Final Exam
Compilation and Program Analysis (CAP)
January, 10 th, 2018
Duration: 3 Hours

No document

Instructions :

1. Every single answer must be informally explained AND formally proved.
2. We give indicative timing.
3. Vous avez le droit de répondre en Français.

Mu-While syntax and semantics

In this exam, we consider the Mu abstract syntax of the course :

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

where numerical expressions are :

$$
\begin{array}{rcll}
e & ::= & c & \textit{constant} \\
& | & x & \textit{variable} \\
& | & e + e & \textit{addition} \\
& | & e \times e & \textit{multiplication}
\end{array}
$$

and Boolean expressions :

$$
\begin{array}{rcll}
b & ::= & \textit{true} & \textit{constant} \\
& | & \textit{false} & \textit{constant} \\
& | & b \textit{ and } b & \textit{and} \\
& | & \textit{not } b & \textit{not} \\
& | & e_1 < e_2 & \textit{leq} \\
& | & e_1 = e_2 & \textit{eq}
\end{array}
$$

We also consider the structural operational semantics depicted in the following table (Source *Nielson*) :

$[\text{ass}_{\text{sos}}]$	$\langle x := a,\, s\rangle \Rightarrow s[x \mapsto \mathcal{A}[\![a]\!]s]$
$[\text{skip}_{\text{sos}}]$	$\langle \texttt{skip},\, s\rangle \Rightarrow s$
$[\text{comp}^1_{\text{sos}}]$	$\dfrac{\langle S_1,\, s\rangle \Rightarrow \langle S_1',\, s'\rangle}{\langle S_1;S_2,\, s\rangle \Rightarrow \langle S_1';S_2,\, s'\rangle}$
$[\text{comp}^2_{\text{sos}}]$	$\dfrac{\langle S_1,\, s\rangle \Rightarrow s'}{\langle S_1;S_2,\, s\rangle \Rightarrow \langle S_2,\, s'\rangle}$
$[\text{if}^{\text{tt}}_{\text{sos}}]$	$\langle \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2,\, s\rangle \Rightarrow \langle S_1,\, s\rangle$ if $\mathcal{B}[\![b]\!]s = \mathbf{tt}$
$[\text{if}^{\text{ff}}_{\text{sos}}]$	$\langle \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2,\, s\rangle \Rightarrow \langle S_2,\, s\rangle$ if $\mathcal{B}[\![b]\!]s = \mathbf{ff}$
$[\text{while}_{\text{sos}}]$	$\langle \texttt{while } b \texttt{ do } S,\, s\rangle \Rightarrow \langle \texttt{if } b \texttt{ then } (S; \texttt{while } b \texttt{ do } S) \texttt{ else skip},\, s\rangle$

Table 2.2: Structural operational semantics for **While**

1 Operational semantics - 40 min

The object of this exercise is to extend the Mu-While semantics with two new different constructions. *Adapted from R. Manevich.*

1.1 Extension with parallel assignments

Parallel assignments are of the form $x_1, x_2 \ldots x_n := a_1, a_2 \ldots a_n$. We consider the following *Fib* program :

```
1 if n=0
2   res := 0
3 else {
4   i,a,res := 1,0,1;
5   while (i<n) do
6      a, res, i := res, a+res, i+1
7   done
8 }
```

Question #1

Give a SOS semantics rule for this new construction.

Question #2

Compute the semantics of the *Fib* program under the initial state where all variables are 0 except n, which is 2. *For succintness, use the notation Fib*$[i]$ *to denote statement at line i and Fib*$[i, j]$ *to denote the subprogram between i and j (inclusive).*

Question #3

Give an alternative implementation *SeqFib* for *Fib* not using parallel assignement.

Question #4

Give two notions of SOS semantic equivalence $\equiv_1$ and $\equiv_2$ such that $Fib \equiv_1 SeqFib$ and $Fib \not\equiv_2 SeqFib$.

1.2 Extension with side-effect expressions

Consider the following modification for **arithmetical** expressions :

$$
\begin{array}{rcll}
e & ::= & c & \textit{constant} \\
 & | & x & \textit{variable} \\
 & | & e + e & \textit{addition} \\
 & | & e \times e & \textit{multiplication} \\
 & | & x{+}{+} & \textit{inc with side effect (1)} \\
 & | & {+}{+}x & \textit{inc with side effect (2)}
\end{array}
$$

Informal semantics :

— $x++$ increments x by one and evaluates to its new value.

— $++x$ increments x by one and evaluates to its old value.

We modify the semantics for expressions in order to compute new environments, now arithmetic expressions have evaluation rules of the form :

$$< a, \sigma > \rightarrow < a', \sigma' >$$

where σs are still States, ie functions from variables to values (same for boolean expressions).

Question #5

Write down rules for all numerical and boolean expressions. *Explain your choices.*

Question #6

Write down rules for statements, including those with parallel assignements. There is no need to repeat rules that remain the same.

Question #7

Show the modifications induced by your semantics (including the new derivation steps for expressions) on the following *SideFib* program :

```
1 if n=0
2   res := 0
3 else {
4   i,a,res := 1,0,1;
5   while (i++ < n) do
6      a, res := res, a+res
7   done
8 }
```

2 Register allocation for trees - 20 min

Adapted from J.C Filliâtre

We want to characterise "à la Sethi Ullman" the number of registers that are necessary to compute a given arithmetic expression following the syntax :

$$\begin{array}{rcll} e & ::= & x & \text{variable} \\ & | & -e & \text{negation} \\ & | & e+e & \text{addition} \end{array}$$

We compile these expressions into a machine with registers and the following instructions :

— `load` x, r reads the memory at address x and stores the result in register r.

— `neg` r performs the negation in place.

— `add` r_i r_j performs the operation $r_i + r_j$ and stores the result in r_j.

All variables have an address.

Here are the rules :

— Given e an expression and r a register, we want to produce a sequence of instructions that finally puts in r the value of the expression e.

— No simplification (common subexpressions, $x + x$, for instance, reloads x twice), nor use of addition associativity.

— We can exploit the property of addition commutativity (evaluate e_1 before e_2 or the converse while evaluating $e_1 + e_2$).

Question #1

For each of the following expressions, generate a program with a minimal number of registers (including the target register) :

1. $((x + y) + z) + t$;
2. $(x + y) + (z + t)$;
3. $x + ((y + z) + t)$;

Solution: voir filliatre 2, 3, 2 registres.

Question #2

Give a way to compute $n(e)$ the minimal number of registers necessary to compute e (structural induction on e).

Question #3

We define $size(e)$ the number of occurrences of the $+$ operator in the expression e. Give a minoration of $t(e)$ by an (exponential) expression in $n(e)$ (and a proof of it). Deduce the minimal size of an expression which is not computable with 5 registers.

Solution: cd filliatre. $t(e) \geq 2^{n(e)-1} - 1$. 31.

3 Code generation and Register Allocation - 40 min

We consider the following Mu program :

```
var x,y,z,t:int;
x=12; y=3+x; z=4+y; t=x-y+z;
```

For readibility reasons, $temp_i$ is renamed into ti. The 3 address code generation process of Lab 5 and 6 produces the following code, where $(t, z, y, x) \mapsto (t1, t2, t2, t3)$:

```
.let  t4  12
copy t3  t4
.let  t5  3
add t6 t5  t3
copy t2  t6
.let  t7  4
add t8 t7  t2
copy t1  t8
sub t9  t3  t2
add t10 t9  t1
copy t0  t10
```

Question #1

Generate (see Appendix) the final code for the first two lines of the 3 address code with the "all-in-memory" allocation strategy. *Use R_0 to compute stack addresses, R_6 as "stack pointer" and R_1 and R_2 to get access to stack elements*.

Solution:

```
;;Automatically generated LEIA code, MIF08 2017
;; all-in-memory allocation version
;; stack management
.set r6 stack
        ;; (stat (assignment x = (expr (atom 12)) ;))
        ;; .let temp_4 12
        .LET r1 12
        SUB r0 r6 3
        WMEM r1 [r0]
        ;; end .let temp_4 12
        ;; copy temp_3 temp_4
        SUB r0 r6 3
        RMEM r1 [r0]
        COPY r1 r1
        SUB r0 r6 2
        WMEM r1 [r0]
        ;; end copy temp_3 temp_4
```

Question #2

Propose a better strategy for allocating a copy in the case of the all-in-mem allocation. *Explain on the example and on the general case*.

Solution: Une copie peut tenir dans une même place. Dans le cas all inmem, rmem copy wmem on peut supprimer la copie.

Question #3

Same question for the "smart" allocation strategy.

Solution: L'idée serait de considérer ensemble les variables copiées dans le graphe de conflit, dans le même noeud. Il faut faire attention aux durées de vie néanmoins. " 'For each copy where the source and destination live ranges don't interfere, union the 2 live ranges and remove the copy"

Question #4

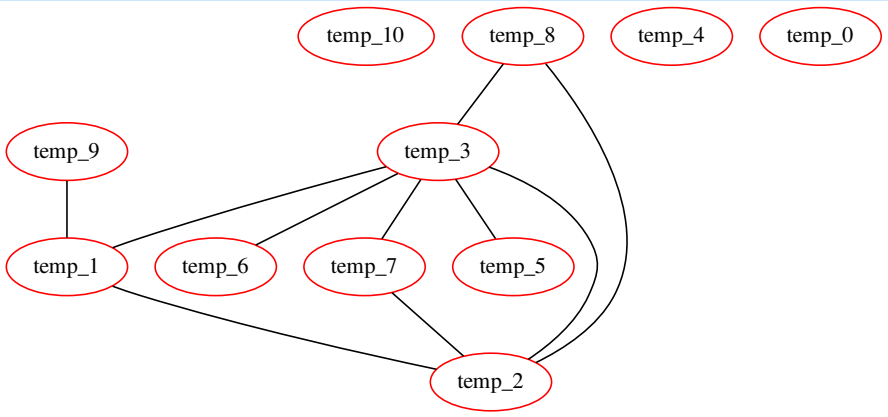
Fill the array in Appendix with the result of the liveness analysis. Each star in a line will mean "the temporary is alive at the entry of this line".

Solution: Il faudra que ce soit cohérent avec le graphe ci-dessous. Attention dans la correction j'utilise le compilateur codé en TP5, donc les temporaires ont des noms *ti*.

Question #5

Draw the interference graph.

Solution:

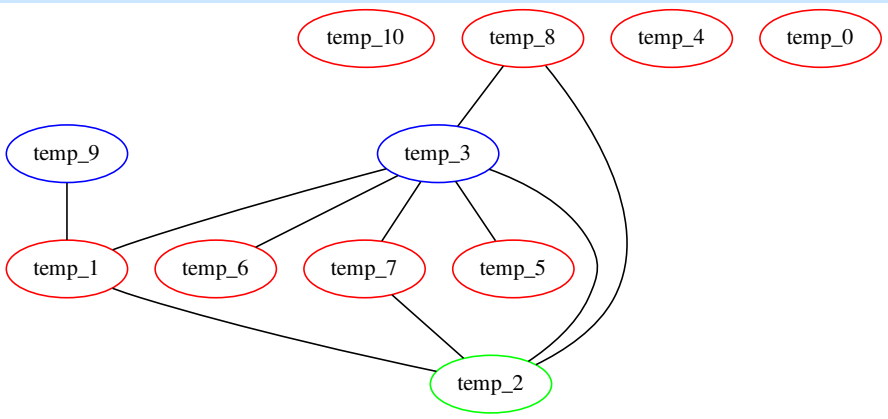


In the rest of the exercice we will use the following notations : — Color 1 is red, and is associated to register R3 ; — Color 2 is blue, and is associated to register R4 ; — Color 3 is green, and is associated to register R5 ;

Question #9

Color the graph with the heuristic of the course and 2 colors. Do not forget to draw the color stack.

Solution: Avec 3 ça donnerait :



Avec deux couleurs on obtient le même coloriage et $t2$ ne peut être colorié.

Question #10

What are the variable(s) to spill ? Explain the spilling process.

Solution: On va donc spiller (ie stocker sa valeur en mémoire) la variable *t*2. Le procédé est décrit dans le cours.

Question #11

Complete the generated code (in Appendix).

Solution:

```
;;Automatically generated LEIA code, MIF08 2017
;;Smart Allocation version
;; stack management
.set r6 stack
        ;; (stat (assignment x = (expr (atom 12)) ;))
        ;; .let temp_4 12
        .LET r3 12
        ;; end .let temp_4 12
        ;; copy temp_3 temp_4
        COPY r4 r3
        ;; end copy temp_3 temp_4
        ;; (stat (assignment y = (expr (expr (atom 3)) + (expr (atom x))) ;))
        ;; .let temp_5 3
        .LET r3 3
        ;; end .let temp_5 3
        ;; add temp_6 temp_5 temp_3
        ADD r3 r3 r4
        ;; end add temp_6 temp_5 temp_3
        ;; copy temp_2 temp_6
        COPY r1 r3
        SUB r0 r6 0
        WMEM r1 [r0]
        ;; end copy temp_2 temp_6
        ;; (stat (assignment z = (expr (expr (atom 4)) + (expr (atom y))) ;))
        ;; .let temp_7 4
        .LET r3 4
        ;; end .let temp_7 4
        ;; add temp_8 temp_7 temp_2
        SUB r0 r6 0
        RMEM r1 [r0]
        ADD r3 r3 r1
        ;; end add temp_8 temp_7 temp_2
        ;; copy temp_1 temp_8
        COPY r3 r3
        ;; end copy temp_1 temp_8
        ;; (stat (assignment t = (expr (expr (expr (atom x)) - (expr (atom y))) + (expr (
    atom z))) ;))
        ;; sub temp_9 temp_3 temp_2
        SUB r0 r6 0
```

```
        RMEM r1 [r0]
        SUB r4 r4 r1
        ;; end sub temp_9 temp_3 temp_2
        ;; add temp_10 temp_9 temp_1
43      ADD r3 r4 r3
        ;; end add temp_10 temp_9 temp_1
        ;; copy temp_0 temp_10
        COPY r3 r3
        ;; end copy temp_0 temp_10
48


;;postlude
jump 0
.align16
53 stackend:
.reserve 42
stack:
```

4 Code generation and Abstract Machines - 60 min

In this exercice we will compile the Mu/mini-while language into a stack abstract machine. *Inspiration and tables from "Semantics with Applications, a Formal Introduction, M.R. Nielson and F. Nielson".*

The abstract machine AM we consider has configurations of the form (c, e, s) where :

— c is the sequence of instructions to be executed,

— e is the evaluation stack (used to evaluate expressions),

— s is a state, used to hold the values of variables.

The instructions of the machine and their semantics are depicted in Table 4.1 (Z are integer values, T Boolean values, tt and ff stands for `true` and `false` respectively) :

$\langle$PUSH-n:c, e, $s\rangle$	$\triangleright$	$\langle c, \mathcal{N}[\![n]\!]$:$e$, $s\rangle$	
$\langle$ADD:c, z_1:z_2:e, $s\rangle$	$\triangleright$	$\langle c, (z_1{+}z_2)$:e, $s\rangle$	if z_1, $z_2 \in \mathbf{Z}$
$\langle$MULT:c, z_1:z_2:e, $s\rangle$	$\triangleright$	$\langle c, (z_1{\star}z_2)$:$e$, $s\rangle$	if z_1, $z_2 \in \mathbf{Z}$
$\langle$SUB:c, z_1:z_2:e, $s\rangle$	$\triangleright$	$\langle c, (z_1{-}z_2)$:e, $s\rangle$	if z_1, $z_2 \in \mathbf{Z}$
$\langle$TRUE:c, e, $s\rangle$	$\triangleright$	$\langle c,$ **tt**:e, $s\rangle$	
$\langle$FALSE:c, e, $s\rangle$	$\triangleright$	$\langle c,$ **ff**:e, $s\rangle$	
$\langle$EQ:c, z_1:z_2:e, $s\rangle$	$\triangleright$	$\langle c, (z_1{=}z_2)$:e, $s\rangle$	if z_1, $z_2 \in \mathbf{Z}$
$\langle$LE:c, z_1:z_2:e, $s\rangle$	$\triangleright$	$\langle c, (z_1{\leq}z_2)$:$e$, $s\rangle$	if z_1, $z_2 \in \mathbf{Z}$
$\langle$AND:c, t_1:t_2:e, $s\rangle$	$\triangleright$		
	$\begin{cases} \langle c, \mathbf{tt} : e, s\rangle & \text{if } t_1{=}\mathbf{tt} \text{ and } t_2{=}\mathbf{tt} \\ \langle c, \mathbf{ff} : e, s\rangle & \text{if } t_1{=}\mathbf{ff} \text{ or } t_2{=}\mathbf{ff}, \text{ and } t_1,\ t_2 \in \mathbf{T} \end{cases}$		
$\langle$NEG:c, t:e, $s\rangle$	$\triangleright$	$\begin{cases} \langle c, \mathbf{ff} : e, s\rangle & \text{if } t{=}\mathbf{tt} \\ \langle c, \mathbf{tt} : e, s\rangle & \text{if } t{=}\mathbf{ff} \end{cases}$	
$\langle$FETCH-x:c, e, $s\rangle$	$\triangleright$	$\langle c, (s\ x)$:e, $s\rangle$	
$\langle$STORE-x:c, z:e, $s\rangle$	$\triangleright$	$\langle c, e, s[x{\mapsto}z]\rangle$	if $z \in \mathbf{Z}$
$\langle$NOOP:c, e, $s\rangle$	$\triangleright$	$\langle c, e, s\rangle$	
$\langle$BRANCH(c_1, c_2):c, t:e, $s\rangle$	$\triangleright$	$\begin{cases} \langle c_1 : c, e, s\rangle & \text{if } t{=}\mathbf{tt} \\ \langle c_2 : c, e, s\rangle & \text{if } t{=}\mathbf{ff} \end{cases}$	
$\langle$LOOP(c_1, c_2):c, e, $s\rangle$	$\triangleright$		
$\langle c_1$:BRANCH$(c_2$:LOOP(c_1, c_2), NOOP$)$:c, e, $s\rangle$			

Table 4.1: Operational semantics for **AM**

Initial configurations always have an empty evaluation stack. A terminal configuration has an empty code component. A stuck configuration is a configuration that cannot evolve any more. A terminating sequence is a finite sequence that cannot evolve any more, thus it can end with a terminal or stuck configuration (non mutually exclusive!). The meaning of a given sequence of functions is defined as usual by :

$$\mathcal{M}[c]s = \begin{cases} s' & \text{if } < c, \varepsilon, s > \triangleright^\star < \varepsilon, e, s' > \\ \text{undef} & \text{otherwise.} \end{cases}$$

The compilation scheme of Mu(While) programs to this machine are given by

$\mathcal{CA}[\![n]\!]$	=	PUSH-n
$\mathcal{CA}[\![x]\!]$	=	FETCH-x
$\mathcal{CA}[\![a_1{+}a_2]\!]$	=	$\mathcal{CA}[\![a_2]\!]$:$\mathcal{CA}[\![a_1]\!]$:ADD
$\mathcal{CA}[\![a_1 \star a_2]\!]$	=	$\mathcal{CA}[\![a_2]\!]$:$\mathcal{CA}[\![a_1]\!]$:MULT
$\mathcal{CA}[\![a_1{-}a_2]\!]$	=	$\mathcal{CA}[\![a_2]\!]$:$\mathcal{CA}[\![a_1]\!]$:SUB
$\mathcal{CB}[\![\texttt{true}]\!]$	=	TRUE
$\mathcal{CB}[\![\texttt{false}]\!]$	=	FALSE
$\mathcal{CB}[\![a_1 = a_2]\!]$	=	$\mathcal{CA}[\![a_2]\!]$:$\mathcal{CA}[\![a_1]\!]$:EQ
$\mathcal{CB}[\![a_1{\leq}a_2]\!]$	=	$\mathcal{CA}[\![a_2]\!]$:$\mathcal{CA}[\![a_1]\!]$:LE
$\mathcal{CB}[\![\neg b]\!]$	=	$\mathcal{CB}[\![b]\!]$:NEG
$\mathcal{CB}[\![b_1 \wedge b_2]\!]$	=	$\mathcal{CB}[\![b_2]\!]$:$\mathcal{CB}[\![b_1]\!]$:AND

Table 4.2: Translation of expressions

$\mathcal{CS}[\![x := a]\!]$	=	$\mathcal{CA}[\![a]\!]$:STORE-x
$\mathcal{CS}[\![\texttt{skip}]\!]$	=	NOOP
$\mathcal{CS}[\![S_1;S_2]\!]$	=	$\mathcal{CS}[\![S_1]\!]$:$\mathcal{CS}[\![S_2]\!]$
$\mathcal{CS}[\![\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2]\!]$	=	$\mathcal{CB}[\![b]\!]$:BRANCH($\mathcal{CS}[\![S_1]\!]$,$\mathcal{CS}[\![S_2]\!]$)
$\mathcal{CS}[\![\texttt{while } b \texttt{ do } S]\!]$	=	LOOP($\mathcal{CB}[\![b]\!]$,$\mathcal{CS}[\![S]\!]$)

Table 4.3: Translation of statements in **While**

Question #1

Compute the operational semantics of the two following sequences of the AM machine :

— P_1 : `PUSH-1:FETCH-x:ADD:STORE-x` with the initial state s where $s(x) = 3$.

— P_2 : `LOOP(TRUE,NOOP)`

Question #2

Compute the generated code for the factorial function :

```
1     y:=1
2     while not(x=1) do
3         y := y * x;
4         x := x-1
5    done
```

Trace its computation from the initial state where $x = 3$.

Question #3

Same question with

```
1     if (x=1) then
```

```
2         y :=2
3      else
4         y:=3
5     done
```

Question #4

Give a code generation rule for the `repeat S until b` construction. The definition has to be compositional, ie by induction on the syntax (and you do not need to extend the instruction set of the machine).

Question #5

Give a code generation rule for the `for x:= a1 to an do S` construction. You may have to introduce a new instruction for the AM machine. If this is the case, also provide its operational semantics.

Now we want to refine the abstract machine in order to be closer to real-life ones. We first define AMbis which differs from AM in that :

— The configurations have the form $< c, e, m >$ where c, e are as in AM but m "the memory" is a finite *list* of values ($m \in Z^*$).

— The instructions `FETCH-X` and `STORE-X` are replaced by instructions `GET-N` and `PUT-N` where n is a natural number (an address).

Question #6

Specify the operational semantics of the AMbis machine. *You may use $m[n]$ to select the nth element of the list m.*

Question #7

Modify the code generation rules so that to transform Mu (while) programs into AMbis. You can assume the existence of a function $env : Var \to N$ that maps variable to their address.

Question #8

Apply this new code generation to the factorial example of Question 2 and trace its computation from initial state where $x = 3$.

The next step is to get rid of the operations `BRANCH` and `LOOP`. The idea is to introduce instruction to define labels and jumping to labels. We thus define AMter which differs from AMbis in that :

— The configurations have the form $< pc, c, e, m >$ where c, e, m are as before and pc is a natural number pointing to an instruction in c ;

— The instructions `BRANCH` and `LOOP` are replaced by the instructions `LABEL-`ℓ `JUMP-`ℓ and `JUMPFALSE-`ℓ where ℓ is a label.

Informal semantics : the labeling has no effect but increments the program counter by 1. `JUMP-`ℓ moves the program counter to the instruction `LABEL-`ℓ if it exists. `JUMPFALSE-`ℓ does the same if an only if the value on the top of the stack is *ff*, else it only increments the program counter by one. In both cases the Boolean value is pop out of the stack.

Question #9

Specify the operational semantics of the AMter machine. *You may use $c[pc]$ to refer to the instruction at line pc.*

Question #10

Modify the code generation rules so that to transform Mu (while) programs into AMter. Be

careful to generate and use unique labels, for instance by having a parameter "next unused label" as an additional parameter to the code generation functions.

Question #11

Apply this new code generation to the factorial example of Question 2 and trace its computation from initial state where $x = 3$.

Question #12

State a correctness property for the compilation process. Explain how you would prove it (5/10 lines).

5 Abstract Interpretation : variation on constants - 20 min

Inspiration from T. Nipkow for TU Munich.

We recall the following principles for abstract interpretation : Sets X of valuations are abstracted by elements of an abstract domain $(\mathcal{A}, \sqsubseteq)$, using an abstraction α and a concretisation γ s.t. $X \subseteq \gamma(\alpha(X))$. Defining an abstract domain amounts to give $\mathcal{A}, \sqsubseteq, \oslash, \bigcirc, \sqcap, \sqcup$ and abstract transfer functions $(+^\sharp, \ldots)$. Then :

— Perform abstract iterations on the CFG.
— If the lattice is of finite height, iterations terminates on a postfixpoint.
— If not : invent a widening operator to ensure finite convergence. Property $(X \sqsubseteq Y) : Y \sqsubseteq X \nabla Y$ and finite chain condition.

In this exercise, we want to design a static analysis that tries to determine whether a variable is -1, 0 or 1, or any other value. The abstract domain consists on the values -1,0,1,`Any`.

Question #1

Properly define the abstraction α, concretisation γ.

Question #2

Define the ordering $\leq$ on the abstract domain.

Question #3

Define the join operator $\sqcup$.

Question #4

Define the abstract operations $+^\sharp$ and $\times^\sharp$.

Question #5

Run the analysis on the following program :

```
1    x=-1;
2    x=x*x + (-1);
3    if (rand(true,false))
4       then x = x+1
5       else x = 10*x
```

Fill and paste section

Report your ANONYMITY NUMBER HERE

.1 Code generation and register allocation