Partial Exam
Compilation and Program Analysis (CAP)
October, 24th, 2016
Duration: 2 Hours

Only one a4 sheet (10pt, recto/verso) is autorized.

Instructions :

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

Exercise #1 ▶ **Attributes (10min)**

Let us consider the following grammar for lists : $L \rightarrow \mathbf{num}L|\{\ L\}L|\varepsilon$

Question #1.1

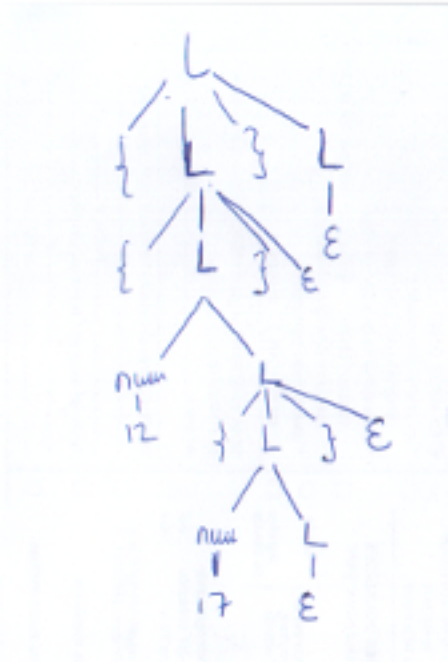
Draw the derivation tree for the string : `{{12{17}}}`.

Question #1.2

Write a syntax-directed attribution (pseudo-code) that computes the product of all numeric elements of a given list.

Solution:

The derivation tree (or Parse Tree) is :



Be careful a parse tree is not an AST, each internal node denotes a rule application where the node is a non terminal and each of its son is either a terminal or a non terminal. Leaves are non terminal

There is no real choice for the attribution, let us invent an attribute `val`, of type int. In the epsilon rule, `L.val<-1`. In the concat rule `L.val<-L_1.val*L_2.val` (rename properly before). In the num rule `L.val<-L_1.val*int(num)`.

For attributions, I expect a type for the attribute as well as the (recursive) rules to compute this attribution.

Exercise #2 ▶ **Hand Assembling (10 min)**

To answer the following questions, you will need the simplified LC3 instruction set depicted in Table 1.

Question #2.1

Assemble by hand the following instruction in LC3 assembly code (with intermediate steps) :

```
1   AND r0 r3 #2 ;
```

Solution: We want the corresponding code for ADD DR ← SR1 + SEXT(Imm5), AND = 0101. r0 = 000, r3 = 011. 2 = 00010. We get : 0101 000 011 1 00010.

Question #2.2

Disassemble by hand the following instruction, given in binary :

```
0000110000000111 ;
```

Solution: BRP nz label=+7 Nothing difficult here.

syntaxe	action	NZP	codage														
			opcode				arguments										
			F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
NOT DR,SR	DR ← not SR	*	1	0	0	1	DR			SR			1	1	1	1	1	1
ADD DR,SR1,SR2	DR ← SR1 + SR2	*	0	0	0	1	DR			SR1			0	0	0	SR2		
ADD DR,SR1,Imm5	DR ← SR1 + SEXT(Imm5)	*	0	0	0	1	DR			SR1			1	Imm5				
AND DR,SR1,SR2	DR ← SR1 and SR2	*	0	1	0	1	DR			SR1			0	0	0	SR2		
AND DR,SR1,Imm5	DR ← SR1 and SEXT(Imm5)	*	0	1	0	1	DR			SR1			1	Imm5				
LEA DR,label	DR ← PC + SEXT(PCoffset9)	*	1	1	1	0	DR			PCoffset9								
LD DR,label	DR ← mem[PC + SEXT(PCoffset9)]	*	0	0	1	0	DR			PCoffset9								
ST SR,label	mem[PC + SEXT(PCoffset9)] ← SR		0	0	1	1	SR			PCoffset9								
LDR DR,BaseR,Offset6	DR ← mem[BaseR + SEXT(Offset6)]	*	0	1	1	0	DR			BaseR			Offset6					
STR SR,BaseR,Offset6	mem[BaseR + SEXT(Offset6)] ← SR		0	1	1	1	SR			BaseR			Offset6					
BR[n][z][p] label	Si (cond) PC ← PC + SEXT(PCoffset9)		0	0	0	0	n		z	p	PCoffset9							
NOP	No Operation		0	0	0	0	0		0	0	0 0 0 0 0 0 0 0 0							
RET (JMP R7)	PC ← R7		1	1	0	0	0 0 0			1 1 1			0 0 0 0 0 0					
JSR label	R7 ← PC ; PC ← PC + SEXT(PCoffset11)		0	1	0	0	1	PCoffset11										

TABLE 1 – LC3 simplified instruction set

EXERCISE #3 ▶ A new case instruction for Mini-While (30min)

The abstract grammar for statements of Mini-While is augmented with a new construction :

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

with the following definition for LS :

$$\begin{array}{rcll} LS & ::= & n : S & \\ & | & n : S, LS & \text{with n integer} \end{array}$$

`LS` is a list of commands labeled by integers ($n \in \mathbb{N}$), separated by colons (',').

Here is the informal semantics of this new construction : the expression e is evaluated in an integer value v. If v is equal to one label n of the case, then the associated command is executed ; else the case behaves like a `skip`. Hence, in the following program :

```
x := 3 ;
y := 2 ;
case x-y of
1 : x := x+y,
0 : x := 2,
3 : y := 0
endcase
```

the command which will be executed is the one labeled by the integer `1` (as the current value of x-y is 1 when the execution flow gets into the case). The memory after the execution of the program will be : $\sigma = [x \mapsto 5, y \mapsto 2]$.

Such a construction is well-formed if all labels are distinct integers.

Question #3.1

Complete the following definition of the B_D attribution that constructs a list of labels defined in the (LS) list, while verifying that all labels are distinct. *Use pseudo-code for lists with the following constructors :* `List.empty()` *constructs an empty list,* `List.add(el,list)` *adds an element in the list (with side-effect), and the predicate* `List.mem(el,list)` *returns true iff the element el is in the list. If there exists a double definition, return an error with an exception.*

$$\begin{array}{rcl} B_D(n : S) & = & ?? \\ B_D(n : S, LS) & = & ?? \end{array}$$

Solution: No real difficulty, for the base case the list $[n]$ is constructed, then for the concat case we search for existance, ...

Be careful to use names, and if you invent data structures, to specify if they are global or an attribute to be returned

Question #3.2

Explain in less than a paragraph how would be the implementation of such two rules inside a ANTLR-Python visitor.

Solution:

To use visitors, we have to instrument the names of the two rules in the .g4 file (say `#baseBD` and `#concatBD` here). In a class that inherits from the grammar's generic visitor, we would have to implement :

```
visitbaseBD(self,ctx)
```

and

`visitconcatBD(self,ctx)`

These two visit methods could return the list defined in the previous question. For the second, we have to make a recursive call to the sublist LS, which can be done by something like

`visitconcatBD(self,ctx.LS())`

I gave some points to people explaining about datastructures. I was however expecting an explanation of the recursive calls' mechanism.

Question #3.3

From now on, we suppose that LS are well-formed. Give natural semantic rules (big steps semantics) for the case construction. To help you, we recall here some of the semantics rules for (some) other Mini-While statements.

$$(x := a, \sigma) \to \sigma[x \mapsto \mathcal{A}[a]\sigma] \qquad (\texttt{skip}, \sigma) \to \sigma$$

$$\text{if } \mathcal{B}[b]\sigma = true : \frac{(S, \sigma) \to \sigma', (\texttt{while } b \texttt{ do } S, \sigma') \to \sigma''}{(\texttt{while } b \texttt{ do } S, \sigma) \to \sigma''}$$

Solution: For instance (needs more explanation) :

$$\text{if } \mathcal{A}[e]\sigma \neq n : (\texttt{case } e \texttt{ of } n : S \texttt{ endcase}, \sigma) \to \sigma$$

$$\text{if } \mathcal{A}[e]\sigma = n : \frac{(S, \sigma) \to \sigma'}{(\texttt{case } e \texttt{ of } n : S \texttt{ endcase}, \sigma) \to \sigma'}$$

$$\text{if } \mathcal{A}[e]\sigma = n : \frac{(S, \sigma) \to \sigma'}{(\texttt{case } e \texttt{ of } n : S, LS \texttt{ endcase}, \sigma) \to \sigma'}$$

$$\text{if } \mathcal{A}[e]\sigma \neq n : \frac{(\texttt{case } e \texttt{ of } n : LS \texttt{ endcase}, \sigma) \to \sigma'}{(\texttt{case } e \texttt{ of } n : S, LS \texttt{ endcase}, \sigma) \to \sigma'}$$

Question #3.4

Apply these rules on the example.

Solution: No difficulty.

Exercise #4 ▶ Mini-While : typing + code generation (30 min)

Here is a program in the Mini-While language seen in the course :

```
x := 8;
y := -1;
if (x<(19+y)) then
   x := 42;
z := x;
```

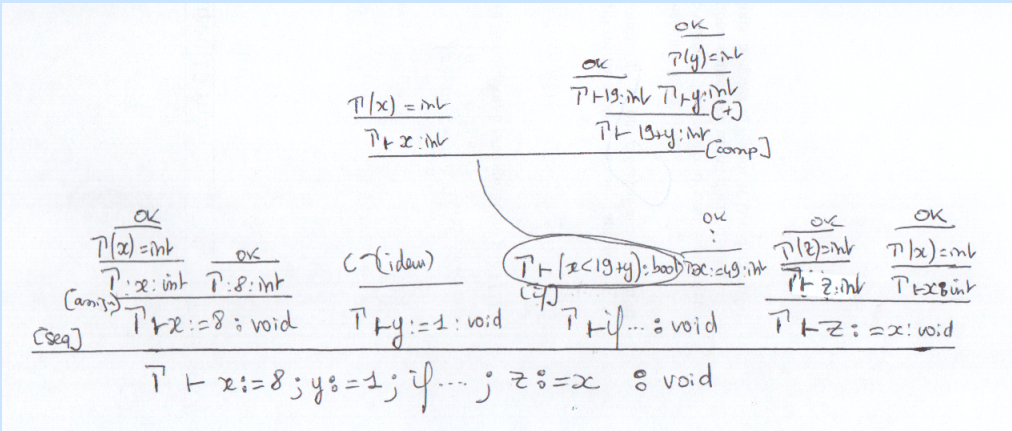
Question #4.1

Show that this program is well-typed, under the following entry typing context : $\Gamma(x) = \Gamma(y) = \Gamma(z) = \texttt{int}$. To help you, we recall here some of the typing rules for expressions and statements :

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 + e_2 : \texttt{int}} \qquad \frac{\Gamma(x) = t \quad t \in \{\texttt{int}, \texttt{bool}\}}{\Gamma \vdash x : t}$$

$$\frac{\Gamma \vdash e : t \quad \Gamma \vdash x : t \quad t \in \{\texttt{int}, \texttt{bool}\}}{\Gamma \vdash x := e : \texttt{void}} \qquad \frac{\Gamma \vdash b : \texttt{bool} \quad \Gamma \vdash S : \texttt{void}}{\Gamma \vdash \texttt{while } b \texttt{ do } S \texttt{ done} : \texttt{void}}$$

Solution: Here it is :



Question #4.2

Generate the LC3 3-address code (LC3 + temporaries/virtual registers) for the given program. *Recursive calls, auxiliary temporaries, code, must be separated and clearly described.* To help you we provide some generation rules in Figure 1 and 2.

Solution: Here it is :

(constant expression) c	`#not valid if c is too big`
	`dr <-newTemp()`
	`code.add(InstructionAND(dr, dr, 0))`
	`code.add(InstructionADD(dr, dr, c))`
	`return dr`

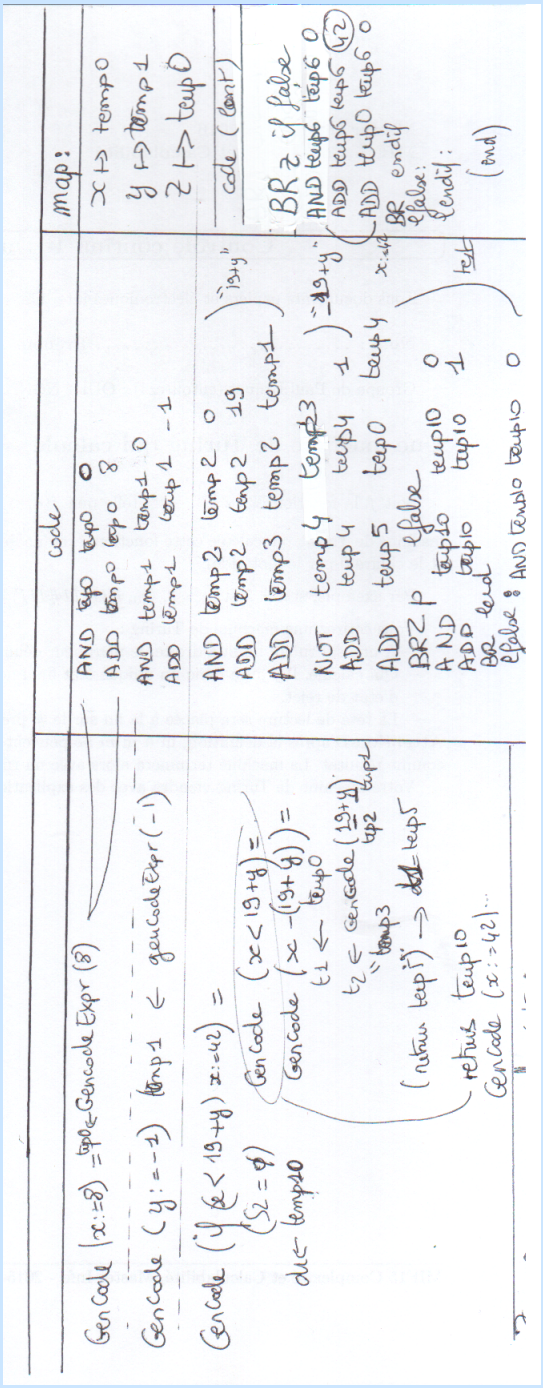
(expression `e1 < e2`)	`dr <-newTemp()`
	`t1 <- GenCodeExpr (e1-e2)  #last write in register`
	`(lfalse,lend) <- newLabels()`
	`code.add(InstructionBRzp(lfalse))  #if =0 or >0 jump!`
	`code.add(InstructionAND(dr, dr, 0))`
	`code.add(InstructionADD(dr, dr, 1))  #dr <- true`
	`code.add(InstructionBR(lend))`
	`code.addLabel(lfalse)`
	`code.add(InstructionAND(dr, dr, 0))  #dr <- false`
	`code.addLabel(lend)`
	`return dr`

FIGURE 1 – 3-address code generation rules 1/2

(Stm) `x := e`	`dr <- GenCodeExpr(e)`
	`#a code to compute e has been generated`
	`if x has a location loc:`
	`  code.add(instructionADD(loc,dr,0))`
	`else:`
	`  storeLocation(x,dr)`

(Stm)`if` b `then` $S1$ `else` $S2$	`dr <-GenCodeExpr(b)  #dr is the last written register`
	`lfalse,lendif=newLabels()`
	`code.add(InstructionBRz(lfalse) #if 0 jump to execute S2`
	`GenCodeSmt(S1)  #else (execute S1`
	`code.add(InstructionBR(lendif)) #and jump to end)`
	`code.addLabel(lfalse)`
	`GenCodeSmt(S2)`
	`code.addLabel(lendif)`

FIGURE 2 – 3-address code generation rules 2/2



I was expecting "real" code generation, thus the recursive calls have to be instantiated and produce real code. I gave very few points to code coming with no exaplanation. Be careful to the application of the while rule, there is some code redundancy in computing the test and then branching.

Exercise #5 ▶ A new Type System for Mini-While (40 min)

Adapted from oldies used in Grenoble a long time ago.

In this exercice, we replace the abstract grammar for the mini-while expressions by

$$\begin{array}{rcll} nexp & ::= & p & \textit{positive or nul constant} \\ & | & n & \textit{stricly negative constant} \\ & | & x & \textit{variable} \\ & | & nexp + nexp & \textit{addition} \\ & | & nexp - nexp & \textit{substraction} \\ & | & nexp \times nexp & \textit{multiplication} \\ & | & \ldots & \end{array}$$

for arithmetic expressions (now there is a distinction between negative and positive constants), and for boolean expressions :

$$\begin{array}{rcll} bexp & ::= & true & \textit{constant} \\ & | & false & \textit{constant} \\ & | & bexp \text{ or } bexp & \textit{logical or} \\ & | & e < e & \textit{comparaison} \end{array}$$

Statements are unchanged :

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

Now we define three types for numerical expressions : `Pos, Neg, Int`, a type for boolean expressions `ok` The idea is now to propagate the sign information for arithmetic expression : the type is its sign (or Int if we cannot conclude). Γ denotes now the typing environment.

We give rules for contants, variables, addition :

$$(\Gamma, p) \longrightarrow \texttt{Pos} \qquad (\Gamma, n) \longrightarrow \texttt{Neg} \qquad (\Gamma, x) \longrightarrow \Gamma(x)$$

$$\frac{(\Gamma, a_1) \longrightarrow \texttt{Pos} \quad (\Gamma, a_2) \longrightarrow \texttt{Pos}}{(\Gamma, a_1 + a_2) \longrightarrow \texttt{Pos}} \qquad \frac{(\Gamma, a_1) \longrightarrow \texttt{Neg} \quad (\Gamma, a_2) \longrightarrow \texttt{Neg}}{(\Gamma, a_1 + a_2) \longrightarrow \texttt{Neg}}$$

$$\frac{(\Gamma, a_1) \longrightarrow \texttt{Neg} \quad (\Gamma, a_2) \longrightarrow \texttt{Pos}}{(\Gamma, a_1 + a_2) \longrightarrow \texttt{Int}}$$

These rules have the following meaning : adding two positive integers gives a positive integer, but adding a positive and a negative integer gives an integer (we cannot conclude. . .).

Rules for boolean expressions always give the type `ok` if the expression is well typed :

$$(\Gamma, \texttt{true}) \longrightarrow ok \qquad \frac{(\Gamma, a_1) \longrightarrow \tau \quad (\Gamma, a_2) \longrightarrow \tau}{(\Gamma, a_1 = a_2) \longrightarrow ok} \qquad \frac{(\Gamma, b_1) \longrightarrow ok \quad (\Gamma, b_2) \longrightarrow ok}{(\Gamma, b_1 \wedge b_2) \longrightarrow ok}$$

Finally, here are rules for statements :

$$\frac{(\Gamma, a) \longrightarrow \tau}{(\Gamma, x := a) \longrightarrow \Gamma[x \mapsto \tau]} \qquad (\Gamma, skip) \longrightarrow \Gamma \qquad \frac{(\Gamma, S) \longrightarrow \Gamma'}{(\Gamma, \texttt{while}\ b\ \texttt{do}\ S) \longrightarrow \Gamma \sqcup \Gamma'}$$

Let us clarify a bit the notation $\sqcup$. For a given variable, for the while, we have to "merge" the information coming from before the while and the one coming from S. If they are both `Pos`, or both `Neg`, then the result of the merge is the very same type. If the two values do not coincide, then the merge of the two types are `Int`. Then the definition of $\sqcup$ is extended pointwise.

The while rule itself needs a little bit more explanation. This rule uses the fact that, in this particular type system, you cannot infer any more information by applying more than one loop : this should have been proven before, and comes from the fact that a given variable cannot changes its type more that twice. This is not the case in general typing systems, in the more general case we should have to test that Γ' is stable by S.

Question #5.1

Type the expression : $(-2 + x) + 8$ under the context $\Gamma(x) = \texttt{Neg}$.

Solution: Type tree like in the course. Do not forget to close the leaves properly.

Question #5.2

Give rules for substraction and multiplication.

Solution: For sub, two cases propagate information : `Neg - Plus` and `Plus - Neg` :

$$\frac{(\Gamma, a_1) \to \texttt{Neg} \quad (\Gamma, a_2) \to \texttt{Pos}}{(\Gamma, a_1 - a_2) \to \texttt{Neg}} \qquad \frac{(\Gamma, a_1) \to \texttt{Pos} \quad (\Gamma, a_2) \to \texttt{Neg}}{(\Gamma, a_1 - a_2) \to \texttt{Pos}}$$

$$\frac{(\Gamma, a_1) \to \texttt{Neg} \quad (\Gamma, a_2) \to \texttt{Neg}}{(\Gamma, a_1 - a_2) \to \texttt{Int}} \qquad \frac{(\Gamma, a_1) \to \texttt{Pos} \quad (\Gamma, a_2) \to \texttt{Pos}}{(\Gamma, a_1 - a_2) \to \texttt{Int}}$$

For multiplication the result is more precise :

$$\frac{(\Gamma, a_1) \to \texttt{Neg} \quad (\Gamma, a_2) \to \texttt{Neg}}{(\Gamma, a_1 * a_2) \to \texttt{Pos}} \qquad \frac{(\Gamma, a_1) \to \texttt{Pos} \quad (\Gamma, a_2) \to \texttt{Pos}}{(\Gamma, a_1 * a_2) \to \texttt{Pos}}$$

$$\frac{(\Gamma, a_1) \to \texttt{Neg} \quad (\Gamma, a_2) \to \texttt{Pos}}{(\Gamma, a_1 * a_2) \to \texttt{Neg}} \qquad \frac{(\Gamma, a_1) \to \texttt{Pos} \quad (\Gamma, a_2) \to \texttt{Neg}}{(\Gamma, a_1 * a_2) \to \texttt{Neg}}$$

there is a slight problem for multiplication with `Pos`, as the operand might be 0, `Neg*Pos!` should give `Int`. I gave the whole number of points for both solutions.

Question #5.3

Give rules for the sequence and test (if).

Solution: For seq, the typing informations are transmitted from the first to the second part :

$$\frac{(\Gamma, S_1) \to \Gamma_1 \quad (\Gamma_1, S_2) \to \Gamma_2}{(\Gamma, S_1; S_2) \to \Gamma_2}$$

For if then else, we have to merge the information from the two branches :

$$\frac{(\Gamma, b) \to ok \quad (\Gamma, S_1) \to \Gamma_1 \quad (\Gamma, S_2) \to \Gamma_2}{(\Gamma, \text{if } b \text{ then } S_1 \text{ else } S_2) \to \Gamma_1 \sqcup \Gamma_2}$$

Since I had not defined the meaning of $\sqcup$ properly, this question was not very interesting. However, even with an informal notion of "merge information", the environnement before the if should not be copied at the bottom right hand side.

Given a type environment Γ and a memory σ (like in the SOS rules of the course, the memory assigns values to variables), we define the following relation :

$$(\Gamma, \sigma) \in \mathcal{R} \text{ iff } \forall x \cdot [(\Gamma(x) = \texttt{Pos} \wedge \sigma(x) \geq 0) \vee (\Gamma(x) = \texttt{Neg} \wedge \sigma(x) < 0) \vee \Gamma(x) = \texttt{Int}]$$

Question #5.4

Show that for all $(\Gamma, \sigma) \in \mathcal{R}$, if $(\Gamma, a) \longrightarrow \texttt{Pos}$ then $\mathcal{A}[a]\sigma \geq 0$ and if $(\Gamma, a) \longrightarrow \texttt{Neg}$, then $\mathcal{A}[a]\sigma < 0$.

Solution: By induction on a :

- if a is a positive constant, then $(\Gamma, a) \rightarrow \texttt{Pos})$ and $\mathcal{A}[a]\sigma = p \geq 0$.
- same for a negative constant.
- if x is a variable, either she is positive or null, either she is positive, which is given by the value during execution : $\mathcal{A}[x]\sigma$
- if $a = a_1 + a_2$, WLOG let us suppose that $(\Gamma, a) \rightarrow \texttt{Pos}$. From the typing rule we have $(\Gamma, a_1) \rightarrow \texttt{Pos}$ and $(\Gamma, a_2) \rightarrow \texttt{Pos})$, thus we can imply the induction hypothesis on both these arguments, thus $\mathcal{A}[a_1]\sigma \geq 0$ and $\mathcal{A}[a_2]\sigma \geq 0$. Then, as $\mathcal{A}[a_1 + a_2]\sigma = \mathcal{A}[a_1]\sigma + \mathcal{A}[a_2]\sigma$, $\mathcal{A}[a_1 + a_2]\sigma$ is positive.
- all the other cases are similar.

Question #5.5

By induction, show that if $(\Gamma, \sigma) \in \mathcal{R}$ and $(S, \sigma) \longrightarrow \sigma'$, then there exists Γ' such that $(\Gamma, S) \longrightarrow \Gamma'$ and $(\Gamma', \sigma') \in \mathcal{R}$.

Solution: Left to the reader. The only difficulties are tests and while. For the test, the stability comes from the union $\sqcup$, and for the while, there is more work to do ...

Question #5.6

What goes that mean for our typing system ?

Solution: This means each time we have a transition in the concrete world we have a transition in the abstract world, this is progression. Together with question 5.4 this means that the type system is safe.