(\ell)$	$Out(\ell)$	$In(\ell)$	$Out(\ell)$	$In(\ell)$	$Out(\ell)$

			Step		Step		Step		Step	
ℓ	$kill(\ell)$	$gen(\ell)$	$In(\ell)$	$Out(\ell)$	$In(\ell)$	$Out(\ell)$	$In(\ell)$	$Out(\ell)$	$In(\ell)$	$Out(\ell)$

```
from FunVisitor import FunVisitor
from FunParser import *

class MyFunVisitor(FunVisitor):
  # Visit a parse tree produced by FunParser#progdef.
    def visitProgdef(self, ctx):




        # Visit a parse tree produced by FunParser#deflist.
    def visitDeflist(self, ctx):



    # Visit a parse tree produced by FunParser#deffun.
    def visitDeffun(self, ctx):



    # Visit a parse tree produced by FunParser#print.
    def visitPrint(self, ctx):




    # Visit a parse tree produced by FunParser#funcall.
    def visitFuncall(self, ctx):




    # Visit a parse tree produced by FunParser#cst.
    def visitCst(self, ctx):
```

```
    # Visit a parse tree produced by FunParser#var.
    def visitVar(self, ctx):




    # Visit a parse tree produced by FunParser#defi.
    def visitDefi(self, ctx):




    # Visit a parse tree produced by FunParser#testz.
    def visitTestz(self, ctx):




    # Visit a parse tree produced by FunParser#plus.
    def visitPlus(self, ctx):




#end
```

```
from FunVisitor import FunVisitor
from FunParser import *

class MyFunVisitor(FunVisitor):
  # Visit a parse tree produced by FunParser#progdef.
    def visitProgdef(self, ctx):




        # Visit a parse tree produced by FunParser#deflist.
    def visitDeflist(self, ctx):



    # Visit a parse tree produced by FunParser#deffun.
    def visitDeffun(self, ctx):



    # Visit a parse tree produced by FunParser#print.
    def visitPrint(self, ctx):




    # Visit a parse tree produced by FunParser#funcall.
    def visitFuncall(self, ctx):






    # Visit a parse tree produced by FunParser#cst.
    def visitCst(self, ctx):
```

```
    # Visit a parse tree produced by FunParser#var.
    def visitVar(self, ctx):




    # Visit a parse tree produced by FunParser#defi.
    def visitDefi(self, ctx):




    # Visit a parse tree produced by FunParser#testz.
    def visitTestz(self, ctx):




    # Visit a parse tree produced by FunParser#plus.
    def visitPlus(self, ctx):




#end
```